CONSTRAINT MANAGEMENT FOR DESIGN

William Bricken
December 1987

The body of this report contains:

-- A brief description of the AI based concepts that have been selected for prototyping, and a rationale for their selection.

-- A description of the development environment within which code development will take place, and the implications of the selection of such an environment.

-- A detailed description of the prototype Constraint Management System (CMS) being developed.

-- A detailed description of an application of the CMS.

-- Appendices that identify a complex constraint problem, and the slides for the talk covering the report.

The Constraint Management System has been developed as a prototype that demonstrates the use of advanced AI reasoning techniques. Two versions are provided. The initial development version is limited to the toy domain of cryptarithmetic. It illustrates the techniques of constraint based reasoning in a difficult but understood domain. The second version, called ConMan, illustrates the same techniques for a small portion of the design domain, that of the placement of seat and controls in a vehicle.

Objectives

The objectives were to identify and define specific AI tools that would be applicable to design automation. A constraint on these tools is that they are to be integrated into a Computer-aided design System (CADS) with relatively little effort. Since AI is basically an experimental field, this eliminated many avenues of long term research. Of the seven tools that were identified as relevant, one was selected for the prototype development.

The rest of this section provides a broad context for the use of the selected AI tool, constraint management. Following sections discuss the tool in depth.

The Design Problem

The design process, as it currently exists, is extraordinarily complex and, as one might expect, this inherent complexity presents the design team with numerous, rather formidable obstacles. This section of the report attempts to identify many of the obstacles which can, we believe, be successfully ameliorated via the introduction of artificial intelligence techniques. More detailed descriptions of these problems, along with proposed methodologies with which to tackle them, may be found in the next section.

Because of the scope of the design process, *a team* of designers will typically be assigned to the project. However, temporal, physical, and intellectual constraints often interfere with communication among design team members. This lack of communication is clearly inefficient since, without a clear communication path among individual team members, there will almost certainly be overlap in terms of work carried out. For example, *designer1* may carry out a database search ignorant of the fact that a nearly identical search had already been undertaken by *designer2*. In addition, when a number of different team members have made separate contributions throughout the design evolution of a particular component, it is difficult to maintain a clear record of the factors and individuals concerned with the making of various design decisions. A method for facilitating both the access to, and the sharing of, individual designer's data is clearly needed.

Hierarchies of Constraints

The design of any complex system is, by necessity, developed using a hierarchy of design abstractions, with design decisions specified at higher levels of abstraction influencing design specifications at the less abstract lower levels in the hierarchy. In particular, design constraints established early in the design process clearly impact subsequent design phases (*downward constraint propagation*). It is important, however that constraints be propagated both up and down the design hierarchy. For example, some high-level design constraints involve the *structural partitioning* of available resources (time, space, power etc.). As a design evolves, such a partitioning is refined and such refinements clearly impact the partitions defined for other components at higher levels in the hierarchy. It is essential, therefore, that information about such alterations in constraint partitioning be propagated *upwards* through the design hierarchy. Other situations in which constraint propagation becomes an issue involve constraints which cannot be partitioned, for example design specifications describing desired component performance. Circumstances may arise in which a constraint specified at a more abstract design stage simply cannot be met, and the designer is then forced to either proceed with the design, hoping that subsequent refinements do not violate other higher level constraints, or he must explicitly alter the design specifications at the higher levels so that they match the current design. A method is clearly needed by which the

automatic upwards propagation of failed constraint reports (along with descriptions of the reasons behind the failure) is facilitated. In general, the scope and magnitude of the design process makes the management of constraints an exceedingly difficult task. However, the problem of constraint management pervades the entire design process and any attempts to improve the design process must therefore address this issue.

In addition to the *partitionable* and *non-partitionable* breakdown described above, constraints are typically categorized as *hard* (exact specifications) or *soft* (guidelines, recommendations). Another useful breakdown is in terms of *spatial*, *temporal* and *logical* constraints, as follows:

Spatial Constraints

As the design process evolves, previous assumptions about geometry will change. It is important that the impact of such changes be quickly transmitted to the designer. For example, a designer may discover (perhaps as a results of exercising a human performance analysis model) that *display1* must be made larger. The designer should be able to quickly and easily obtain the following kinds of information

How large can I make *display1* without violating any constraints imposed by other design decisions?

If I cannot make *display1* larger without affecting constraints for other components, how can I adjust spatial allocations to the other components so as to minimize the overall impact of increasing the size of *display1*?

Temporal Constraints

Historically, as designs have grown more complex, the workload imposed upon the designer has grown dramatically. A modern day designer is faced with the problem of carrying out many tasks which are closely spaced in time. The management of temporal constraints therefore becomes a rather important issue. The designer must be able to quickly ascertain what impact violation of a temporal constraint will have upon other temporal constraints. In addition, given a candidate automation concept that will carry out *action1* automatically, the designer must be able to determine how to optimally allocate the "freed" time to other temporally sensitive tasks.

Logical Constraints

Careful constraint management is required in order to prevent the possibility of certain design activities calling for the simultaneous occurrence of two or more logically inconsistent activities. Identification of logical constraints is a clear necessity. Continuing with the display problem example, the designer should, in addition to simple identification of the

problem, be able to obtain prioritized lists of other candidate display locations, as well as lists of other displays which are likely candidates for the integration of the functionality of the problematic display. Priorities could, for example, be obtained from some function of violated soft constraints.

Managing Knowledge

As a design evolves, a tremendous amount of knowledge is acquired. This knowledge may simply be in the form of results from database searches and simulation studies, or it may be in the form of more general "lessons learned". In any event, managing and making available such knowledge will clearly impact the design process in an important way. Although such a knowledge base should certainly be made available to the designer in such a way that it can be explicitly called up and examined, it is more important that a mechanism be developed such that the knowledge base is *automatically* and *silently* accessed at the appropriate times. In addition, a mechanism should be developed which facilitates the installation of newly acquired knowledge into the knowledge base.

The creation and maintenance of an audit trail is at once crucial and difficult, particularly when many different designers are contributing to an overall design. Much of the maintenance of an audit trail should be automatic and, in addition, the audit trail itself should be maintained in such a way that it can be accessed easily and at many levels of detail and organization.

A tremendous amount of information is available to the designer in the form of data bases. It is unlikely, however, that this information will be utilized effectively unless access to the databases is facilitated. The man/machine interface should be uniform across all databases and, in addition, the designer should be able to access a database *without* having to "exit" from a given design process (i.e. access to the database should be possible without obligating the designer to enter database search "mode"). Database requests more sophisticated than the traditional Boolean keyword searches should be possible, and the designer should be able to obtain the search result in the form of a ranked list. In a similar manner, a wide variety of design tools can be made available to the designer, and it is essential that access to such tools should be provided via a uniform user interface.

CONSTRAINT MANAGEMENT SYSTEMS

Constraints pervade the entire design process and thus, in many ways, *constraint management* (the process by which the consistency of the overall design is tested whenever design changes, additions or refinements are made) is a natural tool for support of automated design. The constraint management process rests firmly upon the assumption that an appropriate representation scheme (a hierarchy of frames) is used to represent the design process. The functionality of a constraint management system, and appropriate sites for such a tool within a CADS architecture are described below.

Constraint based reasoning is a natural way of thinking about design tasks. The designer is free to imagine and construct designs which are limited only by constraints that are specified by the task. In traditional programming environments, one example, or *instance*, is developed and elaborated by the program. In constraint based programming, all possibilities are developed simultaneously. Only those designs that violate constraints are eliminated.

The constraint management tool keeps track of permissible and non-permissible designs, alerting the designer whenever the current design contains a contradiction or an impossibility. The designer is free to assign any permissible values to parameters (such as seat angle, or distance to specific controls), and the constraint system will automatically examine other parameters to determine if any of them are changed.

A Frame Based Constraint Management Mechanism

Constraint management is the key to the inference mechanism operating upon a frame-based data structure. It is quite clear that, in order to permit the designer to focus on using those skills to which he is best suited, he must be relieved of the need to attend to many of the chores to which he is comparatively ill-suited (e.g., the tedious "bookkeeping" aspects of the design process). This section describes how an automatic constraint manager might operate. This manager, in addition to performing rather mundane constraint checking and propagation chores will also serve, in some circumstances, as a *design assistant* in a *mixed-initiative* style of interaction.

Since altering a current design amounts to editing a frame, the potential for a constraint violation arises whenever a slot is filled. It is often the case, however, that a slot is filled not with a value, but with a list of descendant frames, each of which can have a number of slots filled automatically via inheritance or the acceptance of default values. In this way, a seemingly simple change to the design can spawn a plethora of potential constraint violations. This combinatorial explosion is best handled if a means can be found to limit the number of possible constraints that must be checked. Limiting search is a common theme throughout AI

research and, in this instance, one approach to the problem is to limit the search to only certain types of constraints.

Types of Constraints

From the broadest perspective, we can view constraints as being categorized by the types of data they operate on. For example a constraint dealing with limits upon the dimensions of a particular component is "attached to" those slots which describe the height and width of the component (or, more likely, the generic component type). When a change is made to one of these slots, we certainly don't want to search for violations of all possible design constraints and so search is limited via the implementation of an *if-changed* demon The demon is activated upon alteration of the slot value and directs the constraint manager to check only those constraints attached to the changed slot.

Another way of categorizing constraints so as to limit search is to consider the concept of *timeliness*. A constraint is considered to be timely only when all of the slots to which it refers have known values. For example, consider the following constraint:

> Is the range of seat articulations such that both *display1* and *display2* are clearly visible by 10th to 90th percentile drivers?

Now, if the specifications of *display2* had not yet been established, the constraint would be considered untimely, and one of the ways to limit search is to process only those constraints which are timely. Suppose, for example, that one of the slots to which the above constraint is attached (e.g. the width of *display1*) is altered. The alteration will activate the *if-changed* demon, and processing of the constraint will begin. However, because *display2* is not yet fully defined, the constraint manager must explore the implications of all possible dimensions of all possible designs for *display2* which are presently being considered. For a complex constraint which references a number of incomplete designs, this can quickly lead to an explosion of possible search paths. This phenomenon is, however, prevented if search is always limited only to timely constraints. Since, however, the design for *display2* will, with certainty, eventually be completed, the designer may be confident that the constraint will, at that time, be considered timely and will therefore be checked.

Constraints can also be categorized as falling into a number of different *contexts*. Consider, for example, the design context in which a variety of different proportions for *display1* are being explored. Since there is a fixed amount of space available for all vehicle panel components, each change made to the proportion of the display will have an impact upon the space available for the rest of the components. Since, however, the designer is in an exploratory phase, he will have little interest in the impact of each

change on overall seat geometry (unless, of course, constraints specifying the minimum and maximum dimensions for the display are violated) until the proportions of the display have been more fully refined. Associating a design context to a slot's *if-changed* demon will enable the designer to suppress constraint checking until such checking is appropriate.

One could also envision contexts being defined according to the identity of the designer, in the sense that designers would "sign in" and all subsequent constraint checking would take place within the context of the type of design task to which that designer had been assigned.

Another way of tackling the above problem is to categorize constraints by the time at which they should be checked. Some constraints should, for example, be checked every time the appropriate slot is altered (subject, of course, to the timeliness mechanism), whereas other constraints should only be checked at "design submission" time. Returning to the display example, we can imagine that the constraints associated with the minimum and maximum allowed dimensions for the display should be checked whenever the *if-changed* demons for the dimension slots are activated. In contrast, however, the partitionable constraints associated with overall seat geometry would only be checked following the final definition of the dimensions for the display (i.e. at design submission time). A mechanism such as this can be implemented simply by including a "time for checking" entry on each constraint's property list.

Constraints may also be categorized according to their degree of *hardness*. We can imagine that, in order to limit search, only those constraints exceeding a pre-specified degree of hardness would be checked. Specifying a hardness level for a constraint also serves other useful purposes.

When constraints are violated, it makes good sense for the designer to address only with those constraints which cause the biggest problems (i.e. the hardest constraints). For example, it seems fruitless to inform the designer that CRT model K5-A has a green monochrome display (whereas a soft constraint suggests that a yellow monochrome display is desired) when the CRT model itself is too large to fit into the allocated space (violation of a more important constraint).

When a number of possible choices are available for a given slot, each of which succeeds in meeting all of the constraints up to and including the specified hardness level, the choice which satisfies constraints down to the softness level is the natural choice. In this way ranking constraints by hardness serves as a design aid.

Constraint Explanation

The designer can be supplied with an explanation of the mechanism behind a constraint violation, via a simple tracing backwards of the steps which took place during the process of establishing that the violation occurred. For example, consider the following constraint:

```
IF Output Device is crt, AND
      Display location is within (128,342) to (150,420), AND
      left-turn is in process,
THEN constraint violation (hard=1)
      between location, type & turn
```

Given that such a violation occurred, the designer could request an explanation, and thus discover that during left turns, the seat articulation is such that any visual displays located within the specified coordinates are not clearly visible. The designer might then try a different location, consider a different seat type, or even consider a voice output device.

Constraint Inversion

As described previously, when one or more of the slots to which a constraint is attached are empty, the constraint is considered to be untimely. When only one slot is unfilled, it becomes possible to assist the designer in choosing values for the slot by means of a mechanism known as constraint *inversion*. Consider, once again, the display constraint specified above. Given that *display2* is not yet defined, the designer will no doubt be interested in examining candidates for this display. By requesting a constraint inversion (either explicitly or in a more relaxed mixed-initiative framework), the designer can obtain a range of possible locations for the display such that the visibility aspect of the constraint is satisfied for the specified seat dimension. In this manner, selection of candidate displays is facilitated, since only those types which are of the appropriate dimensions need be considered. The mechanism of constraint inversion, then, acts as a designer's "assistant" in this framework. Indeed, one can imagine situations in which the inversion could result in the generation of a single unique value for an unfilled slot, therefore "automating" the design process.

Constraint Based Reasoning

Constraint reasoning is inference over possibilities rather than instances. Constraint reasoning systems have been demonstrated in the domains of simulation and circuit design (A. Borning, *ThingLab - an object-oriented system for building simulations using constraints*; Sussman and Steele, *CONSTRAINTS - A Language for Expressing Almost-Hierarchical Descriptions*).

A Constraint Management System (CMS) can keep track of the total of possibilities remaining whenever a designer enforces a constraint on the design object. Rather than reasoning about exact instances, the CMS reasons about *remaining possibilities* whenever decisions are enacted. Rather than providing points in the space of acceptable instances, the CMS provides the entire space, as it is carved up by imposed constraints.

As a simple example, imagine a switch with four possible positions, labeled 1, 2, 3, and 4. We are told that the current position is not odd. We ask what the current position is. A standard reasoning system, such as Prolog, would select a position, say 1, and test whether it passed the rule that the current position is not odd. Since 1 is odd, the selection would fail, and the system would make another choice. Say it picked 2. Position 2 is not odd, and therefore acceptable as a possible current position. But then, the Prolog system would stop. We might ask it to find *all* solutions, in which case it would select and reject 3, then select and accept 4, then notice that there are no other choices, returning the possibilities {2 4}.

A constraint reasoning system works in the opposite direction. Its world would start with all possibilities, {1 2 3 4}, a full but unconstrained description of the switch. When told that the current position is not odd, the system would impose that constraint directly upon the universe of possibilities. The choices 1 and 3 would be immediately eliminated, leaving {2 4} as the description of the switch. When asked about the current position, the CMS already knows the answer, it returns its description of the switch, {2 4}.

A constraint maintenance system attempts to maintain the truth of a set of constraints while minimizing the range of values that the unknown variables in the problem can assume. Returned results fall into three categories:

Over-constrained: Problems with too few unknowns, and too many constraints are unsolvable. All sets of solutions are contradictory. A CMS should be able to identify impossibilities.

Unique: One exact solution exists that satisfies the constraints and determines a value for all variables. A CMS should find this solution.

Under-constrained: Problems with too many variables and too few constraints permit many different solutions. Any one of the solutions is possible. A CMS should return all solutions, or a procedure for generating all solutions.

The Choice of Constraint Management

Several criteria for selecting a prototyping concept are relevant. The prototype should be visibly demonstrable rather than just a proof of concept. This implies that the prototype should embody a clear functionality, as opposed to being a display with limited capabilities.

Using these criteria, a Constraint Management System (CMS) was chosen for prototype development. This system can be prototyped with substantial functionality, without relying on previous automation of design processes. The knowledge engineering necessary to demonstrate constraint management is minimal, since many design specifications are already expressed in terms of constraints. We have decided to focus the prototyping effort on tool functionality rather than on extensive knowledge engineering of the design domain under the assumption that specific aspects of the domain can be configured relatively easily to be used on a fully functional tool. The prototype tool described within this chapter demonstrates a capability to solve complex and difficult constraint reasoning problems within a domain that has been carefully knowledge engineered.

The Constraint Management System offers functionality in a critical aspect of the design process, that of *bookkeeping*. Design requires the balancing and negotiation of hundreds of constraints, many of which interact. The CMS helps to keep track of design decisions and partial decisions. It can alert the designer whenever a particular decision contradicts other decisions or pre-established specifications. It can enforce the effect of decisions on other aspects of the design, and limit the freedom of the other aspects accordingly. Constraints apply early in the design process; it is efficient and cost effective to alert the designer to any violations of constraints. This early warning can save over commitment to designs that might later be found unacceptable. Designs which are mutually incompatible can be identified during parallel development, and thus save redesign effort when the competing designs are being reconciled.

Extensions to Constraint Reasoning

At a minimum, a reasonable constraint management system should serve to alert the designer when concrete limits are violated. It should be flexible enough to permit *violation alerting* to be customized based upon the "hardness" of the constraint, the level/seniority of the designer, and the level of the design (prototyping/exploring vs. fine-tuning).

Efficiency is critical to constraint reasoning, since the world of possibilities is usually a large domain. To be efficient the constraint management system should be designed so that appropriate pruning of search paths be enacted by existing constraints, reducing the amount of search

necessary to specify solutions. Pruning can take place via *embodiment* of constraints in a matrix of possibilities.

An attractive enhancement to a constraint management system is to allow for a mixed-initiative style *explanation facility*. Such a mechanism would be capable of providing the designer with a description of the process by which the constraint violation was detected. Such a facility would enable the designer to more easily re-specify the proposed design or even to override the violation altogether.

Another useful enhancement involves assisting the designer via a simple mixed-initiative style dialogue. A system could be designed via the process of constraint *inversion* in which ranges of acceptable design parameters (the range determined by the constraints imposed by other design features) are presented to the designer.

To summarize, there a variety of ways in which constraints can be processed. Utilizing demon invocation and constraint categorization facilitates the reduction of search, and constraint inversion and tracing support suggestion and explanation mechanisms which can be presented to the designer in a mixed-initiative framework.

DESIGN OF THE CONSTRAINT MANAGEMENT SYSTEM

The goal of the CMS project is to develop a prototype constraint management system that integrates logical and numerical constraints in a deductive engine.

The expected functionality of this system follows:

1. The CMS should accept as input *rules* and *equations* that contain non-exact logical and numerical descriptions.

-- Logical data structures should include disjunctive choices, such as (A or B or C).

-- Numerical data structures should include ranges, such as 1<x<5.

2. The system should accept user specified constraints and variable bindings interactively.

The system should perform deduction over non-exact data, returning:

-- contradiction warnings when no data can fit the specifications, or when user specified data violates existing constraints,

-- limitation warnings when only one datum fits, and

-- possible worlds when ranges of data fit the specifications.

3. The system would tightly interact with intelligent data structures, which maintain bookkeeping and notification facilities to reduce deductive overhead.

The proposed development methodology follows:

An algebraic boundary logic formalism will be used for deduction. Boundary math is fully described in W. Bricken, *The Efficiency of Boundary Mathematics for Deduction*).

The boundary logic will be integrated with equations (equalities and inequalities) which specify constraints.

We will limit the expressability of the system during development, progressing through propositions, equality, sets, linear equalities with free variables, and inequalities. Examples follow:

Propositions:
A and (if A then B) imply B.

Equality:
A = B and B = C implies A = C.

Sets:
A = {1, 2, 3} and A = B implies B = {1, 2, 3}.

Linear Equalities:
a + b = 6 and a = b implies a = 3.

Inequalities:
a + b < 9 and b > 5 implies a < 4.

We will test the system on a well-formed toy problem, cryptarithmetic. We will then engineer a small subset of Passenger Accommodations Specifications, for demonstration purposes.

The Prototyping Domain

Construction of sophisticated software requires preliminary designs and tests that assure that the software is functional for known problems. In a rapid prototyping environment, a well-understood test domain is critical for principled implementation and debugging.

The field of Artificial Intelligence has a body of well-understood problems for testing and refining software models. These domains are usually called *toy* domains, because they model structured worlds. This nomenclature should not be confused with *trivial* domains, since most toy domains still pose unsolved problems for AI techniques.

We will be testing the Constraint Management System with the toy domain of cryptarithmetic. Cryptarithmetic is basically addition problems, in which each number is unknown. A simple example is

AA + BB = CBC

Each letter stands in place of a specific number. The task is to determine which number is associated with each letter, without any further information. To do this requires complex reasoning capabilities, knowledge of when guessing is needed, and extensive practical domain knowledge about arithmetic. The answer to the problem is

a = 9, b = 2, and c =1.

The domain of cryptarithmetic is used both to pose problems in constraint-based reasoning, and to illustrate our approach to their solution. It has been extensively studied (Newell and Simon, *Human Problem Solving*; H. Simon, *The Sciences of the Artificial*).

The key to make difficult problems tractable is to find a powerful representation. Without intelligent domain engineering, computational effort quickly becomes too expensive to be useful. The cryptarithmetic domain illustrates simple yet powerful representation techniques coupled with simple yet powerful deductive techniques. The examples we present outline the design and mathematical philosophy of a computational approach that permits constraint reasoning over sets of possibilities.

The specific cryptarithmetic problem presented is a complex example of planning and search. It is used in AI courses, and is excellent for highlighting the computational burdens placed upon constraint reasoning systems. The proposed solution strategies generalize to more practical domains, such as automobile design.

Generalizing the CMS to Design

Cryptarithmetic is not in itself a critical skill. But this toy domain models very closely the task of constraint maintenance in design. The unknown letters are like unknown design parameters. They must lie within a specific range, but the only constraint that makes them unique is their inter-relation with other aspects of the problem itself. The constraints that make design parameters unique are specified by the inter-relations of all the design components. A software constraint management system must at least be able to address the toy domain before being extended to the real domain of design.

One generalization that would be necessary for design is to extend the data types to continuous variables. The sets of possible values for a variable would then be expressed as lower and upper bounds. For example,

X = {2.0 8.0} means 2.0 < X < 8.0

When two possibility variables are equal, they each may take on values from the intersection of their ranges.

The physical parameters of the design can then be expressed as possibility variables. In addition, logical relations can be posted as constraints. Possibility variables are fully compatible with logical specification. For instance,

Position-of(x) = {left middle right},

where x is a type of knob.

Lessons learned can also be expressed within a constraint reasoning system. If it is mandatory that a specific lever have three inches clearance from the seat, that fact can be asserted as a hard constraint. Hard and soft constraints are implemented within the selection criteria of the control mechanism. The more important the constraint, the earlier in the design process it is embodied in the data structure.

Constraint reasoning can freely mix constraints from different domains within the same model. Relations about cost, effort and demand can be encoded and can interact with physical and problem-oriented constraints. The trick here, of course, is that the various relationships must be known to the extent that they can be explicitly encoded during knowledge engineering.

Another characteristic of the CMS is that design decisions that are imposed on the model externally can be easily incorporated. Say, for instance, that a designer wanted to see what possibilities remain if a problem was solved using only half of the available resources. By interactively inserting the

half-resource constraint, the system would identify solutions that met that and other existing constraints.

The Application Domain

After developing and testing the techniques of constraint management on cryptarithmetic problems, we next applied these techniques to the domain of design. The resulting system, which is an extension of the CMS, is called ConMan, for CONstraint MANager.

ConMan takes algebraic constraints (equations and inequalities) as input and assures that particular values of the variables in the constraint equations are not contradictory. For design, this means that input equations specify the relationships between measurements of location for the various components of the vehicle seat. Input inequalities specify the limits on the placement of seating components.

The ConMan system provides a diversity of pop-up windows which display the state of the computation and permit changes to the existing databases of equations and constraints.

To utilize the constraint management system, the domain engineer must specify the equations that define relationships between variables in the problem. Usually, these variables will identify some measurement in a design. But as a simple example, consider the design of a triangle. Variables (A, B, C) can identify each of the three angles of the triangle. The equation that defines the relationship between angles in a triangle is

$$A + B + C = 180$$

Next, the designer must specify the desired constraints on the design. These are expressed as inequalities that limit the possible values of the variables. In the triangle example, it may be desirable that angle A be between 40 and 50 degrees. This would be expressed as the constraints

$$A > 40 \text{ and } A < 50$$

Or perhaps, the angle A should be exactly 45 degrees. This is expressed as the constraint

$$A = 45$$

More complex constraints may also be expressed. For instance, if it is desirable that the triangle be isosceles, the appropriate constraint would be (A = B). Whatever value either angle A or B was confined to, the system would assure that both are equal in the final design. Finally, if the designer specifies

A = B and B = C,

then the system would determine that each angle must be 60 degrees.

From an operational perspective, the specified equation database defines the abstract design. This is the parameterized drawing that contains, for example, the seat, instrument panel, windscreen, and other objects in the driver's compartment. The relations between design objects are specified by algebraic letters, which are variables that may change, but generally will not be totally ignored.

The constraint database represents limitations on the parameters in the equation database. The designer may change these during the course of design. One way of contrasting equations and constraints is that equations are *hard* limitations. They define the possibilities. Constraints, on the other hand, are *soft* limitations, they define a range of desirable possibilities but may change. Within the limitations of both equations and constraints, there exists the possibilities of the design. One particular possibility is an *instance*, or instantiation of the design. If there are no possible designs, no instances that satisfy both equations and constraints, then ConMan notifies the user of a contradiction. The user then must change existing constraints to re-establish a possible design. If constraints are inflexible, then the abstract design itself may need to be altered.

In general, then, the equation database *generates* the space of possibilities. The constraint database *evaluates and tests* subgroups of possibilities. The user *modifies* the design in case of failure.

To support visualization of the constraint process, ConMan provides three interactive displays.

The Dependency Graph Browser displays the dependencies between each variable in the constraint database.

The Values Display shows the particular setting of each variable in the current instance of the design, as well as the range of values that each variable may assume.

The Figure provides a two-dimensional scannable diagram of the design.

General Strategy

The facts embodied in the domain, (such as the meaning of a letter in a particular place in a cryptarithmetic puzzle) need to be encoded in rules. A data structure that keeps track of all possibilities is needed to support both bookkeeping and deduction. The software architecture must handle the domain, constraints on the domain, and instances, or problems, within the domain.

The meaning of the input problem is embodied in *transcription* rules. The transcriber converts the peculiarities of the input problems to a more standard domain of algebraic equations. The transcriber is called once, when a problem is entered into the CMS. It generates the *problem specific data structures*.

The rules of column addition form the abstract ground for forming constraints in cryptarithmetic problems. They can be formulated as meta-rules, independent of any particular problem. These meta-rules are applied to the input problem, generating constraint equations which populate the *problem specific constraint database*.

The control mechanism should be general, so that different transcribers can use it for different problem domains. (A limitation is that domain engineering must permit the expression of a domain in the language of the control mechanism.) The proposed control mechanism is simple: it applies constraints from the constraint database to the data structure of possibilities. Limited possibilities in this data structure can in turn refine and augment the constraint database. When a constraint is applied to (embodied in) the data structure it is removed from the constraint database. When no constraints remain, the data structure indicates all possible solutions. The key to a good control structure is to have sufficiently refined *selection rules* which choose the next constraint to apply to the data structure.

Both the data structure and the constraint database are *intelligent*. This simply means that they have rules associated with them that are triggered whenever either is accessed or changed. Part of domain engineering is to identify useful behaviors for these separate intelligent databases. Adding rules that trigger when a database is accessed (called *demons*), makes databases more than static storage locations. This partitioning of bookkeeping functionality is critical for efficient deduction.

The Computational Technique

The power of the constraint maintenance system is demonstrated by its performance on difficult examples from cryptarithmetic. The CMS proceeds directly to the logical solution, requiring a minimum of search. The conditional branches it explores are well selected and quickly terminate. The secret of success is that the CMS uses *equations* to maintain information, and an intelligent database to remove the burden of computation from the deductive engine.

The constraint management principles include the following:

Extensive Domain Engineering

Inference in any domain requires that the facts of the domain are adequately encoded. Constraint based inference is particularly sensitive to domain engineering, since inadequate restriction and definition of a domain will cause the existing constraints to be ineffective. This creates unnecessary search and under specification of the data structure.

Generation of Constraint Equations from Domain Facts

The CMS requires a particular form of data, that of constraint equations. Usually, the representation of domain facts is not in this form. If knowledge engineering identifies abstract structures within the domain, then specific problems can be converted from the native representation to the constraint representation by passing them through a transformation program that applies abstract relations to the specific problem to generate specific constraints. This mechanism permits a single transformation procedure to handle all problems of the class for which it is an abstraction.

Constraint Equations as Data

Once a problem is specified as a collection of constraints, this collection can be viewed as a description of the problem. The constraint equation database is a knowledge representation technique that is particularly well suited for dynamic pruning of the space of possibilities. Rather than specifying specific collections of parameter values as instances, this database specifies relations between parameters that hold under all circumstances. When the relations are suitably constraining, single instances (or impossibilities) can be asserted.

Intelligent Data Structures

Many aspects of domain knowledge should be expressed as part of the data structure itself. It is an historical artifact that data structures are viewed as passive. By attaching programs that are triggered whenever a fundamental aspect of the data is contradicted (such as an attempted assertion that the number 6 is not equal to the sum of 2 and 4), the data structure can behave intelligently about its own form. Traditionally, contradictions are asserted into a database, and the inference procedure is then responsible for eliminating the contradiction. This is inefficient, since the computational procedure is responsible for many other things, and it is misdirected, since the responsibility for some configurations of data is not localized. Active objects, as data structures that can process incoming and outgoing information and changes, permit a wider range of models and transfer many awkward global processing tasks to local maintenance tasks.

Control that Selects and Applies Constraints Efficiently

Some constraints are more revealing than others. In particular, an assertion of fact, such as (x = 3), is a powerful constraint, since it limits the value of *x* to a single choice. The objective of a control mechanism for constraint reasoning is to transfer information between the data structure and the constraint description, and in the process, to minimize the space of possibilities embodied in the data structure. This transfer can be done randomly, but the real power of the control mechanism is in selecting efficient constraints to embody first. Which constraints are efficient is dependent both on characteristics of the constraint language, and on domain characteristics. In general, constraints with fewer variables are more specific, as are equations involving equality rather than inequality. In the case of multi-variable constraints, dynamic domain information can be included. For example, a constraint with variables that have only two possibilities is more powerful than constraints with variables that have many possibilities. The control mechanism can easily incorporate a selection mechanism that evaluates the constraining power of existing equations, and selects the best for embodiment.

Possibility Calculus

Each variable within a domain can be seen to represent the selection of possibilities of any object in that domain. If the variable *x*, for instance, is known to represent an object from the domain of single-digit integers, then *x* represents the possibility set

{0 1 2 3 4 5 6 7 8 9}

A possibility calculus permits computation over variables that represent sets of possibilities. When a variable can assume only a unique value, the possibility set data type is the *oneof* function. One structure of interpreting equations as a possibility calculus is as follows:

x = constant

There is only one possibility for the variable. The possibility is the constant.

x = y

The possibilities of both variables are the set intersection of the possibilities of each variable.

x = y + constant

The possibilities of *x* are the set intersection of *x*'s current possibilities with the possibilities generated by adding the constant to each of *y*'s possibilities. Conversely, the possibilities of *y* are the set intersection of the possibilities created by subtracting the constant from the each of the possibilities of *x*.

x = y * constant

Same as above (addition), but the possibilities of *y* are multiplied by the constant.

x = y + z

The possibilities of *x* are equal to the possibilities generated when each possibility of *y* is added to each possibility of *z*.

The possibility calculus is not a probability calculus, since nothing is averaged. Nor is it a calculus of sets, although operations may be implemented as set operations. Although the examples in this report are from discrete mathematics, the approach generalizes well to continuous domains.

Conditional Constraints as Search

Constraint equations can be mixtures of logic and algebra, as in

```
if x = 1 then y = z
```

One way to generate search trees is to do case analysis by asserting conditional constraints for each case of a particular variable. The

selection heuristics of the control mechanism can then choose between absolute or conditional constraints for embodiment. If absolute constraints are not sufficiently limiting, then a particular hypothesis that does engender tight constraints can be explored. Since the cases are each posted to the constraint database, they can be interleaved or begun then abandoned. The cost of conditional constraints is the dynamic generation of data structure clones, to represent the hypothetical world generated by the constraint.

If-changes Constraints

Constraints with multiple variables may embody some restrictions on the data structure, but fail to achieve a unique value for each variable. Even though these constraints have been embodied, they still contain information which becomes relevant whenever a variable which they contain changes. These constraints are posted as if-changes constraints, and are triggered only when the specific variables change. To remove the effects of if-changes constraints from the constraint database, a variable is selected to be replaced by its algebraic equivalent. This approach achieves simultaneous solution of linear equations, and permits deferment of application of constraints to when they become relevant.

Plans for Integration

The constraint management system is essentially an inference system applied to the frame-based representation of the design process. As such, the most appropriate location for the constraint management system is at the level at which the design itself is managed. If a common design, administered by the CADS host, is implemented then the constraint management system should naturally reside on the CADS host. If however, individual workstations support their own copies of the design, then the workstations should also administer the constraint management tool.

A constraint database will be developed from design specifications and from interviews with domain experts. This database will be stored in the same system as the CMS tool.

In the long term, the CMS might be integrated with a CADS through an attached AI workstation. The CMS would reside on that workstation, while the constraints database would reside within the selected commercial database management system, and be called from the AI workstation. Designs constructed within a CAD tool would need to have their output converted into a format that is compatible with the input expected by the CMS.

Another possible architecture would have the CMS callable as a tool from the designer's notebook. This approach is preferable if the calling protocol

supports calls to the CMS from the design native language. Language incompatibilities can be alleviated by converting the CMS to the language C, but such an effort would require experience with the combined system before being justified.

If the CMS is treated as a callable tool, each analytic tool would need to have its output expressed or expressible in a form relevant to the constraint model. Analytic models would need to be knowledge engineered into the constraint format, critical points in the design path would need to be identified as needing constraint checking, and tools for the reconciliation of contradictory constraints would need to be developed.

A fully useful constraint management system requires several support tools. Specifically, a constraint editing system would be needed for the addition, removal, and change of relevant constraints. And an integrated data retrieval system would be needed to facilitate communication between design data and constraint rules.

THE CRYPTARITHMETIC PROTOTYPING DOMAIN

Cryptarithmetic consists of arithmetic problems in which numbers are replaced by letters. The task is determine what number each letter represents. This task is very well-suited for constraint based reasoning, because the structure of the arithmetic problem offers simple constraints, while the unknown value of each letter variable offers an opportunity for difficult problem solving.

Consider the following problem:

SEND + MORE = MONEY

Each letter stands in the place of a number. The objective is to identify a unique number from 0 to 9 that can be substituted for each letter, while maintaining the truth of the arithmetic equality. The *constraint* that this problem expresses is the equality between two numbers and their sum. The only available facts are those offered by the definition of *addition*. For example,

D + E = Y

Humans find this problem extremely difficult. There are a few initial clues, such as the observation that (M = 1). This fact can be deduced from seeing that no numbers contribute to the ten-thousands column except the carryover from the addition of S and M in the thousands column. The fact that (M = 1) permits further information to be deduced. Specifically,

S + 1 = O

But now there are too few known facts to go further. The unknown S and the unknown O in the above equation are mutually constrained, in that O is the *successor* of S. But both are still unknown. We know that S could be one of the set

{0, 2, 3, 4, 5, 6, 7, 8, 9}

We can make further deductions, such as noticing that the number 9 does not have a successor that is an integer, so S cannot have the value 9. Similarly, the successor of 0 is 1. O cannot be 1, because M already has that value. However, these deductions yield very limited information and require substantial intellectual patience. Very quickly the number of possible alternatives overwhelms what we can keep in our mind. Even principled bookkeeping fails to be enough, since the problem also requires some guessing (search) before yielding its unique solution.

Other cryptarithmetic problems include:

DONALD + GERALD = ROBERT

UNION + SOUTH = AFRICA

UNITED + STATES = AMERICA

CROSS + ROADS = DANGER

LETS + WAVE = LATER

Prototyping

The structure of the cryptarithmetic domain permits a clear exposition of the conceptual and implementation details of constraint maintenance. The central issues of *detailed domain engineering*, *constraint databases*, and *intelligent data structures for possibilities* are easily understood and illustrated using this domain as a prototype.

In particular, the performance of a constraint based reasoning system is clearly illustrated. The example in this section has been chosen to be difficult, and to demonstrate all facets of constraint maintenance. Use of initialization constraints, equality and inequality constraints, conditional constraints and if-changes constraints are each presented in detail.

Coordinating each of these techniques is the control structure for constraint reasoning, which selects constraints for embodiment in the data structure, and manages transactions between unrestricted, conditional, and if-changes constraint classes.

DOMAIN ENGINEERING

Data Structure

Each letter in the puzzle represents a different number uniquely. This can be specified by a set of non-equalities:

$$S \neq E \neq N \neq D \neq M \neq O \neq R \neq Y$$

To place these non-equalities in the constraint database presents a severe computational burden to the deductive system, since it must compare tokens that are different to see if their values are different. It is better to *assume* that different letters represent different numbers, because they are different tokens. Uniqueness and non-equality of the values of letters is then transferred to the data structure that maintains the actual assignment of variables to constants. Whenever a letter is assigned to a number, that number is removed from the possibility set of each other number *by the data*

structure itself. If two letters are asserted to be equal, a contradiction is immediately identified.

To achieve a knowledge-based data structure, each letter is represented by a *universe of possibilities variable*. If we represent a generic letter by X, then

X = {0 1 2 3 4 5 6 7 8 9}

This collection means that X can take on any one of the values between curly braces. It does not mean that X is equal to the set of integers. The concept that a letter can be only one unique number means that the structure

{0 1 2 3 4 5 6 7 8 9}

represents the *oneof* operation. The possibilities are joined by *exclusive-or* extended to be *variadic* (operating on any number of arguments).

The initial data structure is then a matrix of possibilities, with each row labeled with a letter. To remove the remaining semantics of words, the letters are standardized from *a* to *h*. The original letters of the problem are written on the left for ease of translation.

Initial Data Structure:

S	a	0 1 2 3 4 5 6 7 8 9
E	b	0 1 2 3 4 5 6 7 8 9
N	c	0 1 2 3 4 5 6 7 8 9
D	d	0 1 2 3 4 5 6 7 8 9
M	e	0 1 2 3 4 5 6 7 8 9
O	f	0 1 2 3 4 5 6 7 8 9
R	g	0 1 2 3 4 5 6 7 8 9
Y	h	0 1 2 3 4 5 6 7 8 9

The data structure is defined so that each letter label for a row may take on as a value any available number in that row. When a letter is assigned to a numerical value, the column of that number is no longer within the set of possibilities of other letters. If a letter finds itself with only one possible value, that value is assigned automatically.

Implementation of the Data Structure

The data structure represents a possibility matrix. Cells are binary, on (=1) if the value is a possibility for the letter, off (=0) if the value is not a choice. Rows and columns are labeled. The initial matrix follows:

```
   0 1 2 3 4 5 6 7 8 9

a  1 1 1 1 1 1 1 1 1 1
b  1 1 1 1 1 1 1 1 1 1
c  1 1 1 1 1 1 1 1 1 1
d  1 1 1 1 1 1 1 1 1 1
e  1 1 1 1 1 1 1 1 1 1
f  1 1 1 1 1 1 1 1 1 1
g  1 1 1 1 1 1 1 1 1 1
h  1 1 1 1 1 1 1 1 1 1
```

The accessors to the data structure are *get* and *set*.

(get <label>)

returns the set of possible values for that label.

(set <equation>)

changes the data structure. These changes trigger demons which potentially can send *set* messages to the constraint data structure to change it.

In particular, if a variable is assigned a specific value by a constraint equation (such as $a = 3$), the control structure sends (set $a = 3$). The receiver for the *set* command compares the equational constraint to the data structure. If the value 3 is not a possibility for *a* in the data structure, the receiver demon returns failure, since the data structure embodies the assertion that ($a \neq 3$). If 3 is a permissible value, the receiver triggers a local *change* demon which removes alternative values for the variable *a* and removes the value 3 from the possibilities of other variables. If in the course of removing possibilities, other constraints are encountered (for instance, *b* may be one of 3 or 4; then removing 3 fixes *b* to be 4), they are returned as assertions to the constraint database.

Decimal Place Notation

The position of each letter in a word-number contains information about the number power of ten to which the value of the letter is raised. The furthest right letter in a word-number is the units column. Each column to the left is then ten times greater. Thus we know the following meta-information about all word-numbers (to avoid excessive abstraction, this information is expressed herein for four digit numbers only), the *unit-transcription rule*:

wxyz ==> 1000w + 100x + 10y + z

This meta-rule is applied at transcription time to each word-number in the problem. It converts word-number representations into unit representations for numerical processing.

The summation problem itself contains another important piece of domain information, the *summation constraint*. This constraint is formed at transcription time by the substitution rule:

```
            uv + wx = yz  ==>

                  (unit uv) + (unit wx) = (unit yz)
```

The final domain fact about the representation of a word-number is that the first letter cannot be zero, the *Don't-start-with-0* rule:

```
            xyz  ==>  x ≠ 0
```

Applying these rules to the current example generates the constraint database for the example:

```
             abcd  ==>           1000a + 100b + 10c + d
             efgb  ==>           1000e + 100f + 10g + b
            efcbh  ==>  10000e + 1000f + 100c + 10b + h


                 abcd + efgb = efcbh
                           a ≠ 0
                           e ≠ 0
      9000e + 900f + 90c + h = 1000a + 91b + d + 10g
```

Rule of Addition

There are three structural meta-rules that apply to column addition:

```
            Ci-1 + Xi + Yi  =  Ri + 10Ci

            Ci = 0   <==>   Xi + Yi < 10
            Ci = 1   <==>   Xi + Yi >  9
```

For the i-th column of addition, the sum, (Xi + Yi), plus the carried over Carry, Ci, is equal to the Result. This result is decomposed into an integer Ri and the possible carry to the next column. The carry is 1 if the sum is greater than 9 and 0 if the sum is less than 10.

These addition meta-rules introduce the notion of a carry variable, one for each column. This carry variable may be equal to zero or one. The data structure is:

```
C0   0 1
C1   0 1
C2   0 1
C3   0 1
C4   0 1
C5   0 1
```

Here C0 represents the carry into the units column. It is always 0, a domain fact of the units column. Likewise C5 is 0 because there are no numbers being added in the last (here, ten-thousands) column.

The first of the addition meta-rules is applied to each column in the problem at transcription time. The last two rules are substantially different, since they are control rules that depend on known information. They are implemented as demons in the data structure. If possibilities change, these rules can be triggered. When triggered, they placed constraints in the constraint database.

When the column addition rules are applied to the example problem, the following constraints are generated:

```
          C0 = 0
          C5 = 0
C0 + d + b = h + 10C1
C1 + c + g = b + 10C2
C2 + b + f = c + 10C3
C3 + a + e = f + 10C4
C4 + 0 + 0 = e + 10C5
```

Note the necessity in the last equation to represent not existent numbers as zero. Also note that the original equation is depicted in the bottom five equations when the page is rotated counterclockwise by 90 degrees.

The entire equational fact base is presented again in the Figure on the following page. The care taken to specify and engineer this relatively simple domain is indicative of the effort needed to implement constraint management for real world domains, design in particular. The utility and power of deductive constraint management requires careful specification and partitioning of the domain into constraint generators and transcribers, intelligent databases, and sophisticated control structures. All these elements must be integrated into a coordinated system. The sobering fact learned by Artificial Intelligence research over the last decade is that intelligent tools require exacting knowledge engineering.

THE INFORMATION BASE FOR THE CRYPTARITHMETIC PROBLEM

Information Base for SEND + MORE = MONEY

Data structures:

```
S   a  0 1 2 3 4 5 6 7 8 9
E   b  0 1 2 3 4 5 6 7 8 9
N   c  0 1 2 3 4 5 6 7 8 9
D   d  0 1 2 3 4 5 6 7 8 9
M   e  0 1 2 3 4 5 6 7 8 9
O   f  0 1 2 3 4 5 6 7 8 9
R   g  0 1 2 3 4 5 6 7 8 9
Y   h  0 1 2 3 4 5 6 7 8 9

C0   0 1
C1   0 1
C2   0 1
C3   0 1
C4   0 1
C5   0 1

a ≠ b ≠ c ≠ d ≠ e ≠ f ≠ g ≠ h
Ci = 0   <==>   Xi + Yi < 10
Ci = 1   <==>   Xi + Yi >  9
```

Constraint database:

```
          C0 = 0
          C5 = 0
           a ≠ 0
           e ≠ 0
C0 + d + b = h + 10C1
C1 + c + g = b + 10C2
C2 + b + f = c + 10C3
C3 + a + e = f + 10C4
C4         = e + 10C5
9000e + 900f + 90c + h = 1000a + 91b + d + 10g
```

COMPUTATIONAL DETAILS FOR THE EXAMPLE

The deductive strategy of the CMS is to disambiguate algebraic variables by applying constraints to the data structure of possibilities. Since constraints are embodied in equations, solution of equations has the effect of tightening constraints around the remaining variables. To remove a constraint equation from the database, its information is embodied in the data structure. Constraints are selected by the control structure and sent to the data structure. Demons in the data structure make modifications, and may return additional constraints to the constraint database. The control structure selects constraints until the stack is empty. The state of the data structure then represents all possible solutions.

Selection of constraints by the control structure is a difficult task. Usually it is better to leave underconstraining equations as descriptive constraints rather than to attempt to embody them in the data structure. However, many constraints are severe, such as an equation specifying an exact value for a variable.

Just as the data structure has bookkeeping demons to maintain its structural information, the constraint database also has associated demons to effect changes. Since the constraint database contains equations, these demons substitute known values and keep the representation of equations simplified and standardized.

The control level of the CMS solves equations, then instantiates the solutions in the database. The database may then suggest additional constraints that lead to further solutions or to more constraining equations. When all ease fails, the control level can resort to guessing. It may be noted that the problem can always be solved purely by guessing, the difficulty being that such an approach is computationally very costly.

Initialization Constraints

Initialization builds the data structures and fills the constraint database. Now the control structure selects a constraint. Its selection criteria focus on equations with fewest variables, and on variables with fewest possible values. The first constraint to be chosen is (C0 = 0). This is sent to the data structure, which makes the following modification:

```
C0   0
C1   0 1
C2   0 1
C3   0 1
C4   0 1
C5   0 1
```

Asserting the fact that (C0 = 0) changes the database. This change triggers the database rule:

```
            Ci = 0   <==>   Xi + Yi < 10
```

However, there are no values for Xi and Yi, since the 0th column is imaginary. The rule places (0 + 0 < 10) into the constraint database, which simplifies it to *true*. Since this rule does not contain variables, it is discarded by the constraint database simplification demons.

The constraint database now looks like this:

```
                         C5 = 0
                          a ≠ 0
                          e ≠ 0
                     d + b = h + 10C1
                C1 + c + g = b + 10C2
                C2 + b + f = c + 10C3
                C3 + a + e = f + 10C4
                        C4 = e + 10C5
      9000e + 900f + 90c + h = 1000a + 91b + d + 10g
```

The control structure selects another constraint, (C5 = 0). The cycle of assertion into the data structure, demon triggering, and simplification of constraint equations continues. After (a ≠ 0) and (e ≠ 0) are asserted, the information structure looks like this:

```
                S    a    1 2 3 4 5 6 7 8 9
                E    b  0 1 2 3 4 5 6 7 8 9
                N    c  0 1 2 3 4 5 6 7 8 9
                D    d  0 1 2 3 4 5 6 7 8 9
                M    e    1 2 3 4 5 6 7 8 9
                O    f  0 1 2 3 4 5 6 7 8 9
                R    g  0 1 2 3 4 5 6 7 8 9
                Y    h  0 1 2 3 4 5 6 7 8 9

                C0   0
                C1   0 1
                C2   0 1
                C3   0 1
                C4   0 1
                C5   0

                     d + b = h + 10C1
                C1 + c + g = b + 10C2
                C2 + b + f = c + 10C3
                C3 + a + e = f + 10C4
                        C4 = e
      9000e + 900f + 90c + h = 1000a + 91b + d + 10g
```

Equality Constraints

The next constraint to be selected is (C4 = e). Recall that each variable represents *a set of possibilities* rather than a point value. Thus, the knowledge that two variables are equal means that they both have the same possible value. This computational knowledge is contained in the data structure, and is enacted by the data structure when it receives the constraint to embody. Equality of variables is defined as the set intersection of possibilities. To embody this equation is to constrain the choice of value for C4 and for *e* to be the set intersection of their possibilities.

```
{0 1} intersect {1 2 3 4 5 6 7 8 9}   ==>   {1}
```

Here the system has identified a unique value for both variables. The state of the updated data structure becomes:

```
S    a      2 3 4 5 6 7 8 9
E    b  0   2 3 4 5 6 7 8 9
N    c  0   2 3 4 5 6 7 8 9
D    d  0   2 3 4 5 6 7 8 9
M    e    1
O    f  0   2 3 4 5 6 7 8 9
R    g  0   2 3 4 5 6 7 8 9
Y    h  0   2 3 4 5 6 7 8 9

C0   0
C1   0 1
C2   0 1
C3   0 1
C4     1
C5   0
```

The unique values discovered by the data structure are then passed back to the constraint database for updating. Since a Carry has been assigned a value, the carry rules also fire, resulting in a constraint being asserted:

```
Ci = 1   <==>   Xi + Yi >  9
C4 = 1    ==> a + e > 9
e = 1     ==> a > 8
```

The state of constraint database is now:

```
                    a > 8
                d + b = h + 10C1
            C1 + c + g = b + 10C2
            C2 + b + f = c + 10C3
                C3 + a = f + 9
  9000 + 900f + 90c + h = 1000a + 91b + d + 10g
```

Note that the solved equation, (C4 = e), once it has been embodied in the data structure, is removed from the constraint base. Note also the ability of the constraint database to deal with inequalities as well as equations.

Inequality Constraints

The next constraint selected is (a > 8). When this constraint is embodied in the data structure, *a* is left with the unique value 9. The result is communicated to the constraint database, resulting in the following information structure:

```
      S    a                    9
      E    b  0   2 3 4 5 6 7 8
      N    c  0   2 3 4 5 6 7 8
      D    d  0   2 3 4 5 6 7 8
      M    e    1
      O    f  0   2 3 4 5 6 7 8
      R    g  0   2 3 4 5 6 7 8
      Y    h  0   2 3 4 5 6 7 8

      C0   0
      C1   0 1
      C2   0 1
      C3   0 1
      C4     1
      C5   0

         d + b = h + 10C1
    C1 + c + g = b + 10C2
    C2 + b + f = c + 10C3
            C3 = f
900f + 90c + h = 91b + 10g + d
```

The CMS next selects the equation (C3 = f). This constraint is the result of simplification of the previous constraint that

```
C3 + a = f + 9
```

when (a = 9) is asserted.

The set intersection technique yields another unique value.

```
C3 = f

{0 1} intersect {0 2 3 4 5 6 7 8}   ==>   {0}
```

Therefore the constraints (C3 = 0) and (f = 0) are propagated to the data structure and to the constraint database, and a new constraint, (b < 10) is added by the interactive rules. The (b < 10) constraint originates when the carry rule for (C3 = 0) generates

b + f < 10, with f = 0.

The resulting information structure follows:

```
S    a                    9
E    b      2 3 4 5 6 7 8
N    c      2 3 4 5 6 7 8
D    d      2 3 4 5 6 7 8
M    e    1
O    f  0
R    g      2 3 4 5 6 7 8
Y    h      2 3 4 5 6 7 8

C0   0
C1   0 1
C2   0 1
C3   0
C4     1
C5   0

          b < 10
      d + b = h + 10C1
C1 + c + g = b + 10C2
    C2 + b = c
   90c + h = 91b + 10g + d
```

Next, the constraint (b < 10) is selected, and is discarded as a tautology.

Conditional Constraints

Again the CMS selects a constraint, this time the constraint must contain multiple variables.

C2 + b = c

{0 1} + {2 3 4 5 6 7 8} = {2 3 4 5 6 7 8}

The addition of possibility sets specifies multiple possibilities, delineating alternative worlds to explore. In the current case, possibility analysis yields no additional information. The control structure must then fall back onto *case analysis* as a technique. It returns the selected

constraint as a set of *conditional constraints*, one for each possibility of the particular variable with fewest possibilities.

```
if C2 = 0 then b = c
if C2 = 1 then 1 + b = c
```

Since the data structure knows the scope of possibilities for C2, we do not have to specify to the constraint base that 0 and 1 are the only possible values for C2.

Case analysis illustrates the technique for controlling search in the CMS. Conditional constraints are asserted and then explored. The conditional constraints follow the same selection rules as any constraint in the database. The two conditional equations replace the one that generated them. The control structure selects

```
if C2 = 0 then b = c
```

Both premise and conclusion are sent to the data structure. When the data structure receives a conditional constraint, it must clone itself as a hypothesis, embody both premise and conclusion, and continue analysis with the clone until the problem terminates. In the current example, the cloning never takes place The conclusion that (b = c) immediately fails, since it contradicts the knowledge of the data structure. When a conditional rule fails, the database asserts the negation of the premise as a constraint. Here it returns (C2 ≠ 0). That rule is selected next from the available constraints and embodied in the data structure. This leaves C2 with only one possible value, 1. (C2 = 1) is asserted as a constraint, selected, and embodied. It generates the constraint

```
c + g > 9
```

from data structure carry demons. The new constraint database is:

```
                c + g > 9
if C2 = 1, then c = b + 1
                d + b = h + 10C1
          C1 + c + g = b + 10
             90c + h = 91b + 10g + d
```

Note that the constraint database does not have to update the conditional constraint

```
if C2 = 1, then c = b + 1,  given C2 = 1
```

This is a bookkeeping job handled by the data structure.

If-changes Constraints

The new constraint, (c + g > 9), is selected, but fails to alter the data structure. It is maintained by the data structure as an *if-changes demon*. The if-changes demon is expressed as

if (c or g) changes, then c + g > 9

This constraint becomes relevant again only when any one of the variables changes value. Thus it is implemented as a demon that wakes up whenever a value of interest changes.

We have now seen the three ways that the CMS handles underspecified information

-- Tautologies are thrown away.

-- Equations are decomposed into cases, which may be analyzed at different times.

-- Equations that fail to embody constraints in the data structure are saved as if-changes demons. They will be reasserted whenever a relevant variable changes value.

The remaining conditional constraint,

if C2 = 1 then c = b + 1

is tested next. Asserting (C2 = 1) is a tautology. Asserting (c = b + 1) enables the possibilities comparison technique. The meaning of

{2 3 4 5 6 7 8} = {2 3 4 5 6 7 8} + 1

is that the *c-set* must contain all the members the *b-set*, when 1 is added to each. Conversely, the *b-set* must contain all the members of the *c-set* when 1 is subtracted from each one. This test eliminates

c = 2 and b = 8

When an equality does not resolve into a unique solution, as in the case of (c = b + 1), it is maintained by the data structure as an if-changes constraint. However, equalities can also be used to reduce the number of variables in the constraint database. In order to eliminate the information of the if-changes constraint, the specific equality, (c = b + 1), is used to eliminate the variable *c* from the set of constraint equations. In this case, *c* is replaced by (b + 1), yielding:

if (c or g) changes, then c + g > 9
if (b or c) changes, then c = b + 1

```
 d + b = h + 10C1
C1 + g = 9
90 + h = b + 10g + d
```

Although *c* has been removed from the constraints, its definition is still maintained by the if-changes demon.

The constraint (C1 + g = 9) is selected next. The case analysis control structure asserts two conditional facts:

if C1 = 0 then g = 9
if C1 = 1 then g = 8

The first of these constraints, when selected, is immediately rejected by the data structure, since *a* already has the value 9. As a consequence, (C1 ≠ 0) is posted, selected, and embodied. The result is that (C1 = 1) is posted, selected, and embodied.

The second conditional constraint produces two more variable bindings,

C1 = 1 and g = 8

and the carry constraint that

d + b > 9

Changing *g* triggers the if-changes rule (c + g > 9), which asserts (c > 1) into the constraint base. As well, when *g* takes the value 8, that possibility was removed from the possibilities of *c*. Two additional rules are posted by the if-changes demons that are triggered. Note that when an if-changes demon is posted as a constraint, it is removed from the data structure, since it is now incorporated by the constraint base. The current information structure:

```
S   a                   9
E   b     2 3 4 5 6 7
N   c       3 4 5 6 7
D   d     2 3 4 5 6 7
M   e   1
O   f  0
R   g                 8
Y   h     2 3 4 5 6 7
```

```
C0   0
C1     1
C2     1
C3   0
C4     1
C5   0
```

```
          c > 1
       d + b > 9
          c = b + 1
       d + b = h + 10
      10 + h = b + d
```

Tightening the Constraints

All values of the Carry variables are now known. The changes to the constraint base initiate a bookkeeping action. Two constraint equations are now identical, so one is eliminated.

Continuing the selection process, (c > 1) is selected and rejected as a tautology. Next (c = b + 1) is selected. Possibility analysis asserts that (b ≠ 7), making a single modification in the data structure. (c = b + 1) is re-established as an if-changes demon.

Next, the constraint (d + b > 9) is selected and analyzed:

```
d + b > 9

{2 3 4 5 6 7} + {2 3 4 5 6} > 9

      ==>   d ≠ {2 3}, and b ≠ 2
```

The constraint is then made into an if-changes demon. Eliminating a value for *b* triggers the (c = b + 1) if-changes demon. The rule is cycled back as a constraint, selected, analyzed to yield (c ≠ 3), and restored again as an if-changes demon.

Updating the information structure:

```
S    a                    9
E    b        3 4 5 6
N    c          4 5 6 7
D    d          4 5 6 7
M    e    1
O    f  0
R    g                   8
Y    h      2 3 4 5 6 7
```

```
        if (b or c) changes, c = b + 1
        if (d or b) changes, d + b > 9

              d + b = h + 10
```

The only remaining equation is now used:

```
                  d + b = h + 10
  {4 5 6 7} + {3 4 5 6} = {2 3 4 5 6 7} + 10
  {4 5 6 7} + {3 4 5 6} = {12 13 14 15 16 17}

              ==>   h + 10 ≠ {14 15 16 17}
              ==>   h ≠ {4 5 6 7}
              ==>   d ≠ {4 5}
              ==>   b ≠ {3 4}
              ==>   c ≠ {4 5}
```

No combination of *(d + b)* sums to more than 13. Thus *h* cannot be greater than 3. Similarly, *d* must be greater than 5 and *b* must be greater than 4. The restrictions on *b* propagate to *c*, which must now be greater than 5. Changes in *b*, *c* and *d* trigger both if-changes demons. The current equation,

```
        d + b = h + 10
```

since it does not yield a unique solution, is stored as a new if-changes demon.

Updating the information structure yields:

```
              S    a                    9
              E    b            5 6
              N    c              6 7
              D    d              6 7
              M    e    1
              O    f  0
              R    g                 8
              Y    h      2 3

  if (d or b or h) changes, d + b = h + 10

              c = b + 1
            d + b > 9
```

Concluding with Search

Both remaining rules fail to change the data structure. (c = b + 1) is again stored as an if-changes demon. (d + b > 9) is discarded as a tautology. There are no more constraints. In the final phase of analysis, the system resorts to case analysis on if-changes demons. The case analysis proceeds to verify solutions by expanding the if-changes demons into conditional constraints:

```
        c = b + 1

    {6 7} = {5 6} + 1

          ==>   if c = 6 then b = 5
                if c = 7 then b = 6
```

The first conditional assertion is embodied in a cloned data structure. This is the first (and only) resort to explicit search by the CMS. Asserting (c = 6) and (b = 5) leaves only one remaining value for *d*, (d = 7). This is then asserted as a constraint, along with the if-changes demon triggered by the change in *b*.

```
    7 + 5 = h + 10
```

yields

```
    h = 2
```

The cloned data structure now embodies a solution. It remains to clean up the second conditional,

```
    if c = 7 then b = 6
```

When the second conditional constraint is selected, it generates a contradiction, eliminating the cloned data structure it created.

```
    c = 7 and b = 6

          ==>  d = { }                FAIL
```

The failure causes the constraint (c ≠ 7) to be posted. It is then selected and embodied in all other (possibly cloned) data structures. There is only one in this case. Eliminating the possibility of *c* being 7 constraints it to the particular value 6. The if-changes demons then assert that (b = 5), forcing (d = 7). Since *b* changes, the remaining if-changes rule,

```
    if (d or b or h) changes, then d + b = h + 10
```

becomes a constraint, and the steps of the previous successful solution are retraced, resulting in (h = 2).

The final data structure looks like this:

S	a	9
E	b	5
N	c	6
D	d	7
M	e	1
O	f	0
R	g	8
Y	h	2

Since each variable has a single value, and no constraints or definitions remain, the solution is unique. That solution is:

SEND + MORE = MONEY

9567 + 1085 = 10652

CONCLUSION

The knowledge engineering task inherent in the automation of design is immense. Not only must the functionality of the design be formally modeled, but also the job of the designer must be modeled. For the current state of the art in expert systems and related AI technologies, knowledge engineering is limited to domains with a well understood technology that can be communicated to a novice within a week. Thus, it should be expected that the automation of the design process is many years away. We might expect that an extended effort will be successful in formalizing major functions of design. However, formalizing the skills of a designer is likely to be impossible within the next twenty years. This observation argues for a mixed-initiative system, for which the difficult tasks can be solved jointly by the computational system and the human designer.

The course of development of the fully automated design environment should be segmented. First, localized areas of automation must be identified and implemented. As an example, CAD workstations for the design of the three-dimensional geometry represent such an *automation cluster*. Next, each automation cluster must be integrated with AI technologies. In the example, a constraint management system might serve to criticize designs for violation of specifications. Only after several of these *intelligent automation clusters* have been developed and placed in use should an overall integration be attempted. The information gained from experience over several years with automation clusters must feed into the overall integration, because it is

only through experience that designers can know the importance and the architecture of each piece. Experience with the automation of a functional cluster should be expected to change the architecture of the integrated system.

To provide an initial prototype of an automation tool, we have developed a constraint management system with broad functionality. This prototype tool should next be applied and refined in an application environment, in order to evaluate the effect of such a tool on the design process and the effect of modeling the design domain to fit the formalisms of the tool.

The long term prospects of the integration of AI into the design process depend upon the formalization of design tasks and advancements in AI. Certainly some AI tools that are currently available would have substantial impact on the ease of bookkeeping during the design process. However, their utility depends upon the automation of design data structures, the availability of automated design tools, and the coordination of these tools and databases. Such automated systems do not currently exist, and their development will be effort intensive. Even after such automated systems are in place, the formalization of the process for application of AI techniques will be expensive in knowledge engineering effort. It would not be unreasonable to budget hundreds of person-years to the development of a fully automated AI-based design workstation.

A useful and effective system can be achieved through a more modest investment in a mixed-initiative AI system within a design workstation. The constraint management system delineated in this report is an example.

APPENDIX: VEHICLE SEAT PLACEMENT DATABASE

This appendix includes the database for vehicle seat placement design that was used as a prototype development problem for ConMan.

The information in this appendix includes the *legend* of variable names and meanings, the equation database which defines variables, the constraint database which places limits on variables, and the values database, which specifies initial values for variables.

Legend

The legend provides short descriptions of variables used in the specification of the design. For the prototype domain, these include the following:

```
nsrp:  neutral seat reference point
dep:   design eye position
srp:   gear-shift reference point
trp:   brake reference point
hrl:   heel reference line
hvl:   horizontal vision line
dvl:   downward vision line
a:     vertical height from NRSP to DEP
b:     vertical height from NRSP to seat top
c:     vertical height from NRSP to heel line
d:     vertical seat adjustment above nrsp
dbar:  vertical seat adjustment below nrsp
e:     horizontal seat adjustment fore of nrsp
ebar:  horizontal seat adjustment aft of nrsp
f:     dep from front of seat
g:     trp in front of nsrp
gg:    trp above of nsrp
h:     srp in front of nsrp
hh:    srp above of nsrp
i:     min distance nsrp to panel
j:     same as i, but for portion over leg
k:     pedals (neutral) forward of nrsp
l:     forward travel of gear-shift
lbar:  aft travel of gear-shift
m:     forward travel of brake
mbar:  aft travel of brake
```

Equations

The equation database contains definitions of variables in terms of other existing variables. These equations represent the abstract design of the vehicle seat. The equation database for the prototype domain follows. There are three types of variables: dimensions, angles, and reference points. Within the equation database, these are not distinguished.

Reference Points and Lines:

```
hvl = a + nsrp-y

hrl = nsrp-y - c

nrsp-x = dep-x - (a * tan(ang-a)) + (f / cos(ang-a))

nrsp-y = hrl + c

dep-x = (a * tan(ang-a)) - (f / cos(ang-a)) + nsrp-x

dep-y = hvl

srp-x = h + nsrp-x

srp-y = hh + nsrp-y

trp-x = g + nsrp-x

trp-y = gg + nsrp-y
```

Dimensions:

```
a = dep-y - nsrp-y

c = nsrp-y - hrl

f = cos(ang-a) * (dep-x - (a * tan(ang-a)))

g = nrsp-x - trp-x

h = nrsp-x - srp-x

ph = (c + a - q) / (cos(pi / (2 + ang-e - ang-f)) * tan(ang-e)) -
(cos(pi / (2 + ang-e - ang-f)) / tan(pi / (2 + ang-e - ang-f)))

q = a + c - (ph * cos(pi / (2 + ang-e - ang-f)) * tan(ang-e)) -
(ph * cos(pi / (2 + ang-e - ang-f)) / tan(pi / (2 + ang-e - ang-f)))
```

```
r = q - (t * sin(pi / (2 + ang-e - ang-f)))

t = (q - r) * sin(pi / (2 + ang-e - ang-f))
```

Angles:

```
ang-a = x

ang-b = (pi / 2) + ang-a - ang-sp

ang-e = x

ang-f = x

ang-sp = ang-b - (pi / 2) - ang-a
```

Constraints

The constraint database is derived for lessons learned. Internal constraints on data types includes keeping all measurements greater than zero, and all angles between 0 and 2pi radians. (The greater-than-or-equal-to relation is expressed as ≥.)

```
d ≥ 2.5

dp ≥ 2.5

e ≥ 1.5

ep ≥ 1.5

a + c > 37

a + c < 41

abs(gg - hh) < 5

l + lp ≤ 7

m ≤ 5

mp ≤ 5

r ≥ 16

h + l < j
```

```
q ≥ 16

r ≥ 16

ang-e ≥ 11

abs(ang-f - (pi / 2)) < 5
```

Values

The values database contains initial values for variables, as determined from lessons learned, and from the definition of reference points.

```
nsrp-x = 0
nsrp-y = 0
a      = 31.5
b      = 40
c      = 8.5
f      = 13
g      = 20
gg     = 13.5
h      = 19
hh     = 13.5
k      = 36.25
ang-a  = 13
ang-b  = 97
ang-sp = 6
```

APPENDIX: TRANSPARENCIES FOR CONSTRAINT MANAGEMENT TALK

SLIDE 1: (title)

"Applying AI to the Design Process"

SLIDE 2: (overview)

* The Design Problem
* Classes of Constraints
* Knowledge Representation for Constraint Management
* Limiting Search
* Constraint Explanation and Inversion
* Technical Details and Current Status
* Related Work

SLIDE 3: "The Design Problem"

* extraordinarily complex
* requires cooperating team of designers
* design developed as hierarchy of design abstractions
* design inertia is characteristic phenomenon
* design activities occurring asynchronously & in parallel
* constraints pervade the entire design process
* communication & conflict resolution are thorny issues
* audit trail maintenance both crucial and difficult

We propose to provide a mechanism to facilitate audit trail maintenance, designer communication, conflict resolution, and constraint management.

SLIDE 4: "Classes of Constraints"

Constraints may be classified along many dimensions:

1. Partitionable & Non-Partitionable

   * partitionable:

     e.g. structural partitioning of available resources,
     such as space, time, power etc.

   * non-partitionable:

     e.g. design specs, describing component performance

2. Hard & Soft

   * hard:

     e.g. exact specifications like required geometry

   * soft:

     e.g. guidelines, specifications, estimates

3. Spatial, Temporal & Logical

   * spatial:

     "How large can I make display2
     without violating other design constraints?"

   * temporal:

     "Given design timelines,
     how much time can be allowed to process display1
     prior to initiation of function2?"

   * logical:

     "Since display1 is needed during activity1,
     will existing constraints impact display1
     visibility during such activities?"

Regardless of the type of constraint, constraints must be propagated both upwards and downwards in the design hierarchy.

SLIDE 5: "Knowledge Representation for Constraint Management"

* proposed data structure is hierarchical network of frames

  < diagram of a simple design hierarchy inserted here >

* we have both generic frames and instances of these frames

* frames can also contain attached properties such as:

  -- constraints
  -- slot ordering rules
  (which slots should be filled first ?)
  -- heuristic-rules

SLIDE 6: "Limiting Search"

* design alteration amounts to frame editing

* potential for constraint violation occurs
  whenever a slot is filled or edited

* because of inheritance, a seemingly simple change
  high in the design hierarchy can spawn a plethora
  of potential violations at a more detailed level

* mechanism must be found for limiting search

* one approach is to limit search
  to certain types of constraints:

  -- search only for violations of those types
  of constraints associated with the type
  of the slot being filled.

  -- search for violations of timely constraints
  (i.e. those for which all of the associated slots
  have been filled) only.

  - search according to design context

  - search according to hardness

SLIDE 7: "Constraint Explanation/Inversion"

* useful to provide designer with explanation of mechanism underlying a constraint violation

* a backtracking scheme, coupled with both (simple) natural language and network display techniques can facilitate this.

* given a design frame with only one unfilled slot, the designer can be assisted (in a mixed-initiative style) via the inversion of any attached constraints

* the designer can thus obtain a range of possible values for an unfilled slot, such that no existing related constraints will be violated.

SLIDE 8: "Technical Details and Current Status"

SLIDE 9: "Related Work"