Specification-By-Demonstration: The ViCCS Interface

Steven Minton and **Andrew Philpot**
USC/ISI, 4676 Admiralty Way
Marina Del Rey, CA, 90292, USA
{minton,philpot}@isi.edu

Shawn Wolfe
NASA ARC, Caelum Research, MS 269-2
Moffett Field, CA, 94035, USA
shawn@ptolemy.arc.nasa.gov

Abstract

Our goal is to make combinatorial problem-solving technology available to users who have no formal training in AI or OR. To achieve this aim, we have built a system, ViCSS (Visual Constraint Specification System) that enables users to specify problems graphically. To simplify the specification process, ViCSS relies heavily on programming-by-demonstration (PBD) techniques. We believe, however, that the key to making PBD succeed for our application is to constrain the generalization process so that the system makes only "simple", easy-to-understand generalizations. In this paper we describe several design decisions that enable us to achieve this end, and illustrate them in the context of an extended example.

Introduction

MULTI-TAC (Minton 1993b) is a system for synthesizing combinatorial search programs. To use MULTI-TAC, one must provide a description of the problem constraints in predicate calculus. This limits the practical use of the system, since most potential users of the system are not fluent in predicate calculus. We refer to this as the *specification bottleneck*.

The specification bottleneck is an important issue for all existing combinatorial solvers, not just MULTI-TAC. In fact, MULTI-TAC is relatively easy to use, compared to previous approaches to combinatorial problem solving. Most of the previous methods/systems originated in the operations research (OR) community, and they often rely on mathematical, equation-oriented specification languages that untrained users find obscure. For instance, one common approach to specifying such problems is to represent them using sets of equations over linear and integer variables.

One of our goals is to make combinatorial problem-solving technology available to users who have no formal training in AI or OR. To achieve this aim, we have been developing a front end to the MULTI-TAC system that enables users to specify problems graphically. We refer to this front end as ViCSS (Visual Constraint Specification System). To simplify the specification process, ViCSS relies heavily on programming-by-demonstration techniques. We believe, however, that one key to making programming-by-demonstration succeed for our application is to constrain the generalization process so that the system makes only "simple", easy-to-understand generalizations. To achieve this end, we rely on several ideas. First, the user demonstrates only a small "chunk" of the problem at each step. Second, the system is restricted to a small (almost trivial) hypothesis space. Third, the generalization language is closely tied to the way that data is graphically displayed. And fourth, we enable the user to algebraically manipulate the graphical structures so that the required generalizations can be expressed in the hypothesis space. In effect, this last capability enables to the user to engineer the description language so as to make the learning problem simpler.

Specifying Problems

MULTI-TAC is designed for a scenario where a combinatorial search problem must be solved routinely, such as a scheduling application where each week a set of manufacturing tasks is assigned to a set of workers. The system takes as input a specification of the generic problem and a set of representative problem instances. Because MULTI-TAC employs a library of generic algorithms and heuristics, it can be used with a large variety of combinatorial problems. The user need not worry about about selecting an algorithm, since the system can automatically synthesize a specialized algorithm appropriate for the application using the library. Specifically, the system's objective is to synthesize a program that is as efficient as possible for the instance population.

In order to present a problem to MULTI-TAC it must be formalized as an *integer constraint problem*, that is, as a set of constraints over a set of integer variables. A solution exists when all the variables are assigned a value such that the constraints on each variable are satisfied. (The system can also solve constrained *optimization* problems, but we will only consider *satisfaction* problems in this paper since they are a bit simpler to specify.)

For example, consider an NP-complete problem, daily crew scheduling. An instance of this problem consists of crew with C workers, each with a set of skills, a set of T tasks that must be accomplished by workers with appropriate skills, and a set of time slots for accomplishing those tasks. For simplicity, we will assume there are a 15 time slots, each one hour long. Each job requires k man-hours, where k is an integer, but the hours can be put in at any time during the day, and more than one person can work on the job. The problem is to assign each job to a time slot, and assign an appropriate person to each job, subject to the constraint that workers can only be assigned to one job at a time. There may be additional constraints as well, for instance, we might require that no person works more than 8 hours.

First, let us consider how this problem would be directly formulated in MULTI-TAC as a constraint satisfaction problem. Then we will show how a user interacts with VICSS to formulate the same problem, demonstrating the utility of our "specification-by-demonstration" approach.

In a MULTI-TAC problem specification, constraints are described using a first-order, typed predicate logic. The language is relatively expressive, allowing for full first-order quantification plus set and bag creation.

To formulate the crew scheduling problem in MULTI-TAC, we define three types: Crew, Hour and Task. Specifically, we represent crew members by a number between 1 and C, where C is the total number of crew members. We represent each one-hour time slot by a number between 1 and 15, and the Tasks by a number between 1 and T. Then for each crew member c, hour h, and task t we create a relation, **(scheduled** c h t s**)**, mapping c, h and t to a boolean variable s that is true iff the task is assigned to that crew member during that time slot. We also define a relation **(Has-Skill** c,t**)** that is true iff crew member c has the skills for performing task t. We also define a functional relation **(Num-Hours** t, n**)**, which indicates the number of hours required for each task.

In MULTI-TAC, the constraints defining an acceptable schedule can be stated as follows:

- Each job must be scheduled for exactly the required number of hours.

```
(forall t suchthat (Task t)
  (exists reqhours (Num-Hours t reqhours)
    (exists bag (bag-of s suchthat
                  (exists h suchthat (Hour h)
                    (exists c suchthat (Crew c)
                      (scheduled c h t s))))
      (bag-size bag reqhours))))
```

- No one can be assigned more than a single job at a time.

```
(forall c suchthat (Crew c)
  (forall h suchthat (Hour h)
    (forall t1 suchthat (scheduled c h t1 true)
      (forall t2 suchthat (scheduled c h t2 true)
        (equal t1 t2)))))
```

- No one can work more than 8 hours.

```
(forall c suchthat (Crew c)
  (exists tasks (bag-of t suchthat
                  (exists h suchthat (Hour h)
                    (scheduled c h t true)))
    (exists numtasks (bag-size tasks numtasks)
      (less-than-or-equal numtasks 8))))
```

- A person can be assigned to a job only if he has the required skills.

```
(forall c suchthat (Crew c)
  (forall h suchthat (Hour h)
    (forall t suchthat (scheduled c h t true)
      (Has-Skill c t))))
```

In addition to the generic constraints above, there may be constraints that are particular to a problem instance. For example, because the crew member "McCoy" is grumpy and likes to work alone, we might require that if McCoy is assigned to a task, no one else can be working on that task at the same time. This constraint would be specified as follows:

- No one can work with McCoy:

```
(forall h suchthat (Hour h)
  (forall t suchthat (scheduled McCoy h t true)
    (forall c suchthat (scheduled c h t true)
      (equal McCoy c))))
```

Once the problem has been specified, MULTI-TAC will synthesize an algorithm for solving the problem. The user can provide examples of problem instances (e.g., specific instances of the crew scheduling problem) in order to help Multi-TAC find a relatively efficient algorithm. As described in previous papers (Minton 1993a; 1993b) MULTI-TAC relies heavily on machine learning techniques to guide the synthesis process. In experiments with some NP-hard problems, we found that MULTI-TAC produced code that performed on par with, and occasionally better than, code produced by human programmers (Minton 1993b).

ViCSS Interface

We will now describe how the same crew scheduling problem is formulated using VICSS, in order to illustrate our Specification-by-Demonstration approach.

VICSS provides several varieties of graphical structures that can be used to display data, including lists, grids, graphs and sequences. For the crew scheduling problem, the schedule itself is represented as a grid, as shown in Figure 1. We use lists to represent the crew members and the time slots (the hours); when the user sets up the grid, he names the grid "Schedule", and he specifies that the axes of the grid should be labeled using these two lists. The tasks also are entered in a list. (These lists are iconified on the screen in Figure 1. Note also that the schedule in the Figure has been partially filled out.)

When the user sets up the grid, he specifies that the grid should be filled out (by the solver) using the list of tasks. Lists can contain duplicates in VICSS, and

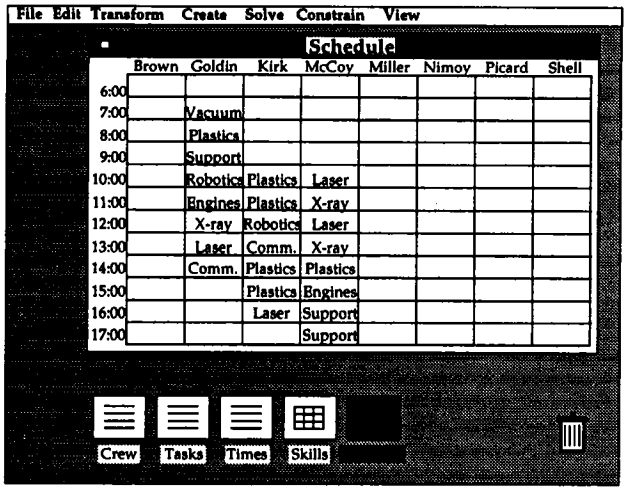

Figure 1: *Partial Schedule*

we can make use of this flexibility as follows: if a task requires k hours, we can enter it k times on the list of tasks. Because of this choice of representation, there is no need to explicitly enter the first constraint described earlier "Each task must be scheduled for exactly the required number of hours"; This constraint is implicitly enforced by the fact that every item of the list of tasks must be placed on the schedule.

When the grid is set up, the system asks the user several questions, including he following: "Can the cells of the grid be assigned more than one task?" For this problem, the answer is "no", since each person can only work on one task at a time. Thus, the second constraint above is also implicitly enforced by the way the problem is set up.

The remaining constraints are described by "demonstrating" them to the system. To demonstrate the constraints, the schedule must be partially filled out. (The schedule shown in Figure 1 has been partially filled out for exactly this purpose.) Either the user can fill out some of the schedule in order to demonstrate the constraint, or the system can start filling out the schedule and the user can interrupt the system when he sees a violated constraint.[1] A constraint can be demonstrated using either a *positive demonstration* or a *negative demonstration*; in the former case the user illustrates the constraint with an example where the constraint holds, in the latter case the user demonstrates a situation where the constraint is violated. We will only consider positive demonstrations here. Negative demonstrations are conducted in a similar manner.

All constraints are specified in two phases. First the user creates the graphical structures that he needs to demonstrate the constraint. Second, the user selects specific examples within the graphical structures and illustrates how these examples relate to each other.

[1] If the user starts filling out the schedule to demonstrate the constraint, he can later erase the parts he has filled in, once the constraint has been specified.

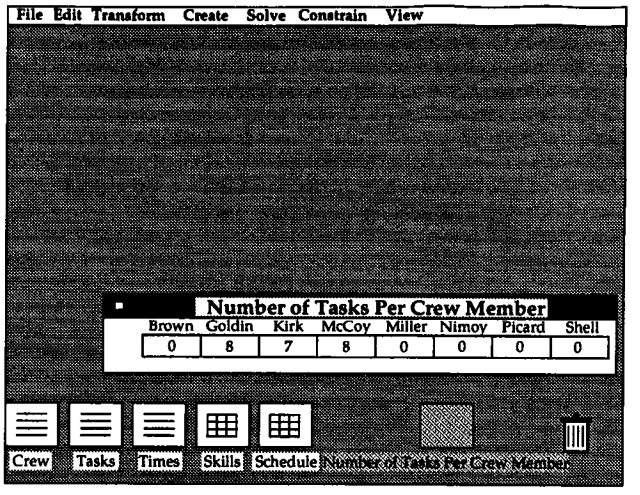

Figure 2: *Counting Tasks*

Consider the constraint "No one can work more than eight hours". In the first phase, the user creates a new grid that indicates the number of hours worked by each crew member. To do so, the user applies a short series of algebraic manipulations to the original schedule grid. He begins by collapsing the columns by applying the "projection" operator to the grid, so that each cell now lists the tasks for a given crew member. The user labels this grid "**Tasks per crew member**". (After each algebraic manipulation, a new grid is created, and the previous grid is iconified and saved for future use.) Then the user applies the "number of" operator to this grid, which calculates the the number of items in each cell. The end result is a new grid indicating the number of tasks currently assigned to each crew member, as shown in Figure 2. The user labels this grid "**Number of Tasks Per Crew Member**." (The other grids are not displayed on this screen; they are iconified.)

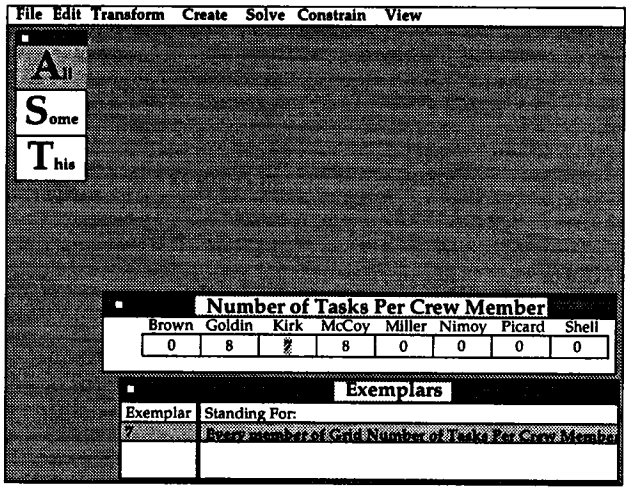

Figure 3: *Selecting an Example*

In the second phase, the items in this grid are con-

strained to be less than or equal to 8. This can be accomplished as follows. The user selects an item in the grid as an example. When the user picks the example, he also selects a quantification mode, either ALL (universal quantification), SOME (existential quantification) or THIS (no quantification). Once the example is selected the system guesses the "scope" of the example. In this case, the system infers that the appropriate generalization is "every item in the grid **Number of Tasks per Crew Member**", as shown in Figure 3. The user now enters the constraint by selecting a relation (either $=, \neq, \leq, \geq, >$, or $<$) In this case, the user selects $\leq$, "less than or equal to". The $\leq$ template pops up, and the user indicates that the example item must be less than or equal to the number 8, completing the process of specifying the constraint. The system then displays the constraint in pseudo-english "Every item in the grid **Number of Tasks per Crew Member** is less than or equal to 8", as shown in Figure 4.

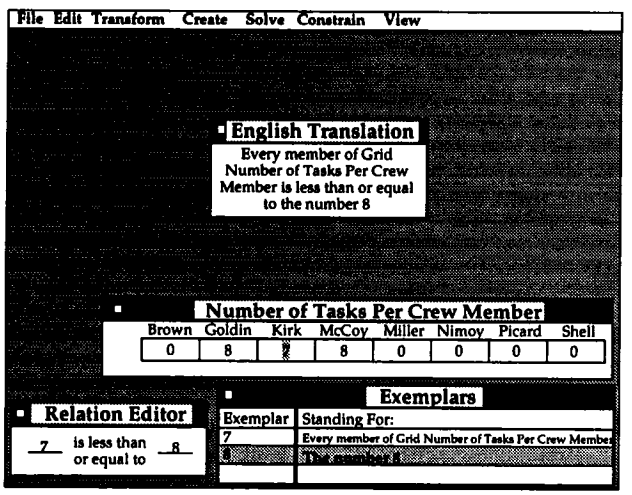

Figure 4: *Constraint is Described*

As this scenario illustrated, VICSS employs both "programming-with-example" and "programming-by-example", as defined by Myers (Myers 1992). Programming-with-examples is used in the first phase, when the user applies algebraic operations to create new graphical structures, such as the grid **Number of Tasks per Crew Member**. Here the user is explicitly applying operators (i.e., programming) and the system does not need to make any inductive inferences. Although the grids have been partially filled-out, the information in the grids serves no functional purpose other than to aid the user in understanding the abstract operations he applies. Programming-by-example is used in the second phase, when the user specifies a constraint by demonstrating with an example. The system must infer the scope that the example is intended to represent.

We note that the system does not have to infer quantifiers, since this information is explicitly provided by the user when he selects an example and specifies the quantification mode (ALL, SOME or THIS). In the scenario above, the example stood for "every item in the grid **Number of Tasks per Crew Member**" since the user employed the ALL quantification mode. If the SOME mode had been used instead, the example would have stood for "Some item in the grid **Number of Tasks per Crew Member**". The THIS quantification mode is to indicate that the system shouldn't generalize. Had the user employed THIS mode in the above scenario, the system would have simply inferred that the item in the column headed by "Goldin" should be less than or equal to 8.

Because the quantifiers are explicitly specified, the generalization process is used only for determining an example's scope, i.e., what the example "stands for". In order to increase the system's transparency and ease-of-use, the hypothesis space that VICSS's considers is very restricted; consequently the inductive leaps the system involve only minimal "guessing".

For example, only the following five types of objects can be referred to in a grid: an item (an entry in a cell), a cell, a row, a column, or a header (of a row or column). There is a separate generalization language for each of these objects. For example, the possible generalizations for an item (i.e., an entry in a cell of the grid) includes the concepts listed below. Each of these concepts corresponds to a "scope" that an exemplar item could represent.[2]

- an item in the grid
- an item in the currently selected cell
- an item in the currently selected row
- an item in the currently selected column
- an item in the column of the currently selected column header
- an item in the row of the currently selected row header
- an item in a cell of another grid that corresponds to the currently selected cell in the current grid.
- another item in the same cell as the currently selected item
- another item in the same row as the currently selected item
- another item in the same column as the currently selected item

Note that all but the first three concepts are "relative concepts", since they are specified relative to another object. As we will see, during the inference process, if there is more than one concept (i.e., hypothesis) that is a candidate generalization of the current example, the system will select the most specific hypothesis consistent with the current context.[3] Formally, the con-

[2]Admittedly some of these concepts are a bit awkward to describe at an abstract level. They make more sense in a graphical context. As we will see in the next scenario, the concepts are easy to understand when presented in context.

[3]If there is no unique concept that is the most specific, the user is queried as to his preference.

text consists of a pair $\langle G, O \rangle$, where G is the graphical structure that is currently the "focus structure" and and O the last object that was "selected".

In order to illustrate the role that the context plays in the generalization process, we will consider a more complex scenario, where the user specifies the constraint "No one can work with McCoy" (i.e., if McCoy is working on a task during some time slot, no one else can work on that same task during that time slot). The user will use the same partially-filled out schedule to demonstrate the constraint. In other words, the grid labeled "Schedule" will be the "focus structure" during this demonstration. (The concept of "focus" is similar to that used in many window systems, such as X. In VICSS the user simply clicks on a structure to change the focus.) To demonstrate the constraint, the user chooses an example of a task in McCoy's column and an example of a task at the same time in another column, and specifies that these must not be equal.

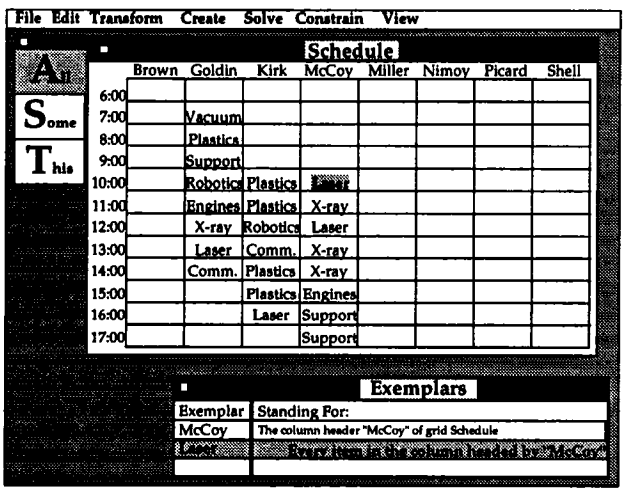

Figure 5: *Selecting an Item in McCoy's Column*

Let's look at this process in more detail. The user begins by selecting the column header "McCoy", using the THIS quantification mode, indicating that McCoy simply stands for itself ("the column header McCoy"). Then he selects an arbitrary item in McCoy's column, Laser, using the ALL quantification mode. At this point there are at least two consistent generalizations of this example. One possible generalization is that Laser is "an item in the column headed by McCoy". Another possible interpretation is that Laser is "an item in the grid `Schedule`". As shown in Figure 5 (see the "Exemplars" window), the first interpretation is selected, because it is the most specific generalization that is consistent with the context. Here, the context is $\langle$Grid:Schedule, Column-header:McCoy$\rangle$, since the Grid called "Schedule" is the current focus structure, and the column header "McCoy" was the last object selected. Since the quantification mode ALL was used, the system notes that the example "Laser" stands for

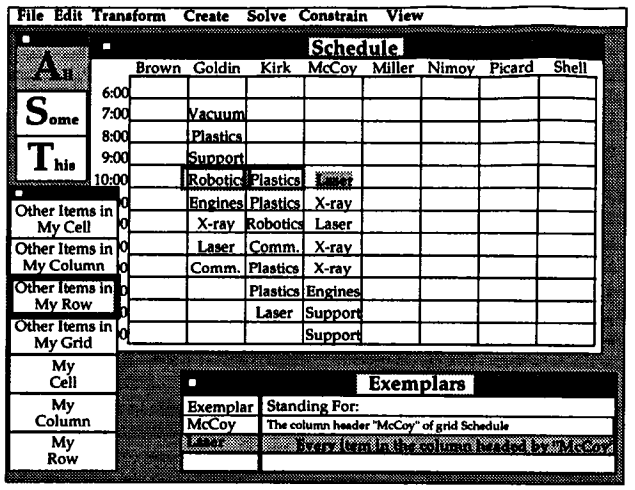

Figure 6: *Using the Generalization Palette*

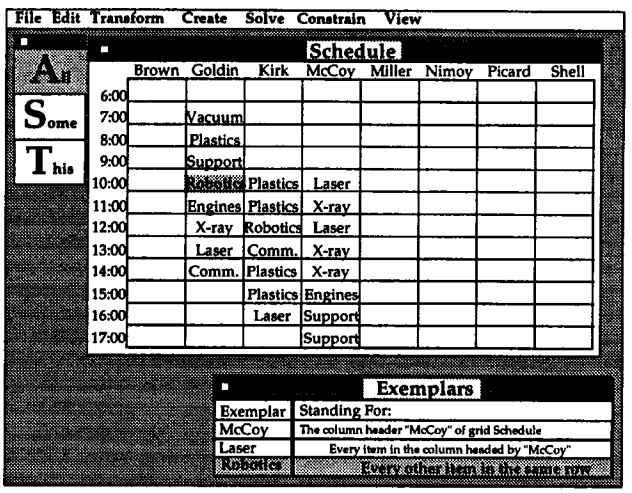

Figure 7: *Selecting an Item in the Same Row*

"every item in the column [headed by McCoy]".

Since the hypothesis space might not be obvious to users, the system includes an optional palette enabling users to explicitly control the generalization process. This palette provides additional feedback for the user; Before selecting an example, the user can pull up the palette to see the possible concepts that can be described in the current context. Clicking on a concept in the palette will cause all the possible exemplars of that concept (that are currently on the screen) to become highlighted. The user can then select one of these as his exemplar of the concept, and continue on as usual. (As the user becomes more familiar with the interface, we presume that he will discontinue using on the palette.)

To illustrate this, let us continue with our scenario. In Figure 6, the user has selected "Laser", as described above, and now the palette shows the possible concepts consistent with the current context. Using the palette the user can display the examples consistent with the concept "every other item in my [i.e. the current] row".

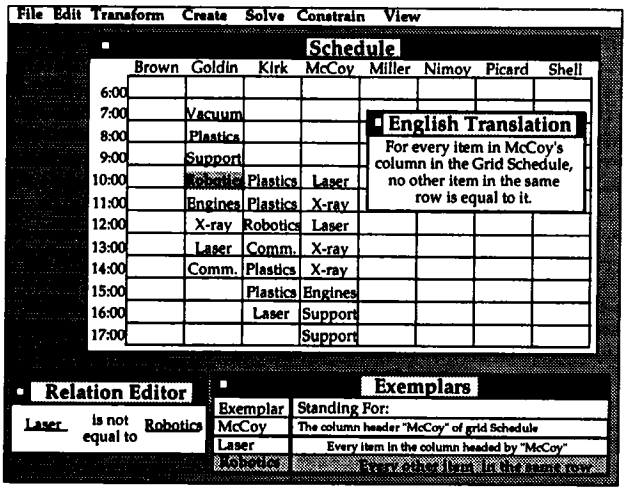

Figure 8: *Constraint is Described*

He selects one of these, the item "Robotics", as shown in Figure 7, again using the ALL quantification mode. The system reports that the generalization is "another item in the same row". No guessing is involved, since the user made the choice explicitly via the palette. Note that if the palette had *not* been used and the user had directly selected "Robotics", there would be several plausible generalizations. Nevertheless, the system would have guessed the right concept since the context (which is ⟨Grid:Schedule, Item:Laser⟩) provides the appropriate bias.

At this point, the user can finish specifying the constraint by invoking the $\neq$ template and stating that *Laser* $\neq$ *Robotics*, as shown in Figure 8. The system states the constraint as follows: For every item in McCoy's column in the grid "Schedule", no other item in the same row is equal to it.[4]

The two scenarios that we presented both involved describing a constraint within a single graphical structure. However, one can also describe constraints involving more than one structure. For instance, consider the constraint "A person can be assigned to a task only if he has the required skills". To specify this, the user must use both the `Schedule` and the `Has-Skills` grids. (Assume that the latter lists the permissible tasks for each crew member.) The constraint is specified simply by choosing an arbitrary item in the grid `Schedule`, such as Laser in McCoy's column, and then, after changing the focus structure to `Has-Skills`, indicating that some item in McCoy's column in this second grid must be equal to "Laser" (i.e., McCoy must have the skill required for the "laser" task).

Discussion

In the Introduction, we stated that VICSS was designed so that the system makes only "simple", easy-to-understand generalizations. In particular, there are four design decisions that we believe are important:

- Constraints demonstrated piecemeal: Not only is each constraint demonstrated separately, but each constraint is demonstrated by showing the relations between several objects. Thus, the generalization process is only used to infer the individual relations between the objects.
- Small hypothesis space: The generalization language is very restrictive, so that the hypothesis space is small.
- Graphically-oriented generalization language: The concepts in the generalization language are closely tied to the graphical display of the data.
- User-defined description language: The user can algebraically manipulate the graphical structures, in effect, "engineering" the domain description language. This simplifies the demonstrations that the user is required to make.

Let consider this four points, and how they are illustrated by the scenarios we described. The first point was most clearly illustrated in our second scenario ("No one can work with McCoy") where the user had to demonstrate the relationship between several items to describe the constraint. Because the problem specification is broken down into separate constraints, and each constraint is broken down into descriptions of a set of objects, the generalization problem at each step is relatively simple. On the other hand, this might be considered one of the drawbacks to using VICSS, since the user effectively has to "program" the constraint in a series of steps. This is why it is best to characterize VICSS as a programming-by-demonstration system, rather than a machine learning system.

The next two points are tied together. If we think about a graphical structure such as a grid, it defines some "natural" concepts: items, cells, rows, and columns. There are a relatively small set of such concepts for each type of structure. This makes it easy for the user, because he only needs to remember (and think about) a few different type of concepts, and moreover, the graphical display constitutes a visual reminder of what these concepts are. As illustrated in the scenario "No one can work with McCoy" relatively complex constraints can often be specified, using the concepts that are natural to a grid, such as row and column. We believe that the generalization language *must be* simple if the user is to successfully demonstrational techniques, since the user must understand what he intends to demonstrate, if he is to verify that the system has made the correct inference. The more complex the generalization language, the more sophisticated the user must be.

Of course, not everything can be easily described using the concepts associated with grids and lists. Thus, our plans also call for the implementation of structures such as graphs and sequences. For a graph, natural concepts are nodes, links, and neighbors. For se-

[4] Our natural language generator is rather simplistic and has not yet been completed, so it may be possible to generate a more natural phrasing of the constraint.

quences, natural concepts are first, next, last, previous, and subsequent. Are these structures enough for our purposes, or will we need a whole host of different graphical displays? We believe (and hope) that the former is the case; researchers have found these particular structures very useful for describing a wide variety of combinatorial problems (Garey & Johnson 1979).

Finally, consider the fourth point. Because the generalization language is limited to a few natural concepts for each type of graphical structure, the capability to change the domain description language by creating new structures is crucial. For example, in our first scenario, we showed how the user would take the original schedule, and create the grid **Number of Tasks per Crew Member**. Once this new relation was created, specifying the constraint "No one can work more than 8 hours" was trivial. This illustrates the relationship between domain engineering and the complexity of the constraints. In general, there is a tradeoff between the amount of domain engineering the user is required to do and the complexity of the individual constraints that need to be demonstrated. Because VICSS enables the user to use algebraic manipulations *and* demonstrational techniques, the overall complexity of the specification process is greatly reduced.

The four design decisions we have discussed all contribute to making the generalization problem rather simple, or even trivial, from a machine learning point of view. Why go to such an extent to simplify the generalization problem? Certainly, we could use an first-order learning method such as Minton and Underwood's BFSF method (Minton & Underwood 1994) to learn the constraints from a series of examples. It's an intriguing possibility, but we can see two ways in which this would make life difficult for a user. First, it requires the user to provide a series of examples, until the system correctly guesses the constraint. But currently, the greater problem is that it would require that the system be able to formulate its guesses in some language so that the user could indicate whether a guess is correct or not. While English is a possibility, what is really required is first-order logic. But the whole point of the interface is to relieve the user from having to know first-order logic. (Nevertheless, requiring a user to verify that a logical statement is the intended constraint would still be improvement over requiring the user to write the constraint himself.)

Related Work

This work was largely inspired by programming-by-demonstration techniques (Cypher 1993) and, in particular, the work on demonstrational interfaces (Myers 1992). In many cases, these systems are only capable of making relatively simple generalizations, but, as we have argued, this can be an advantage when one must interact with a user. There has also been a tremendous amount of work within the field of machine learning on induction and on learning program specifications from examples (e.g, (Muggleton 1992)), but the techniques explored in this literature are generally more powerful than we require. In fact, the very power of standard induction methods can make it difficult for users to understand the generalizations! Nevertheless, the concepts and terminology of machine learning, especially those related to generalization languages and hypothesis spaces, have played an important motivational role in defining our research direction.

There has also been relevant work in the area of relational databases and spreadsheets, where the graphical displays are similar to the grids described here. Pioneering work in this area was IBM's early system Query-by-Example (Zloof 1977). However, with regard to our application area, we know of no other work where "specification-by-demonstration" methods are being used as an interface to combinatorial problem solvers. We believe, however, that this is a very exciting application area, and the potential to produce useful systems is enormous. Operations research has been the dominant approach in this area for many years, but OR methods are generally difficult to use unless one has had extensive education in this area.

Project Status

The VICSS system still under development, but we have completed the second prototype of the interface. At the time of the writing of this paper, we have recently complete user-testing, and plan to write up what we've learned from this experience and redesign some aspects of the interface.

Acknowledgments

The authors acknowledge Keith B. Olson and John Allen for there many contributions to this project. A slightly revised version of this paper appeared in a workshop on Programming-By-Demonstration at the 1995 International Machine Learning Conference.

References

Cypher, A. 1993. *Watch What I Do: Programming by Demonstration.* MIT Press.

Garey, M., and Johnson, D. 1979. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W.H. Freeman and Co.

Minton, S., and Underwood, I. 1994. Small is beautiful: A brute-force approach to learning first-order formulas. In *Proceedings AAAI.*

Minton, S. 1993a. An analytic learning system for specializing heuristics. In *IJCAI-93.*

Minton, S. 1993b. Integrating heuristics for constraint satisfaction problems: A case study. In *AAAI-93.*

Muggleton, S. 1992. *Inductive Logic Progamming.* Academic Press.

Myers, B. 1992. Demonstrational interfaces: A step beyond direct manipulation. *IEEE Computer* 25(8).

Zloof, M. 1977. Query-by-example: A data base language. *IBM Systems Journal* 16.