Design and Implementation of a Qualitative Constraint Satisfaction System

P. Thyagarajan
and
Arthur M. Farley

CIS-TR-87-03
March 19, 1987

Abstract

This report describes the design and implementation of a constraint-based environment for modeling the structural description of physical systems in qualitative space. The variables and constraints define the structural description and the constraint propagation derives the behavioral descriptions. The constraint propagation finds an interval value for each variable by shrinking the initial interval values of the variables, such that all the solutions are captured by the final values of the variables. During the propagation of constraints, justifications are built for each variable so to give an *explanation* for a behavior of the system. The constraint system is incorporated with an Assumption-based Truth Maintenance System (ATMS) to avoid recomputation.

Department of Computer and Information Science
University of Oregon

*Supported in part by grant NAG-2-383 from the National Aeronautics and Space Administration (NASA) through Ames Research Center, Moffett Field, CA 94035

TABLE OF CONTENTS

CHAPTER I

INTRODUCTION

The research we shall discuss here is an attempt to build a constraint-based environment from the ground up that will be useful for modeling the structural description of physical systems in qualitative space.

An expert system is often a *shallow model* of its application domain, in the sense that conclusions are drawn directly from the observable features of the situation presented. Researchers have long felt that a genuinely expert performance must also rest on the knowledge of *deep models*, in which an underlying mechanism, whose state variables may not be directly observable, accounts for the observable facts [11]. The qualitative simulation utilizes such deep models to derive a system behavior from its structure. The concept of qualitative simulation derives from the common intuition of simulating a machine in the mind's eye.

To build a system for qualitative simulation, one of the steps is to decide on a *simulation language*. A constraint language seems to be a natural choice for qualitative simulation; as the system components can be modeled as maintaining constraints among their state variables. This calls for a constraint system to interpret the constraint language, to propagate constraint implications and to determine constraint solutions. The computational model of constraints requires solution of constraint satisfaction algorithm which generally is a NP-complete problem [13].

The variables and constraints define the structural description; and the constraint propagation derives the behavioral descriptions. Unlike constraint satisfaction, which finds an assignment to all the variables in the system such that all the constraints are satisfied; constraint propagation finds an interval value for each variable by shrinking the initial interval values of the variables, such that all the solutions are captured by the final values of the variables. The constraint system has an associated Assumption-based Truth Maintenace System (ATMS) to avoid value recomputation.

The rest of this chapter is an introduction to constraint propagation and satisfaction, qualitative simulation, and assumption-based truth maintenance. The Qualitative Constraint Satisfaction System is written in Common-Lisp for the Symbolics and Sun workstations. The object-oriented programming technique (flavor package) is extensively used in its design and implementation. Finally a brief description of related works and the scope of the thesis is discussed.

1. Constraints

Any language is basically a means of communicating ideas. Generally a language is specialized for communicating a class of ideas effectively. For example, most computer languages are designed for expressing algorithms. They are optimized for communicating imperative and procedural notions. The theme of a constraint language is declarative [18]. This is well suited for expressing the description of structural relationships and physical situations.

There are two important aspects of constraints which together make the constraints unique [15]. First, a constraint is the declarative statement of a relationship. If the sum of the quantities X, Y and Z is constrained to be zero, then there is a stated relationship between the three quantities, $X + Y + Z = 0$. There may be various interpretations of this constraint for the sake of convenience, such as $X = -(Y + Z)$, $X + Y = -Z$ etc. Second, a constraint is a computational device for enforcing a relationship. The constraint does not have any designated input or output associated with it. When the value of any quantity changes within a constraint, due to an external or global change, the constraint may demand modification of the values of other quantities in the constraint, to enforce the relationship defined by it. The enforcement of relationships in a constraint is called the *local propagation* of a constraint.

A by-product of constraint propagation, one of particular importance for our purpose, is that of a network of dependencies that maintains the history of computation. This history can be used to identify dependency information: what values are derived from what other values. This dependency information can be used to construct a dependency-directed graph to provide an explanation of the computation.

2. Qualitative Simulation

We have a natural tendency to try to understand how things work and to explain that understanding to others. It is possible to derive a qualitative description of a behavior of a system from the qualitative description of its structure. The structural description consists of individual variables that characterize the system; their interaction is derived from the components and connections within the physical device. A behavioral description describes the potential *activities* of the system. The functional description reveals the intended *purpose* of a structural component in producing the behavior of a system. Let us consider a clock. The structural description would include variables and constraints governing the hands, gears and springs etc. The behavioral description would include the movement of the hands caused by the rotations of the gears, which is in turn caused by the unwinding of the spring. The functional description of the clock is the purpose of the clock: to maintain and show the correct time.

Quantitative simulation is one way of producing behaviors of a system from its structural description. This is conceptually a simple simulation, computing the values of all variables only at important points in time. The structural descriptions given as input to qualitative and quantitative simulations are of different types. Quantitative simulation

requires complete and exact knowledge of structural descriptions of the system, whereas for qualitative simulation the structure is defined in terms of differential equations which provide useful but partial knowledge about the system.

Qualitative simulation can be viewed as an inference process for producing behavioral descriptions from the qualitative structural description of a system [11]. Given a set of variables, a set of qualitative constraints, and a few assumed variable values; the inference process tries to satisfy all the constraints by assigning consistent values to unassumed variables.

3. Assumption-based Truth Maintenance System

A Truth Maintenance System (TMS) [17,6] serves three roles in a reasoning system[1]. First, it caches all the inferences made by the problem solver; this avoids unnecessary recomputation. Second, it supports non-monotonic inferences, i.e., inferences that cause previously valid statements to become invalid or vice versa. Third, it ensures that the database is contradiction-free. De Kleer [3] has proposed an assumption-based TMS (ATMS) which allows context switching and has a coherent interface between the TMS and the problem solver without giving up exhaustivity. According to De Kleer [3], unlike previous truth maintenance systems which primarily manipulated justifications, ATMS manipulates assumption sets. As a consequence, it is possible to work effectively and efficiently with inconsistent information; the context switching is computationally (storage regarded) free, and most of the backtracking is avoided. These capabilities motivate a different kind of problem-solving architecture in which multiple potential solutions are explored simultaneously. This architecture is particularly well-suited for tasks where a reasonable fraction of the potential solutions must be explored.

In this constraint system, an ATMS is used to cache all the inferences made by the system and to realize context switching, enabling the simultaneous exploration of multiple solutions.

4. Related Research

The two different research issues considered in this thesis are: Constraint Satisfaction Systems and Qualitative Simulation. The following are works that involve one of the above research issues:

Alan Borning's ThingLab system [1] is a continuation of Sketchpad [19]. It is a drawing and simulation system that allows the definition of arbitrary objects and constraints. Users sketch a design interactively using graphics and inform ThingLab what the parts were and how they behaved; where upon ThingLab then performs the simulation. The constraint satisfaction is in two phases: *planning* and *execution.* In the planning phase, for every change the user makes to any object[2], the system constructs Smalltalk routines

[1] This consists of a problem solver which draws inferences from TMS

[2] Objects refers to graphical pictures that can be changed by dragging it on the screen.

to satisfy all the constraints incorporating the change. The execution phase invokes all the constructed routines. ThingLab exploits Smalltalk's hierarchical structure in defining hierarchical objects.

Sussman and Steele [18] present a language for the construction of hierarchical constraint networks and local propagation techniques for constraint satisfaction. Dependency analysis is used to spot and track down inconsistent subsets of the constraint set. They also briefly discuss algebraic manipulations of constraint networks.

Steele's thesis [15] is an examination of the methods for implementing constraint systems. His thesis also presents a constraint language and its implementation. This system can explain its decisions. Conflict resolution in a constraint network is done by retraction and dependency-directed backtracking. Constructs are introduced for making assumptions about the values of certain variable. Due to global considerations, guesses may turn out to be inconsistent, and so *nogood sets* are introduced to record the forbidden guesses.

Gosling's thesis [10] presents a progression of constraint-satisfaction algorithms with analyses and emphasizes that no universally good satisfaction algorithm can be built. Hence special properties of the situation at hand must be used when constructing an algorithm. The example constraint system that is presented is for an interactive, graphical layout system. The algebraic transformation of sets of constraints is described with algorithms for identifying difficult subregions of a constraint graph and replacing them with transformed and simplified constraints.

5. Scope of the Thesis

The major part of the thesis is concerned with constraint propagation over a qualitative space as basis for modeling physical systems. It is my intent to demonstrate a qualitative constraint system that can be used to model physical systems in restricted domains.

This thesis provides an environment for modeling physical systems and performing qualitative simulations of their behaviors. The system's representation reflects a hierarchical approach by dividing the system into components and connectors. Each component has a set of variables and constraints describing its *desired* behaviors. The connectors are used to represent the interconnections among components. If all the constraints of a component are satisfied by the values of its variables, then the behavior described by the values of the variables is one of the *desired* behaviors of the component. During the propagation of constraints, justifications are built for each variable, thus giving *explanations* for the behavior of the component or system.

Our method of constraint propagation utilizes variables with ranges for values. This is different from the usual constraint propagation method, which finds an assignment of point values to variables that is consistent with the constraints. An ATMS is also introduced in the constraint system, so an inference is never re-computed. The ATMS makes the system more efficient and suitable for an interactive environment. All the design and

implementation details will be discussed extensively as appropriate.

The thesis is divided into three parts:

Constraint System: Describes how to build a constraint system and discusses the various operations that could be performed on the constraint system interactively. It also gives the limitations of the constraint system.

Constraint Implementation: Deals with the implementation details of the system. All the major operations are described at the implementation level. The data structures and algorithms are explicated with lisp code and discussions.

Modeling: Mapping constraint systems to physical systems. Two examples are considered for modeling: a pipe valve and cardiovascular system.

CHAPTER II

CONSTRAINT SYSTEM

A constraint system consists of a set of constraints and a constraint satisfaction mechanism. The task of the satisfaction mechanism is to find a set of values that satisfy the given set of constraints. The general satisfaction problem is NP-complete [13]. To avoid this computational complexity, one should either take a approximation approach [1,19], a restricted problem domain approach [10,13], or a combination of the two in constructing the satisfaction mechanism.

The constraint system introduced here takes the restricted problem domain approach. It assumes that the solution space of the constraint system in this domain is a convex space. A *solution* is a set of values with the corresponding variables which satisfies all the constraints in a constraint system. A *convex space* is a n-dimensional space in which for any two points in the given space, all points on the line joining those two points are also in that space. The dimension of the convex solution space is equal to the number of variables in the constraint system. From the property of convex space, we derive the fact that if there are two solutions, S_1 and S_2, for a constraint system, and V is a variable with value X_1 in S_1 and X_2 in S_2, then for any value between X_1 and X_2 for the variable V, there exists a solution. This result for convex solution space is used in constructing the satisfaction mechanism for this constraint system.

In this chapter, we introduce the constraint system. The presentation is in three parts - definition language, operation language and limitations of the constraint system. We assume that the reader is familiar with flavors [8]. As mentioned earlier, the whole system is built according to an object-oriented paradigm using the flavor package of Zeta-Lisp.

1. Definition Language

This language provides the basic features to define a constraint system. In order to define a constraint system, one has to first define the type of *qualitative space* over which the *values* of the *variables* are defined and then define the *constraints* over the variables.

1.1. Qualitative Space

The qualitative space defines the type of a data object or value. A qualitative space is an interval space represented by sequence of symbolic points in ascending order. An

```
(defqual-space integer-range
     :type integer-space    ; integer-space is a pre-defined qualitative space
     :value-set (-20 -15 -10 -5 0 5 10 15 20))
; It defines a qualitative space, integer-range which inherits the arithmetic
; operations from the qualitative space, integer-space.

(defvariable X :type integer-range)
(defvariable Y :type integer-range)
; It defines two qualitative variables. The type indicates qualitative space
; for the variables.

(defconstraint first-law
     :lhs (+ X Y)
     :comparator >
     :rhs 10)
; This represents a constraint: X + Y > 10. This states that sum of the values of X
; and Y must always be greater than value 10 for all behaviors of the system.
```

Figure 1: Declarations of: Qualitative Space, Variable and Constraint.

instance of a qualitative space may be $(a\ b\ c\ d\ e\ f)$, such that $a < b < c \ldots < f$. This allows us to represent any kind of value space symbolically. But it also requires arithmetic operators to be defined over the symbolic value space for the computational purposes. This creates restrictions on the qualitative space, as it is not simple to define $a + b$ and $a - b$, let alone $a * b$ and a/b. Difficulty arises due to the limited symbols in the qualitative space. What happens if $c + d$ results in a value that is greater than e and less than f ? There are two options, either a new symbol, representing the new value, can be inserted in the qualitative space at the appropriate position or the new value can be represented as an interval between e and f. There is another problem: representing a value greater than f. One solution is to treat a as minus infinity and f as plus infinity, but this introduces more ambiguities into the system. The approach depends upon the domain.

There are two qualitative spaces pre-defined in the constraint system - *integer-space* and *+0-space* [See definitions in appendix] but users may define their own qualitative space. *Integer-space* introduces new symbols into the qualitative space whenever a new value results from computations during constraint propagation, whereas *+0-space* uses the other approach.

Let us consider a qualitative space, integer-range, defined in Figure 1. The type *integer-space* is a flavor name which inherits a generic qualitative space, *qual-space*. All user-defined qualitative spaces must inherit this generic qualitative space and need at least two methods defined over the space: *:plus*, which takes two symbols from the space and returns the sum as another symbol of the space, and *:minus*, which similarly takes two symbols and returns the difference. The value of *:value-set* in Figure 1 is a list of ordered symbolic points in ascending order. *Integer-range* defines an interval space of integers between -20 to 20. Figure 2 gives a template for user defined qualitative space.

```
(defflavor <flavor-name>
    ()              ; user may define instance variables for :plus and :minus
    (qual-space)    ; inherits flavor object, qual-space
        :inittabel-instance-variables
        :settable-instance-variables
        :gettable-instance-variables)

; For a + b
(defmethod (<flavor-name>:plus)(a b)
    <body>)

; For a - b
(defmethod (<flavor-name>:minus)(a b)
    <body>)
```

Figure 2: Template for a Qualitative Space definition.

1.2. Variable

In modeling a system, each characteristic value of the system is associated with a name, which represents a variable. When defining a variable, one has to specify its value type and the qualitative space from which the variable will take values. The variables are also called qualitative variables. Figure 1 gives the declaration of variable X.

1.3. Variable Value

The value of a qualitative variable is defined to be an interval; in the degenerate case, it may be a point. An interval is represented by two values from the qualitative space. For example a value between point 5 and point 15 that includes point 5 but excludes point 15 is represented as (5 (<15))[3]. See Figure 3 for examples of values over the qualitative space defined in Figure 1.

When a value is defined as an interval, the actual value can be any point or sub-interval from the qualitative space that lies within the interval. This is how *ambiguities* in the target system are represented in the constraint system. We expect that when two ambiguous values are composed into one by an operator, the resulting ambiguity of the value will be more than the ambiguity of either initial values. This is clear from the definitions of the two arithmetic operators. Since the values are intervals, the usual definitions of *plus* and *minus* do not hold. Let us consider two variables V_1 and V_2 having values as follows: $V_1 = (a\ b)$ and $V_2 = (c\ d)$, where a, b, c and d are symbols in a qualitative space.
Following are the definitions[4]:

Plus: $V_1 + V_2 = (a\ b) + (c\ d) = (a + c\ b + d)$

[3]This is NOT an internal representation of the value.

[4]These are *plus* and *minus* for variable values - which are evaluated and is different from the *:plus* and *:minus* operators defined by the user while defining a new qualitative space.

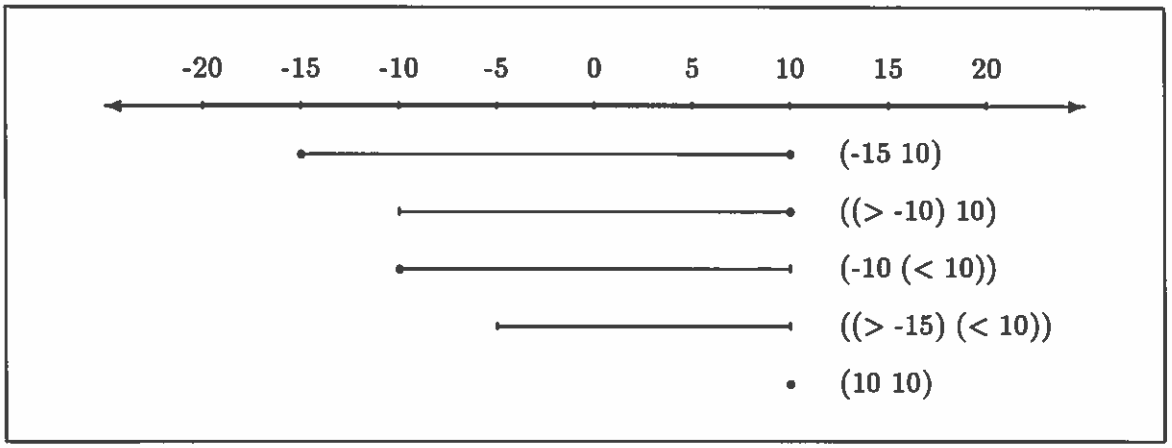

Figure 3: Examples of Value representations.

Minus: $V_1 - V_2 = (a\ b) - (c\ d) = (a - d\ b - c)$

1.4. Constraint

The constraints are ordinal relationships between expressions over variables whose values are intervals in a qualitative space. An ordinal relationship is one of >, >=, =, < or <=. An expression is a simple expression, such as A, or a prefix notational arithmetic expression, such as (+ A B). The operators allowed are only + and −. These operators can take more than two arguments, for example A + B + C is expressed as (+ A B C) and A − B − C is expressed as (− A B C). See Figure 1, for a declaration of a constraint. Each constraint has a left-hand-side (LHS), right-hand-side (RHS) and a comparator. The LHS and RHS are prefix arithmetic expressions. The comparator is one of the ordinal relationship.

2. Operation Language

This language describes various operations that can be performed by the constraint system. The major operators are *set environment, propagate constraints, setup operating region* and *get solution space.*

Consider a trivial example of a system, as shown in Figure 4. This system is simple but allows us to demonstrate the various operations that can be performed by the constraint system. Let us assume that variables currently have the following values:

X = (-20 20) Y = (-20 20)

To check whether a constraint is satisfied, the LHS and RHS of the constraint are evaluated. Then the system checks whether the ordinal relationship defined by the comparator can be satisfied for the evaluated values of LHS and RHS. There are three possible outcomes:

```
(defqual-space integer-range
     :type integer-space
     :value-set (-20 -15 -10 -5 0 4 5 10 15 20))

(defvariable X :type integer-range)
(defvariable Y :type integer-range)

(defconstraint C-ONE
     :lhs (+ X Y)
     :comparator >
     :rhs 5)
(defconstraint C-TWO
     :lhs (- X Y)
     :comparator >
     :rhs 3)
```

Figure 4: An Example.

1. *not satisfied* - if there do not exist any sub-intervals in the LHS and RHS, such that the relationship holds; for example, (5 10) > (15 20) cannot be satisfied.

2. *possibly satisfied* - if there exist sub-intervals in the LHS and RHS, such that the relationship holds and there also exist sub-intervals in the LHS and RHS respectively, such that the relationship does not hold; for example, (5 15) > (10 20) is possibly satisfied because (15 15) > (10 10) is satisfied and (5 10) > (10 20) is not satisfied.

3. *definitely satisfied* - if neither of the above; that is, for all possible two sub-intervals from LHS and RHS respectively, the relationship holds; for example, (5 10) < (15 20) is satisfied for any sub-intervals of the above values.

A constraint is *satisfied* if the ordinal relationship is *possibly satisfied* or *definitely satisfied.* For the initial values of the variables in our example, constraints C-ONE and C-TWO are satisfied:

C-ONE	C-TWO
(+ X Y) > A	(− X Y) > B
(+ (-20 20) (-20 20)) > (5 5)	(− (-20 20) (-20 20)) > (3 3)
(-40 40) > (5 5)	(-40 40) > (3 3)

For both the constraints, the ordinal relationships is *possibly satisfied.*

In the rest of the chapter, any variable or constraint refers back to this example, unless otherwise specified. Now we look at the basic operations.

2.1. Set Environment

An *assumption* states that a particular variable takes on a given interval value from its qualitative space. An assumption does not take effect and the value of the variable is

```
(defassumption Y-assumption
     :variable Y
     :low-value 5
     :high-value (< 15))
;It states that variable, Y is assumed to be an interval 5 (including) to 15 (excluding).

(defenvironment Y-environment :assumptions (Y-assumption))
```

Figure 5: Declaration of: Assumption and Environment.

not changed to the given interval value until an environment is selected. See Figure 5 for a declaration of an assumption.

In qualitative simulation or diagnosis, we are generally interested in a particular set of behaviors of the system. This can be acheived if we can *assume* the values of a subset of the variables in the system, such that the assumptions define only the set of behaviors we are interested in examining. An *environment*[5] is such a set of assumptions. See Figure 5 for a declaration of an environment.

If one is interested in examining the system for different environments, then all the required environments are declared initially. Through the *set environment* operator, an environment can be made *current*. There are three possible outcome of set environment operation:

1. If the environment is already present in ATMS, then the variables in the system are set to the corresponding values contained in the environment.

2. If there are environments in ATMS whose assumption-sets are subsets of the assumption-set of the environment being set, then the variables in the system are set to the intersection of the corresponding values contained in all the subset environments.

3. If none of the above, then set the variables in the assumption-set of the environment to the assumed values and the rest of the variables to the full interval range of its qualitative space.

At any instant, there is a unique current environment. This allows the dynamic switching of environments, to study and contrast behaviors.

2.2. Propagate Constraints

Constraint propagation shrinks the interval value of each variable in the constraint system to a minimum interval value for which all the system constraints in the system are satisfied. The propagation continues until either one of the constraints is not satisfied or none of the value of the variables can be shrunk further. The constraint propagation

[5] Environment is same as *context* used by researchers in fault diagnosis [14]

applies a *pruning*[6] method for shrinking intervals of the variables. It is assumed that all variables have an initial value before starting constraint propagation. The constraint propagation *fails* if any of the constraints are not satisfied during the propagation.

Formally, pruning a variable X is accomplished by rewriting the constraint C-ONE, solving for X, as X > (− A Y). The LHS is the variable to be pruned and the RHS is an expression. Then the RHS can be evaluated and compared with X for pruning. For example:

X	>	(−A Y)
(-20 20)	>	(−(5 5) (-20 20))
(-20 20)	>	(-15 25)

New pruned value of X is ((> -15) 20), by intersecting (-20 20) with (-15 25)

The new value or *pruned* value of variable X is a sub-interval of its original value interval such that for every point in the new interval there is a corresponding value in the RHS that maintains the ordinal relationship. During propagation an attempt is made to prune all variables of each constraint. The order in which the variables are pruned within a constraint does not affect the final values of the variables after propagation. This agrees with the multi-directionality of constraints, i.e., there is no input or output to a constraint. Since we have assumed that the values are continuous, pruning of a variable moves the lower limit up and/or moves the upper limit down.

After the completion of constraint propagation in the example of Figure 4, the value of X could be ((> -12) 20) and Y could be ((> -15) (< 17)) (as defined in *integer-space*, new points such as -12, -15 and 17 are inserted into the qualitative space). Figure 6 shows the region after constraint propagation. Let us closely examine this region. Consider the sub-interval ((> 12) 0) (marked as I_x) of the value of variable X. For this sub-interval of the variable X, constraint C-ONE is satisfied for the sub-interval (5 (< 17)) (marked as I_{y1}) of the variable Y. Similarly, for the same sub-interval of the variable X, the constraint C-TWO is satisfied for the sub-interval ((> -15) 2) (marked as I_{y2}) of the variable Y. The intervals I_{y1} and I_{y2} are disjoint, therefore there is no region for the simultaneous solution of C-ONE and C-TWO when X has value I_x. This shows the limitation of *constraint propagation*: it only considers individual constraints so it provides an over-generalized region within which all the constraints may be satisfied.

During constraint propagation, when the value of a variable gets modified, it is recorded along with the constraint that modified the value and the values of the variables in the constraint at that instant. This justification does not reflect the causal behavior of the system but provides a history of computation. Constraint propagation is the basic operation of the constraint satisfaction mechanism in this system.

[6]Even though *pruning* and *interval shrinking* means essentially the same thing, we use *pruning* for a particular type of interval shrinking which is described above.

2.3. Setup Operating Region

Unlike constraint propagation, which only considers the satisfaction of individual constraints, construction of the operating region considers the satisfaction of composed constraints. Composed constraint satisfaction insures that if any two constraints can be satisfied then they can be satisfied for any sub-interval value of any variable in either constraint. As this operation is more expensive than constraint propagation, setting up operating regions only uses constraint propagation.

An *operating region* is defined as a set of interval values associated with the variables in a system such that if there is a solution, it is in the region defined by the values of the variables. This region is a subset of the region obtained from constraint propagation. Due to the limitations of constraint propagation (discussed in previous section), we cannot be sure that for any variable assumed any point or a sub-interval from its interval value in the operating region, there exist a solution for the new assumption. To determine the operating region of any system, each variable is assigned the whole interval value of its qualitative space. The first phase of pruning is done through constraint propagation. The second phase pruning mechanism, which is explained in the next chapter, computes the operating region. Figure 6 shows the operating region for the example being considered.

From Figure 6, it is clear that even though the operating region is a refinement of the region produced by constraint propagation, there are still sub-regions in which the constraints are not satisfied. We could say that *operating region* provides a *tight* cover with an easy representation of the cover, which is a set of interval values of all the variables in the system.

2.4. Get Solution Space

A qualitative behavior of a system is characterized by the set of values of the variables in the system, where each value is either a point or an indivisible interval. The *solution space* is the set of all qualitative behaviors of the system within the current environment. By qualitative behavior, we do not always mean one *actual* behavior of the system, because in an indivisible interval there are many points; and each one of them may define a *actual* behavior of the system. The idea we are trying to capture in a qualitative behavior is a class of actual behaviors; in the degenerate case, it may be one actual behavior. We assume that the qualitative space is defined such that the value symbols define the various classes of behaviors. According to Kuipers [12], these values in the qualitative space are termed *landmarks*. Even though Kuiper assumes that crossing a landmark may involve a new set of constraints governing the system, we consider the same set of constraints to hold true and restrict ourselves to such domains.

The solution space illustrated in Figure 6 is a sub-region of the operating region and a convex space. Single behaviors that corresponds to an actual behavior (that is, all the values are points) are indicated by a dark dot, and single behaviors corresponding to a class of actual behaviors (that is, values are intervals) are represented as dark rectangular figures.

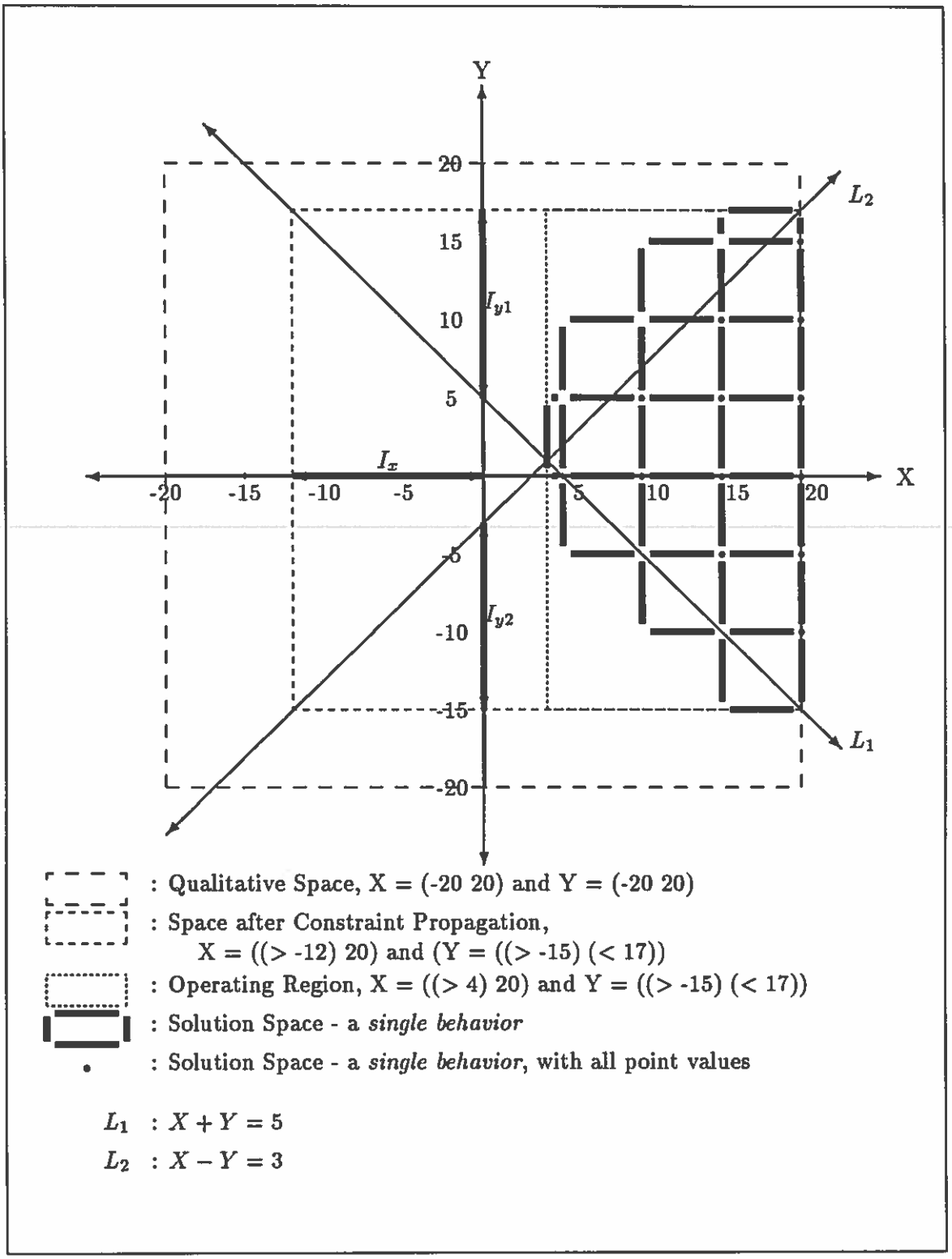

Figure 6: Solution space of an example.

2.5. Other Operations

There are other operations such as showing nogood environments, solutions and operating region; creating and listing qualitative spaces, variables, constraints, assumptions and environments; loading and saving the system; and initializing the system. Among these operations, we consider only showing nogood environments.

We have shown that to examine a certain set of behaviors an environment could be defined so that it captures the behaviors. It is possible that an environment does not capture any behaviors of the system. This implies that at least one constraint could not be satisfied during constraint propagation in that environment. Since the environment is a set of assumptions, for this set of initial assumptions there are no behaviors in the system. Such an environment is called a *nogood environment.* The nogood environments capture important information about the system which is not explicitly evident in a qualitative model. Qualitative simulation *generates* all possible behaviors of the system. By examining the nogood environments, we find conditions for which there is no possible behavior.

This operator lists all the nogood environments currently identified by the constraint system. In the example shown in Figure 4, one nogood environment would be the single assumption that the value of variable X is (-20 0). The set of all possible nogood environments for a given set of constraints is the complement of its solution space.

3. Limitations of the Constraint System

The constraints are ordinal relationships of expressions which involve only *plus* and *minus* arithmetic operators. These constraints cannot capture conditional (i.e., if-then-else) constraints. This is a major limitation for this constraint system. It is possible to extend this system to incorporate conditional constraints by having various sets of constraints in the system with each set having an initial set of conditions. All sets of constraints satisfying the intial set of conditions can be propagated independently by maintaining multiple images of variables.

CHAPTER III

CONSTRAINT IMPLEMENTATION

This chapter deals with the implementation details of the constraint system. It is built on top of a Lisp system in the dialect known as Common-Lisp [16]. The flavor package [8] is used extensively. The system was developed on the Symbolics workstation and is ported to the Sun workstation. The window package is used to build an interactive interface for the constraint system. There are popup windows and menus to make the system user-friendly. In the following implementation description, all of the interface parts are ignored.

This chapter includes a discussion of the definitions of all the objects used in the system, the various operations performed on these objects and the implementation details of the ATMS. To make the implementation description complete, Lisp code for the flavor definitions and the methods associated with the key operations are shown. In flavors, functions associated with an object are called methods and are represented by names prefixed by colons. The names of methods are in *slant* type style for easy reference.

1. Definition Representation

This section deals with the representation of the important data-structures of the constraint system. Each data-structure is a flavor definition, each instance is an object. The flavors described are qual-space, value, variable, constraint, assumption, environment, justification and constraint-system. The Lisp code definitions of the above objects are shown in Figures 7, 8 and 9. A brief description of each instance variable accompanies each flavor definition. In this section, only the non-trivial instance variables are described.

1.1. Qual-Space

value-set: This is a ordered list consisting of symbols representing the points and intervals of the qualitative space. The symbols are organized in ascending order. Depending upon the qualitative space definition, new symbols may be added to *value-set* during computation.

1.2. Value

type: An instance of a qual-space flavor, which is either integer-space, -0+space (predefined qualitative spaces) or a user-defined space. This specifies the qualitative space over which

```
(defflavor qual-space
    (symbol          ; user given name for the qualitative space
    value-set)       ; list of ordered points
    ()
    :inittable-instance-variables
    :settable-instance-variables
    :gettable-instance-variables)

(defflavor value
    (type            ; instance of a qualitative space flavor
    low-value        ; lower value of the interval
    high-value       ; higher value of the interval
    time-stamp)      ; time stamp when values assigned
    ()
    :inittable-instance-variables
    :gettable-instance-variables
    :settable-instance-variables)

(defflavor variable
    (symbol          ; name of the variable
    status           ; either assumed, computed, initialized, constant or unknown
    assumption       ; if status is assumed then it has the assumption instance
    value            ; instance of value corresponding to the current environment
    constraints      ; list of all constraints which involve this variable
    justification    ; justification for the value
    justification-flag) ; indicates whether to add justifications
    ()
    :inittable-instance-variables
    :gettable-instance-variables
    :settable-instance-variables)

(defflavor constraint
    (symbol          ; name of the constraint
    status           ; either consistent, contradiction or unknown
    variables        ; list of variables in the constraint
    generators       ; list of constraints in terms of each variable in the constraint
    comparator       ; comparision sign of the constraint, i.e., <, =, > etc.
    lhs              ; left hand side of the constraint
    rhs)             ; right hand side of the constraint
    ()
    :inittable-instance-variables
    :gettable-instance-variables
    :settable-instance-variables)
```

Figure 7: Flavor Definitions: Qual-Space, Value, Variable and Constriant.

```
(defflavor assumption
    (symbol          ; name of the assumption
    id               ; each assumption is associated with a number
    variable         ; variable instance
    value)           ; value instance of the variable that is assumed
    ()
    :inittable-instance-variables
    :settable-instance-variables
    :gettable-instance-variables)

(defflavor environment
    (symbol                ; name of the environment
    environment            ; integer value representing the assumption set
    var-val-justfn         ; list of variable, value and list of justfication instances
    partial-environments)  ; list of environment inst which are subsets of environment
    ()
    :initable-instance-variables
    :gettable-instance-variables
    :settable-instance-variables)

(defflavor justification
    (value          ; justification for this value-inst
    constraint      ; instance of constraint from which the value was derived
    var-value       ; list of (var-inst value-inst) for all vars in the above constraint
    assumptions     ; set of assumptions, represented by an integer
    reason)         ; non-nil if value is not due to any computation
    ()
    :initable-instance-variables
    :gettable-instance-variables
    :settable-instance-variables)
```

Figure 8: Flavor Definitions: Assumption, Environment and Justification.

```
(defflavor constraint-system
    (name                          ; name of the system
    window                         ; window instance assoc with it
    system-status                  ; either unknown, consistent or inconsistent
    popup-menu                     ; instance of popup menu for setting curr. env.
    qual-space-symbol-table        ;
    variable-symbol-table          ; symbol
    assumption-symbol-table        ; tables
    constraint-symbol-table        ;
    current-environment            ; current environment being considered
    nogood-environments            ; list of environments which results in contradiction
    assumption-map-table           ; table with assumption ids and its instance
    assumption-index               ; counter to give each assumption a index
    constraint-queue               ; queue used for propogation
    system-stack                   ; used to store intermediate results during computation
    system-environments            ; ATMS
    solutions                      ; list of environments and its solution space
    perfect-environment-match      ; true, if current environment was found in ATMS
    operating-environment)         ; instance of the operating environment for the system
    ()
    :inittable-instance-variables
    :gettable-instance-variables
    :settable-instance-variables)
```

Figure 9: Constraint System Definition.

the value will have its interval defined. The purpose of *type* is to identify the qualitative space during computation involving the value.

low-value and *high-value*: These are the lowest and highest values of the interval being specified by the value object. The value of these instance variables could either be an atom, if either end of the interval is a point, or a list, if either end of the interval is not a point, but greater or lesser than a point. Note, if low-value is a list then it has to be greater than a point, (i.e., (> point-value)) and if high-value is a list, it has to be less than a point, (i.e., (< point-value)).

time-stamp: When a new interval value is assigned to a value object, this instance variable records the time of assignment. This helps explain computations. We do not record the real time but simulate it with a counter which is incremented everytime the counter is referenced. This insures that every time-stamp has a unique value.

1.3. Variable

status: The variable object can be catagorized either as a constant or a state variable[7]. If it is a state variable, the status is one of *unknown, assumed, computed or initialized*; otherwise, the status is *constant*. In case of a state variable the *status* gives a very brief justification of why the variable has a certain value. If the value is due to a computational result, the status is *computed*. When an environment is set, the status of each variable is

[7] By state variable, we mean a variable in the constraint system

either *assumed*, if an assumption forces the variable to a certain value, or *initialized*, if the value is the whole interval range and is not due to an assumption. The *unknown* status indicates that the value of the variable has yet to be determined.
constraints: This is a list of instances of constraints in the system that involve a given variable. When the value of a variable changes during propagation, this gives the list of suspected constraints which have to be checked for satisfaction.
justification-flag: This boolean indicates when to record all the changes to the value of the variable to provide justification. The purpose of this will become clear in the next section.

1.4. Constraint

generators: This is a list of constraints determined by re-structuring the lhs and rhs of the constraint. All these constraints are conceptually the same and have a single variable in their lhs. For constraint $A + B > 5$, the generators are $A > 5 - B$ and $B > 5 - A$. These generated constraints are used in pruning variables during the constraint propagation.

1.5. Assumption

id: Every assumption has a unique fixed number as its identifier. The number indicates the position of this assumption in the bit-vector which is used to represent the set of assumptions in the system. The presence of an assumption in a set is indicated by setting the bit denoted by its id. A set of assumptions (an environment) is represented as an integer number corresponding to the value of the bit-vector.

1.6. Environment

environment: This is an integer corresponding to the bit-vector representing the set of assumptions in the environment.
var-val-justfn: This is a list of variable, value and justification instances for all variables in the system. As an environment is a basic building block of our ATMS, this instance variable keeps the history of the computation to avoid re-computation.
partial-environments: This is a list of environment instances whose assumption sets are subsets of the assumption set of this environment object. This defines the structure of the data-base for the ATMS. The environment instances in the list are ordered by increasing size of the assumption set of each environment. This makes the search for an environment more efficient.

1.7. Justification

value: This is a value instance corresponding to a final or an intermediate value of a variable, the justification is intended.
constraint: This is an instance of the constraint that pruned the value of a variable to the new value defined by the above instance variable, *value*, during constraint propagation.
var-value: This has the list of all variables and their value instances in the constraint at

the time of justification.
assumptions: This is a set of assumption instances. It indicates the assumptions which did contributed to the value of the justification.
reason: If the value in the justification is not due to computation, then this answers the question, *why ?*, through a canned english phrase. For example, if the value of the variables in the system are assigned to the operating region values, the *reason* would be operating-region.

1.8. Constraint System

constraint-queue: This contains a list of constraint instances. Before starting constraint propagation all relevant constraints are pushed onto the constraint queue, and propagation continues until this queue becomes empty or a constraint fails.
system-environments: This is the data-base of the ATMS in the constraint system. All the environments are arranged in a partially ordered tree to make search efficient. This instance variable acts as the root of the tree, and has only the top level nodes (which are environment instances). These environments are sorted by the cardinality of the set of assumptions of each environment.
solutions: This represents the solution space of environments for which a solution space has been computed. It is an associated list of environments and their solution spaces. The solution space is defined by a list of environment instances.
perfect-environment-match: This boolean indicates whether the current environment was in the ATMS. The ATMS decides to add the current environment to the data-base if the instance variable is false.

2. Operation Implementation

In describing the implementation details of the system, a bottom-up approach is considered. We initially define the basic operations and show how complex operations are built on top of them. We start from *value*, illustrating the methods defined on it, and go on to constraints to illustrate their satisfaction and the pruning of variables. The operations on a variable are skipped, because most of them deal with a value.

2.1. Value Object

The value object has three main types of methods: arithmetic operations, ordinal relation operations and pruning operations.

The arithmetic operations involve the *:plus* and *:minus* methods. When a value object recieves a *:plus* or *:minus* message with a list of values as its argument, the arithmetic operation defined by the message is performed on argument. Figure 10 shows the definition of *:plus*. In the resulting value instance of the *:plus* method, the *:high-val* is the sum of all the high values of the argument values and the value that received the *:plus* message.

```
(defmethod (value :plus) (y)
    (cond ((flavor-typep y 'value) (setq y (list y))))
    (let ((low low-val) (high high-val))
        (for elem in y do
            (setq low (send type :plus low (send elem :low-val) 'low))
            (setq high (send type :plus high (send elem :high-val) 'high)))
        (send type :add-new-value low)
        (send type :add-new-value high)
        (make-value type low high)))

(defmethod (value :less) (v2)
    (cond ((send type :less high-val (send v2 :low-val)) t)
          ((send self :overlap? v2) 'ok)))

(defmethod (value :prune-less) (value2)
    (let ((high2 (send value2 :high-val)) (result (send self :less v2))
        (cond ((send self :point?) result)
              ((and (send type :equal high-val high2) (send type :land-mark high-val))
               (setq high-val (send type :next-lesser-of high2)) self)
              ((send type :greatereq high-val high2) (setq high-val (send type :next-lesser-of high2)) self)
              (t result))))
```

Figure 10: Value methods: :plus, :less and :prune-less.

Similarly the *:low-val* is the sum of all the low values.

The ordinal relation operations are *:less, :greater, :equal, :lesseq* and *:greatereq* methods. These methods receive a *value* as an argument. The message-receiving value object does the appropriate comparison with its argument value. These methods are three-valued: *true, nil or ok* rather than the usual two-valued: *true or false* semantics. This is because the values being considered are intervals. These three-valued methods return: *true*, if for all points and sub-intervals in the corresponding value objects the relationship holds; *nil*, if for no point or sub-interval in the corresponding value objects does the relationship hold; and *ok*, if for some points or sub-intervals in the corresponding value objects the relationship holds, and for some other points or sub-intervals, the relationship does not hold. Figure 10 shows the definition of the *:less* method.

The pruning methods are *:prune-less, :prune-lesseq, :prune-greater, :prune-greatereq* and *:prune-equal* methods. These methods get a *value* as an argument. The value object receiving this message may prune its value depending on the argument value and the message. If the value is not pruned, then these methods act as the corresponding ordinal relations method. Otherwise, it returns a new value instance reflecting a pruned value. Figure 10 shows the definition of *:prune-less*. Let value X receive the *:prune-less* message with argument Y. Also, let X_l, X_h, Y_l and Y_h be the low and high values of X and Y respectively. If X is a point, we know it can not be further pruned, hence simply apply the *:less* method; otherwise, if X_h is greater than or equal to Y_h, then make X_h strictly less than Y_h by setting X_h to the next lower value of Y_h. This is done by the *:next-lesser-of* method defined on the qualitative space. An exceptional case is if X_h and Y_h are equal, and both are not points, then X is not pruned; because the next lesser value of ($<$ a-point)

is itself.

2.2. Constraint Object

The methods associated with constraints are used to evaluate the constraint and, if possible, prune the variables in the constraint. Figure 11 shows the methods involved in the evaluation of a constraint and the pruning of variables. The *:eval* method evaluates the lhs and rhs of a constraint using the *:eval-expr* method. The *:eval-expr* method evaluates any prefix notational arithmetic expression involving plus and minus operators and variables and return a *value*. Finally the lhs and rhs values are checked for the ordinal relationship to hold. As described before, *:eval* is also a three-valued method.

If evaluation of a constraint results in *ok*, then an attempt is made to prune all the variables in the constraint through the *:propagate-var-changes* method. We evaluate the rhs of each generated constraint in the *generators* of the constraint. The lhs value, which corresponds to a variable, is compared with the rhs value depending upon the comparator of the generated constraint. When a value gets pruned, a justification for the new value is built.

2.3. Constraint Propagation

Let us consider the constraint-propagation example in Figure 4. Figure 12 shows the code for constraint propagation. It starts by pushing all the constraints in the system, that is, constraints C-ONE and C-TWO, onto a queue. First constraint C-ONE is popped from the queue and analyzed by *:analyze* method in the constraint. This returns a list of variables X and Y, which implies that variables X and Y have been pruned. Variable X gets pruned from (-20 20) to ((> -15) 20) and variable Y gets pruned from (-20 20) to ((> -15) 20)[8] This causes all the constraints which contain at least one of the pruned variable onto the queue. This is performed by the *:push-var-constraints* method. To insure that the constraint that pruned the variables does not get pushed back onto the queue, the constraint is passed as one of the arguments to the *:push-var-constraints* method. This method does not push a constraint onto the queue if it is already in the queue. Therefore, in our example, neither of the constraints, C-ONE or C-TWO is pushed onto the queue. As explained above, the *:analyze* method may also result in either *contradiction*, stopping the propagation as it has failed; or *consistent*, popping the next constraint from the queue and analyzing it. The propagation continues by popping the next constraint in the queue, C-TWO, and analyzing it. The propagation stops when either a contradiction results from constraint analysis or the queue becomes empty. The former implies that the set of constraints can not be satisfied and hence no behavior exists in the current environment; the latter implies a successful constraint propagation.

The analysis of constraint C-TWO results in the pruning of variable X from ((> -15)

[8] Note that both variable X and Y are pruned to same value for same initial values. This a supporting fact that the order in which the variables are pruned within a constraint does not matter.

```
(defmethod (constraint :eval-expr) (expr)
   (cond ((flavor-typep expr 'variable) (send expr :value))
         ((and (listp expr) (null (cdr expr)) (flavor-typep (car expr) 'variable))
          (send (car expr) :value))
         (t (let ((evaled-args (for x in (cdr expr)
                                    collect (send self :eval-expr x))))
              (send (car evaled-args)
                    (make-message (send *system* :lookup-symbol (car expr)))
                    (cdr evaled-args))))))

(defmethod (constraint :eval) ()
   (send (send self :eval-expr lhs)
         (make-message (send *system* :lookup-symbol comparator))
         (send self :eval-expr rhs)))

(defmethod (constraint :propagate-var-changes) ()
   (let (value (val-list nil))
      (for x in variables do
         (setq val-list (cons (list x (send (make-instance 'value) :copy (send x :value))) val-list)))
      (for gen in generators do
         when
         (flavor-typep (send (car gen)
                             (make-message 'prune
                                (send *system* :lookup-symbol (caadr gen)))
                             (send self :eval-expr (caddadr gen)))
                       'variable)
         collect
         (progn (setq value (make-instance 'value))
                (send (car gen) :set-status 'computed)
                (send value :copy (send (car gen) :value))
                (send value :put-time-stamp)
                (send (car gen) :add-justification
                      (make-justification value self val-list (send self :assumpn-justifn-from-variables)))
                (car gen)))))

(defmethod (constraint :analyze) ()
   (let ((result (send self :eval)))
      (cond ((equal result 'ok) (setq status 'ok) (send self :propagate-var-changes))
            (result (setq status 'consistent) t)
            (t (setq status 'contradiction) 'contradiction))))
```

Figure 11: Constraint methods: :eval-expr, :eval, :propagate-var-change and :analyze.

```
(defmethod (constraint-system :propagate-change-loop) ()
   (cond ((send constraint-queue :empty?) nil)
         (t (let* ((constraint (send self :pop-constraint))
                   (result (send constraint :analyze)))
              (cond ((equal result 'contradiction) 'contradiction)
                    ((listp result)
                     (send self :push-var-constraints result)
                     (send self :propagate-change-loop))
                    (t (send self :propagate-change-loop)))))))

(defmethod (constraint-system :propagate-constraints)()
   (send self :push-constraints (send self :all-constraint-instances))
   (cond
         ((eq (send self :propagate-change-loop) 'contradiction) (setq system-status 'inconsistent))
         (t (setq system-status 'consistent))))
```

Figure 12: Constraint Propagation.

20) to ((> -12) 20) and variable Y from ((> -15) 20) to ((> -15) (< 17)). Then, C-ONE is pushed onto the queue. The analysis of constraint C-ONE results in *consistent*; that is, no pruning is necessary. The propagation terminates as the queue is empty, resulting a successful constraint propagation.

In the worst case, the complexity of the constraint propagation algorithm is O(nmp), where n is number of constraints in the system, m is number of variables in the system and p is the number of points or landmarks in the variables' qualitative space over which the variables are defined. This case is possible when each variable gets finally pruned to a point or an indivisible interval starting from the full interval range of its qualitative space; such that a variable gets pruned every time only by a point or an indivisible interval. Of course, in the best case, complexity is O(n), in the case that no pruning takes place.

2.4. Operating Region

A constraint system has only one operating region. By our definition of operating region, all behaviors of the system are within the operating region. Therefore, when interested in examining a set of behaviors in a particular environment and the environment is not found in the ATMS, then we could consider a region which is the intersection of the operating region and the given environment[9]. Let us call this region the *interested region*. The possible behaviors in the given environment are in the interested region; therefore, we could say that these two regions are equivalent. If the interested region is null, this implies that the given environment is outside the operating region and is a *nogood* environment.

In computing the operating region (See Figure 13) all the variables in the system are initialized to the whole interval value of their corresponding qualitative spaces. The first phase of pruning is done through constraint propagation. This may shrink the intervals

[9]We know that environment is a set of assumptions. With these assumptions, if we propagate the constraints, it would define a *region* for the environment.

```
(defmethod (constraint-system :set-up-operating-region)()
   (let (status var-list)
      (cond (operating-environment (send self :update-variables-from-environment operating-environment))
            (t (send self :update-variables-from-environment current-environment)
               (for x in variable-symbol-table do
                     (send (cdr x) :set-justification-flag nil))
               (send self :propagate-constraints)
               (setq var-list (send self :sort-variables-on-value-length))
               (for x in var-list
                     until (eq system-status 'inconsistent)
                     do
                     (setq status (send self :find-upper-and-lower-bound x))
                     (cond ((eq status 'no-bounds)
                           (msg N "There is no OPERATING REGION." N)
                           (msg "As variable, " (send x :symbol) " cannot take any values " N)
                          (msg " in the range " (send (send x :value) :value-range) N))
                          ((eq status 'changed) (send self :propagate-constraints))))
                     (for x in variable-symbol-table do
                           (send (cdr x) :set-justification-flag t))
                     (cond ((eq system-status 'consistent)
                           (send self :set-up-nogood-environments)
                           (send self :update-environment-from-variables current-environment)
                           (setq operating-environment (make-environment))
                           (send operating-environment :copy current-environment)))))))

(defmethod (constraint-system :find-upper-and-lower-bound)(var)
   (let* ((value (send var :value))
          (value-set (send value :get-value-set))
          (initial-limits (list (first value-set) (car (last value-set))))
          min-value final-limits (max-value nil))
     (setq min-value (send self :compute-limit var value-set))
     (setq value-set (reverse (cdr (member min-value value-set :test 'equal))))
          (if (and value-set (eq system-status 'consistent))
           (setq max-value (send self :compute-limit var value-set)))
          (setq final-limits (if max-value (list min-value max-value) (list min-value min-value)))
          (cond ((eq system-status 'consistent)
                  (cond ((equal initial-limits final-limits) nil)
                 (t (send (send var :value) :new-value-set final-limits) 'changed))
                 (send var :set-justification
                     (make-justification (send (make-value) :copy (send var :value)) nil nil nil 'operating-region)))
                (t 'no-bounds))))

(defmethod (constraint-system :compute-limit) (var value-set)
   (let (limit-value)
      (setq system-status 'unknown)
      (for x in value-set
          until (eq system-status 'consistent)
          do
          (setq limit-value x)
          (send self :store-all-variables)
          (send (send var :value) :new-value-set x)
          (send self :propagate-constraints)
          (send self :restore-all-variables))
      limit-value))
```

Figure 13: Setup Operating Region.

of certain variables. For efficiency, all the variables are sorted by ascending order on the lengths of their interval values using the *:sort-variables-on-value-length* method. By "length of an interval", we mean the number of points or the landmarks in the interval value. For each variable in the sorted variable list, the interval value of the variable is reduced to a minimum interval such that any solution that can exist is within the interval, based on knowledge determined by the constraint propagation. In this attempt, either the system would come up with a region, which is the operating region; or there may be a null region, which implies that there is no operating region for the system.

The method *:find-upper-and-lower-bound* returns: *no-bounds*, if the interval value of the variable becomes null in the process of finding the minimum interval value; *changed*, if a new minimum interval is found; finally, *nil*, if the existing interval itself is the minimum interval value. Initially turned off, the justification-flag for any variable is turned on at the end of operating region computation. This is done because we are just interested in the operating region; having the justification-flag on would not explain how the operating region was computed.

Let us consider the example in Figure 4 to compute the operating region. Constraint propagation shrinks the intervals of X to ((> -12) 20) and Y to ((> -15) (< 17)). Sorting the variables X and Y on their value length by method *:sort-variable-on-value-length*, returns ordered list of variables, (Y X). First, an attempt is made to prune the variable Y by *:find-upper-and-lower-bound*. The various points and indivisible intervals that the variable Y can take are ((> -15) (< -10)), (-10 -10), ((> -10) (< -5)), (-5 -5), ((> -5) (< 0)), (0 0), ((> 0) (< 5)), (5 5), ((> 5) (< 10)), (10 10), ((> 10) (< 15)), (15 15) and ((> 15) (< 17)). This list of values is generated by the method *:get-value-set* defined on the value object. The *:compute-limit* method returns the first value from the given list of values (the above list) such that all the constraints are satisfied when variable Y assumes that single value. By using the *:compute-limit* method, it is possible to get both lower and upper bounds for the value. If the list of values is in ascending (descending) order, we get the lower (upper) bound; for the variable Y, the lower and upper bounds are ((> -15) (< -10)) and ((> 15) (< 17)), respectively; Y has not changed. When a similar attempt is made to shrink variable X, the lower and upper bounds become ((> 4) (< 5)) and ((> 15) (< 20)) respectively. Therefore, the final operating region is as shown in figure 5.

2.5. Solution Space

A solution space is defined for a set of constraints and an initial environment, under the assumption that the environment is either the operating region or a sub-region of the operating region. Initially all the variables are assigned to their value in the environment. Then, all variables are collected whose values are neither points nor indivisible intervals. If there are no such variables, then the given environment itself is the solution space. Otherwise we select a variable from the collection, and for each point or indivisible interval in the value of the variable, a new environment is created: the old environment with an additional assumption that the variable has the point or an indivisible interval. Before

```
(defmethod (constraint-system :solve-for-environment)(env)
   (let ((interval-vars nil) var value old-assumptn old-env)
      (send self :update-variables-from-environment env)
      (for x in variable-symbol-table do
         (if (not (send (send (cdr x) :value) :single-value?))
            (setq interval-vars (cons (cdr x) interval-vars))))
      (cond ((null interval-vars) (send self :add-to-solutions env))
            (t (setq var (car interval-vars))
               (setq value (send var :value))
               (setq old-assumptn (send env :assumption-of-variable var self))
               (setq old-env
                     (if old-assumptn
                        (remove-from-set (send env :environment) (send old-assumptn :id))
                        (send env :environment)))
                        (send self :solve-for-each-var var value old-env env)))))

(defmethod (constraint-system :solve-for-each-var)(var value old-env env)
   (let (new-value new-env new-assumptn new-env-inst (found nil))
      (for x in (send value :get-value-set) do
         (send value :new-value-set x)
         (setq new-value (send (make-value) :copy value))
         (setq new-assumptn (send self :find-assumption var new-value))
         (setq new-env (add-to-set old-env (send new-assumptn :id)))
         (setq found (send self :present-in-system-environment new-env))
         (setq new-env-inst (make-environment))
         (cond (found (send new-env-inst :create-copy found)
                      (send self :solve-for-environment new-env-inst))
               (t (send new-env-inst :create-copy env)
                  (send new-env-inst :set-environment new-env)
                  (send new-env-inst :set-value-justfn var new-value
                        (make-justification new-value nil nil (list new-assumptn) 'set-by-solve-constraints))
                  (send self :update-variables-from-environment new-env-inst)
                  (send self :propagate-constraints)
                  (send self :update-environment-from-variables new-env-inst)
                  (cond ((equal system-status 'consistent)
                  (send self :add-to-system-environments new-env-inst)
                  (send self :solve-for-environment new-env-inst))
                  (t (send self :add-to-nogood-environments new-env-inst))))))))
```

Figure 14: Solve Environment.

adding the new assumption to the set of assumptions in the environment, verify that there does not exist an assumption in the set which involves the same variable as the new assumption. This is achieved by removing the old assumption from the set of assumptions. Every new environment is first searched in the ATMS (discussed in the next section). If there is no perfect match, then propagate constraints for the new environment. For each such new environment, a solution space is recursively computed. Figure 14 shows code for computing the solution space. The set of all solutions found in this way constitutes the solution space associated with the initial environment.

3. ATMS

The ATMS is an integral part of our constraint system. In general, the problem solver and the ATMS are two seperate modules. The problem solver draws inferences from the constraints, and the ATMS does its part by examining and saving all the new inferences. In our system the problem solver and ATMS are together.

The basic element of the ATMS data-base is an environment instance. All the values and justifications of the variables in an environment, holds valid for the set of assumptions composing the environment. The instance variable *partial-environments* of the environment object is a list of environment instances whose assumption sets are subsets of the set of assumptions of the environment which owns the partial-environments. Since each environment in partial-environments could also have a list of environments in its partial-environments, an environment represents a tree. An environment whose set of assumptions is a subset of another environment's assumption-set need not be in the partial-environments of the environment or the partial-environments decendents. This partially orders the tree and reduces the cost of inserting a new environment. The list of environments in the system-environments instance variable of the constraint system represents the sub-trees of all the environments in the system.

The operations performed on this partially-ordered tree structure are insertion and removal of or searching for an environment. Insertion adds the environment to the system-environments list, it is of $O(n)$, where n is the length of *system-environments*, the top-level environment list. Similarly, the removal is of $O(n)$, because it removes only environments which are in the *system-environments* list.

Before explaining the search for an environment, we will present an important property of the continuous and convex spaces, which is the domain. If there is an environment whose assumption-set is a subset of another environment's assumption-set, then the operating region and the solution space of the former environment are supersets of the operating region and solution space of the later environment, respectively. In other words, adding assumptions to an environment (a set of assumptions) may shrink its operating region and solution space. From the above property, if there are three assumption sets (A B), (B C) and (A B C) corresponding to environments, E_1, E_2 and E_3, then the operating region and solution space of E_3 are subsets of the intersection of the operating regions and solution spaces of E_1 and E_2.

```
(defmethod (environment :search-environment) (env-list & optional (env environment))
   (let ((perfect-match nil) (top-partial-match nil) (partial-match nil) (result nil))
      (for x in env-list
           until perfect-match
           do
           (cond ((equal (send x :environment) env) (setq perfect-match x))
                 ((sub-set (send x :environment) env)
                  (setq top-partial-match (cons x top-partial-match)))
                 ((and (set-intersect (send x :environment) env)
                       (send x :partial-environments))
                  (setq result (send self :search-environment
                                     (send x :partial-environments)
                                     (set-intersect (send x :environment) env)))
                  (setq partial-match
                        (cond ((car result) (cons (car result) partial-match))
                              (t (append (append (cadr result) (caddr result))
                                         partial-match)))))))
           (setq partial-match (send self :remove-sub-sets top-partial-match (purge partial-match)))
           (list perfect-match top-partial-match partial-match)))

(defmethod (environment :set-up-variables) (cs)
   (let* ((result (send self :search-environment (send cs :system-environments)))
          (perfect-match (car result))
          (partial-matches (append (cadr result) (caddr result))))
      (send cs :set-perfect-environment-match nil)
      (cond (perfect-match
             (send self :copy perfect-match)
             (send cs :set-perfect-environment-match t))
            (partial-matches
             (setq partial-environments partial-matches)
             (send self :get-var-val-justfn cs)
             (send self :set-var-val-justfn (send (car partial-matches) :var-val-justfn))
             (send self :intersect-regions (cdr partial-matches)))
            ((send cs :operating-environment)
             (send self :set-var-val-justfn
                   (send (send cs :operating-environment) :var-val-justfn)))
            (t
             (send self :get-var-val-justfn cs)
             (for x in var-val-justfn do
                  (send (cadr x) :set-full-range-value)
                  (send self :add-justification (car x)
                        (make-justification (send (make-value) :copy (cadr x)) nil nil nil 'whole-value-range)))))))

(defmethod (environment :intersect-regions)(env-list)
   (for x in env-list do
        (for y in (send x :var-val-justfn)
             z in (send self :var-val-justfn) do
             (send (cadr z) :intersection (cadr y))
             (send (cadr z) :put-time-stamp)
             (send self :add-justification (car z) (caddr y))))
   (for x in var-val-justfn do
        (send self :add-justification (car x)
              (make-justification (send (make-value) :copy (cadr x)) nil nil nil 'intersection-of-values))))
```

Figure 15: Environment Methods: :Search-Environment and :Set-All-Variables.

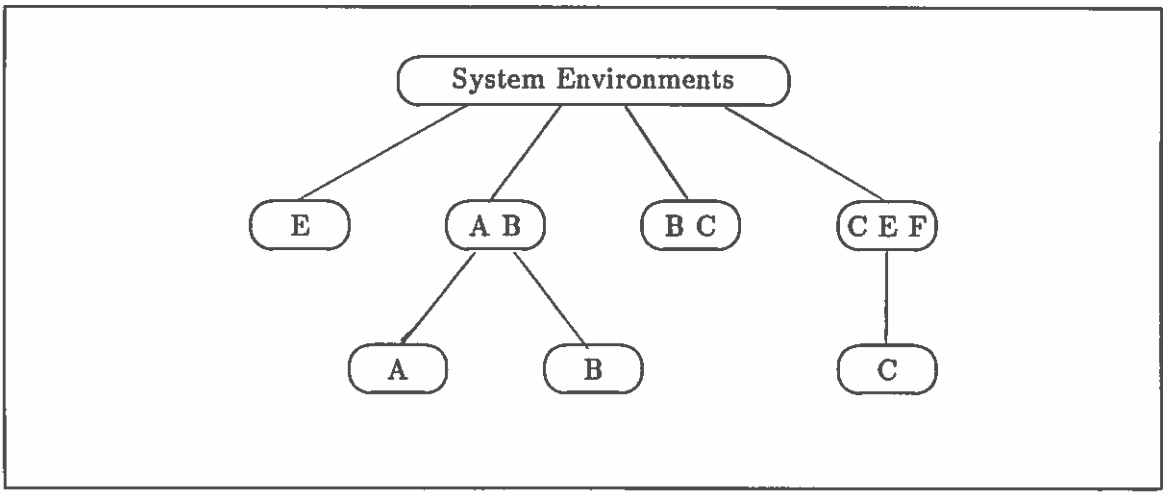

Figure 16: System Environments.

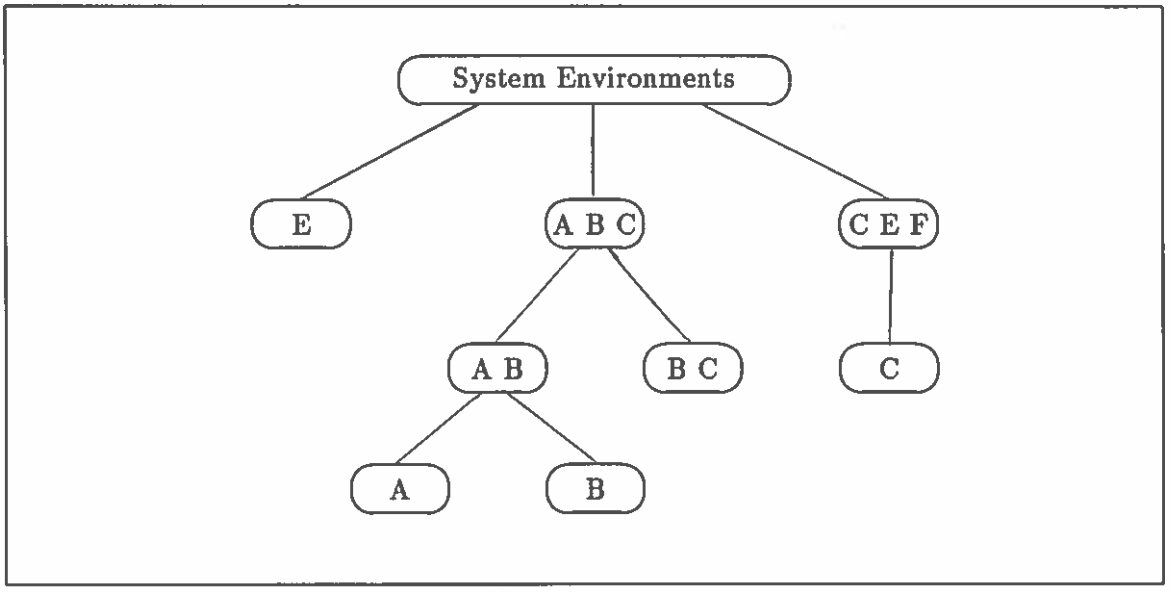

Figure 17: System Environments: After Addition of Environment (A B C).

Consider the situation in which environments E_1 and E_2 have operating regions. Now, if we are interested in the environment E_3, instead of computing the operating region from scratch, (i.e., all the variables initialized to the whole qualitative space value) we could start from the region that is the intersection of environments E_1 and E_2. In Figure 15, the *:search-environment* and *:set-up-variables* methods perform variable initialization in a new environment.

The search operation is more complex because it involves two kinds of search. First, we look for an exact environment in the partial-ordered tree. If this fails, we look for all possible subset environments of the environment being searched. The variable *top-partial-match* of the *:search-environment* method in Figure 15 has a list of environments that are subsets of the environment being searched. This list is also a subset of environment instances listed at the top-level, *system-environment*. The variable *partial-match* has a list of environments which are subsets of the environment being searched, but the environment instances are below the top-level environments. For the example in Figure 16, if we search for environment (A B C), then the *top-partial-match* would be list of (A B) and (B C); and the *partial-match* would be (C). Of course there is no perfect match, so *perfect-match* is nil.

To maintain the partial ordering in the tree, after identifying the environments which are subsets of the environment being searched (in the above example E_1 and E_2 are subsets of E_3) the subset environments are removed from the top-level of the tree. All the removed subset environments are put in the *partial-environments* of the new environment and the new environment is added to the top-level of the tree in the proper position to maintain the ordering on the size of the assumption set. Figure 17 shows the tree-structure after adding the new environment to the environments in Figure 16.

CHAPTER IV

MODELING

Every continuous system has a potentially infinite set of possible actual behaviors. By defining qualitative behavior (refer to section 2.4), it is possible to capture all the actual behaviors in a finite set of qualitative behaviors. Each behavior can be defined by a set of characteristic values for all variables of the system. These behaviors are governed by a set of axioms and laws, represented as constraints in our system. According to [4], a robust qualitative simulation is achieved by ensuring that the system's simulation model does not presuppose the very mechanism it is trying to describe. This is the well-known *no-function-in-structure* principle [4].

In this chapter we consider two examples to show how the qualitative constraint system can be used to model and generate the behaviors of the system. The first example is a simple system: pipe valve with a single constraint. The second example is a cardiovascular system. This is relatively a complex system to highlight features of the constraint system.

1. Example: Pipe Valve

A valve controls the flow of fluid in a conduit. The three main characteristic variables of a valve are the pressure of the incoming fluid, the cross sectional area of the valve and the quantity of the fluid coming out on the other side of the valve. We are interested in studying the behavior of the valve in qualitative terms, i.e., not in the real values of pressure P, area A and quantity Q; but how the changes in these values affect each other. A valve model has three states: open, working and closed. For each state, a different set of constraints governs the behavior of the system. In terms of the variables P, Q and A : open state implies $P = 0$; working state implies P, Q and $A > 0$; closed state implies $Q = 0$. In this example we only model the most interesting state, the working state. In modeling the valve, the variables in the constraint system would be DP, DA and DQ representing the changes in pressure, area and quantity, respectively. These variables are defined over $-0+$space qualitative space. The relationship between these variables is a qualitative differential equation which defines the constraint: $DQ = DA + DP$. The qualitative differential equations are called *confluences* in [5]. The confluence represents multiple competing tendencies: an increase in area positively influences flow of the quantity and negatively influences pressure, the decrease in pressure negatively influences the flow of quantity, etc. As there may be not just one, but a set of constraints describing a component, variables may appear in many constraints and thus can be influenced in

different ways.

For this simple example with only one constraint, we derive all the possible behaviors by computing the solution space. Figure 18 shows the solution space for the valve given its operating region. Consider solutions in which DQ = 0, this implies that there is no change in quantity of fluid coming out of the valve. The corresponding three possible behaviors are: there has been no change in both pressure and area (DP = DQ = 0), the system continues to be in the same previous state; the pressure has increased (DP = +) and it is compensated by a decrease in the area (DA = −), maintaining the same flow of quantity; the pressure has decreased (DP = −) and it is compensated by an increase in the area (DA = +), maintaining the same flow of quantity. In a similar fashion all the behaviors in the solution space could be described.

The constraint system also allows us to study the relationship between any two variables with all other variables fixed to certain values. For the valve example, the influences of DP and DQ on each other is as shown in Figure 19 for DA = 0. A *nil* entry in the matrix indicates that such a behavior is not possible and a *T* indicates a behavior. When dealing with huge system (say, one with 50 variables), we generally focus on a certain set of variables to study the influence of these variables among each other. Thus the relationship matrix gives an explicit relationship between any two variables for the current environment.

2. Example: Cardiovascular System

The cardiovascular system consists of the heart - which is divided into left and right pumps, lungs, and body. The left heart pumps oxygen-rich blood through the systemic arteries to the body's capillaries. The capillaries are a major source of resistance to the blood flow; the blood that exited the left heart at a pressure of approximately 100mm Hg (millimeters of mercury) returns to the right heart via the systemic veins at a pressure close to 5mm Hg. This carbon-dioxide-laden blood gets pumped by the right heart to the lungs through the pulmonary arteries. In the lungs, blood gets re-oxygenated and returns to the left heart through the pulmonary veins. Thus completing a cycle.

In modeling the cardiovascular system, observable parameters are divided into *properties* and *variables*. The properties represent the relatively static values of a real circulatory system such as vascular resistance to blood flow or heart strength. They are assumed to be constant over the interval of interest. Variables, such as cardiac blood flow or atrial pressure, change value either in direct response to property changes or indirectly through other variable changes.

Using the standard mapping of pressure to voltage and flow to current, Figure 20 abstracts the cardiovascular system into an electrical circuit. The arteries and capillaries act as resistors to the flow of blood. The highly compliant veins stretches to accomodate more blood without drastically raising their pressure, functioning like a electrical capacitor. The left and right hearts are modeled as having an internal voltage source that is sensitive to the pressure of the returning blood. The actual pumping of heart is modeled

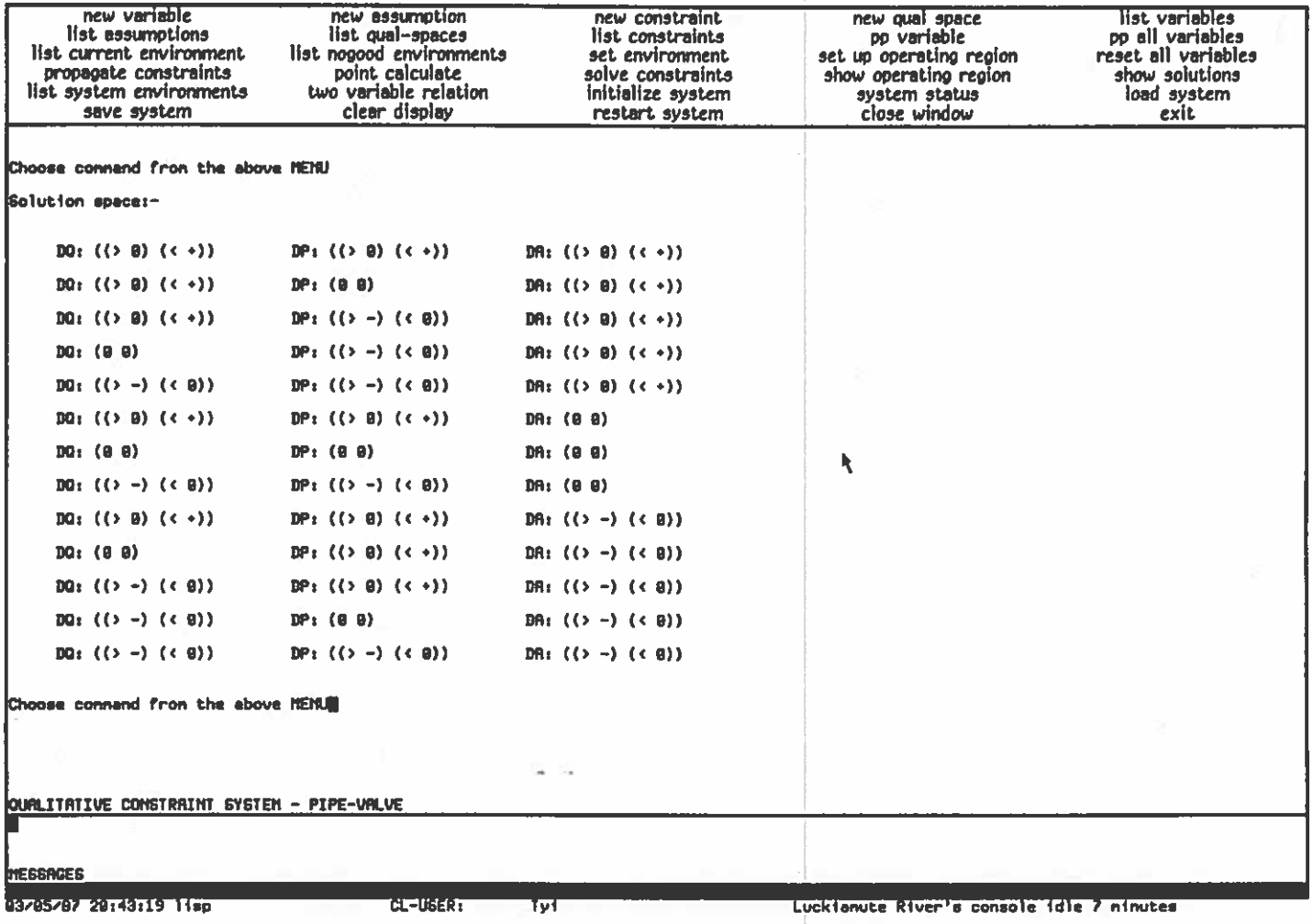

Figure 18. Valve: Solution Space.

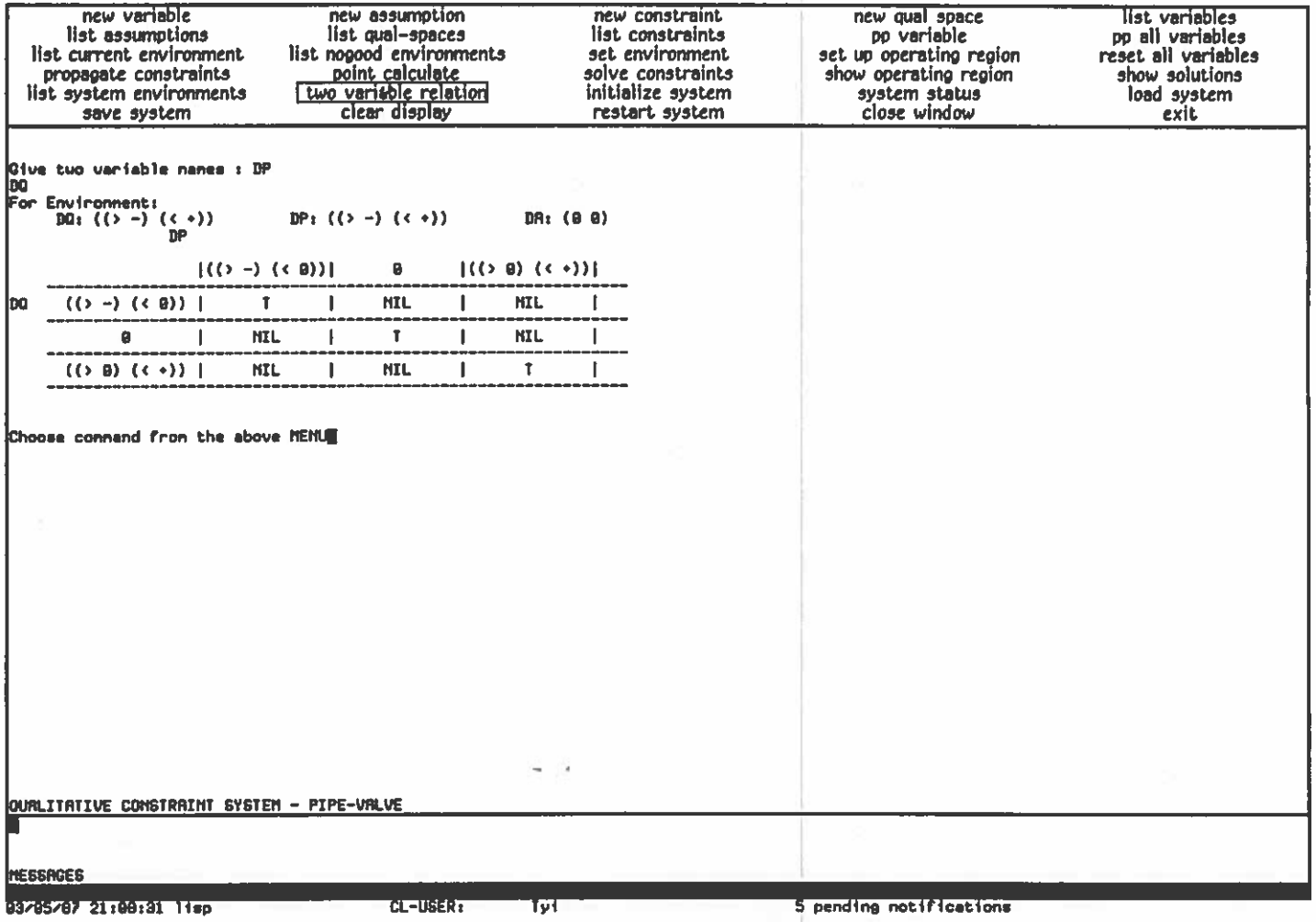

Figure 19. Relationship between DQ and DP.

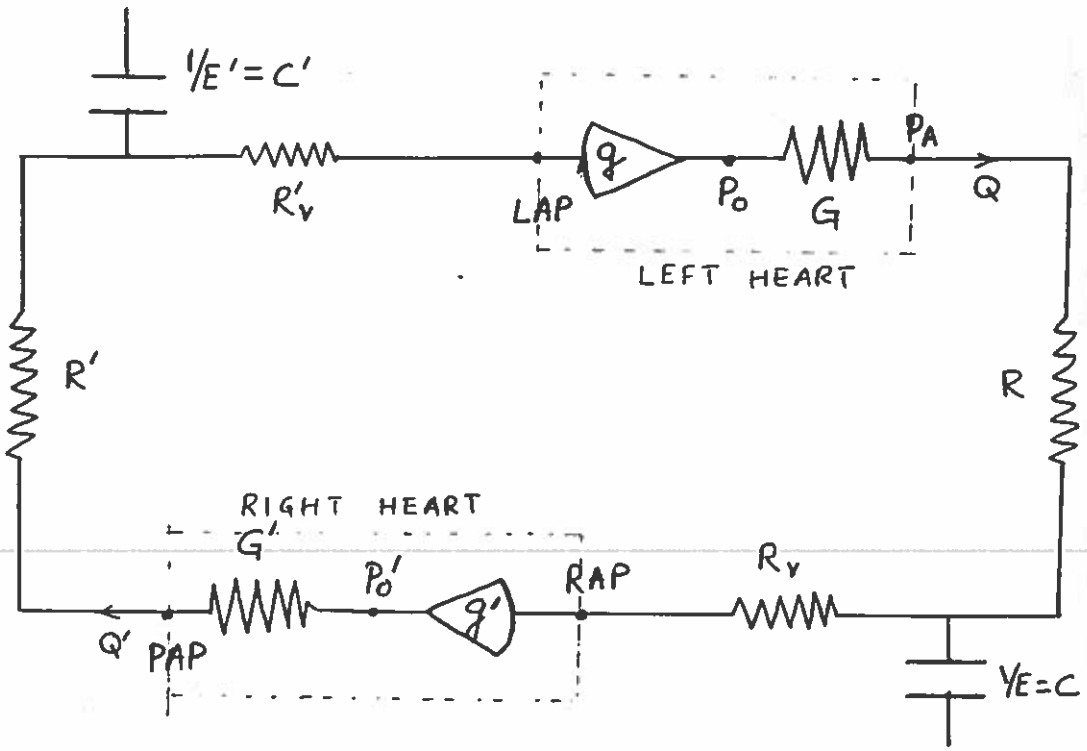

Figure 20: Electrical Circuit Equivalent to Cardiovascular System.

as conductance that converts this internal potential to an out-flow. Following are the definitions of various properties and variables in Figure 20.

Left heart and systemic circulation properties:

R	Systemic arterial resistance
R_v	Systemic venous resistance
E	Systemic vascular stiffness
G	Internal conductance of left heart
g	Factor of maximal pump potential of the left heart

Right heart and pulmonary circulation properties:

R'	Pulmonary arterial resistance
R_v'	Pulmonary venous resistance
E'	Pulmonary vascular stiffness
G'	Internal conductance of right heart

$g\prime$ Factor of maximal pump potential of the right heart

Left heart and systemic circulation variables:

P_A Mean systemic artery pressure

Q Left cardiac output

LAP Left atrial pressure

P_o Left maximum potential pressure

V Volume distending veins

Right heart and pulmonary circulation variables:

PAP Mean pulmonary artery pressure

$Q\prime$ Right cardiac output

RAP Right atrial pressure

$P_o\prime$ Right maximum potential pressure

$V\prime$ Volume distending veins

We are trying to model the normal functioning of the circulatory system, therefore all properties of the system are assumed to be normal, i.e., there is no change in the values of the properties. The model describes the average behavior of the system and not its pulsation. With this assumption, the qualitative differential equations defining the cardiovascular system are shown in Figure 21. According to [2], Figure 22 illustrates the behavior of the system. Note that the variable names in Figures 21 and 22 are slightly different from that in the above list of variables, as the subscripts are raised a level higher, the "$\prime$" is expanded to "-dash" and the prefix "D" is added to all the variables to indicate *change* in the value.

Before arriving at the solution space shown in Figure 22, initially we started with only the first eight constraints (See Figure 21). This resulted in 73 solutions and it implied that the system needed to be further constrained. By adding a constraint for conservation of mass (blood) this solution space was reduced to 13 solutions. Again 13 was reduced to 7 solutions by adding the constraint that direction of change in the left cardiac output (DQ) is the same as right cardiac output (DQ-DASH), another requirement of equilibrium operation. This indicates that one could develop a system in a stepwise manner, understanding the system better at every step.

The remaining degrees of freedom within the model are physiologically consistent; that is if the variable P_0 and$P_0\prime$ were changing, then either the flows would rise or the arterial pressure would rise and the other would fall. Secondarily, volumes would reflect a shift of blood into or out of the systemic veins. Thus the equilibrium model can be used to study the behavior of the heart.

From Figure 22, the most trivial behavior of the system, indicated by solution 4, is no changes in the variable values. This states that the system continues to be in the same state of behavior. The remaining behaviors can be grouped in to two classes - before and after the heart pump. The solutions 1, 3 and 6 belong to former class and the remaining solutions, 2, 5 and 7 belong to later class. In each class, all the variables

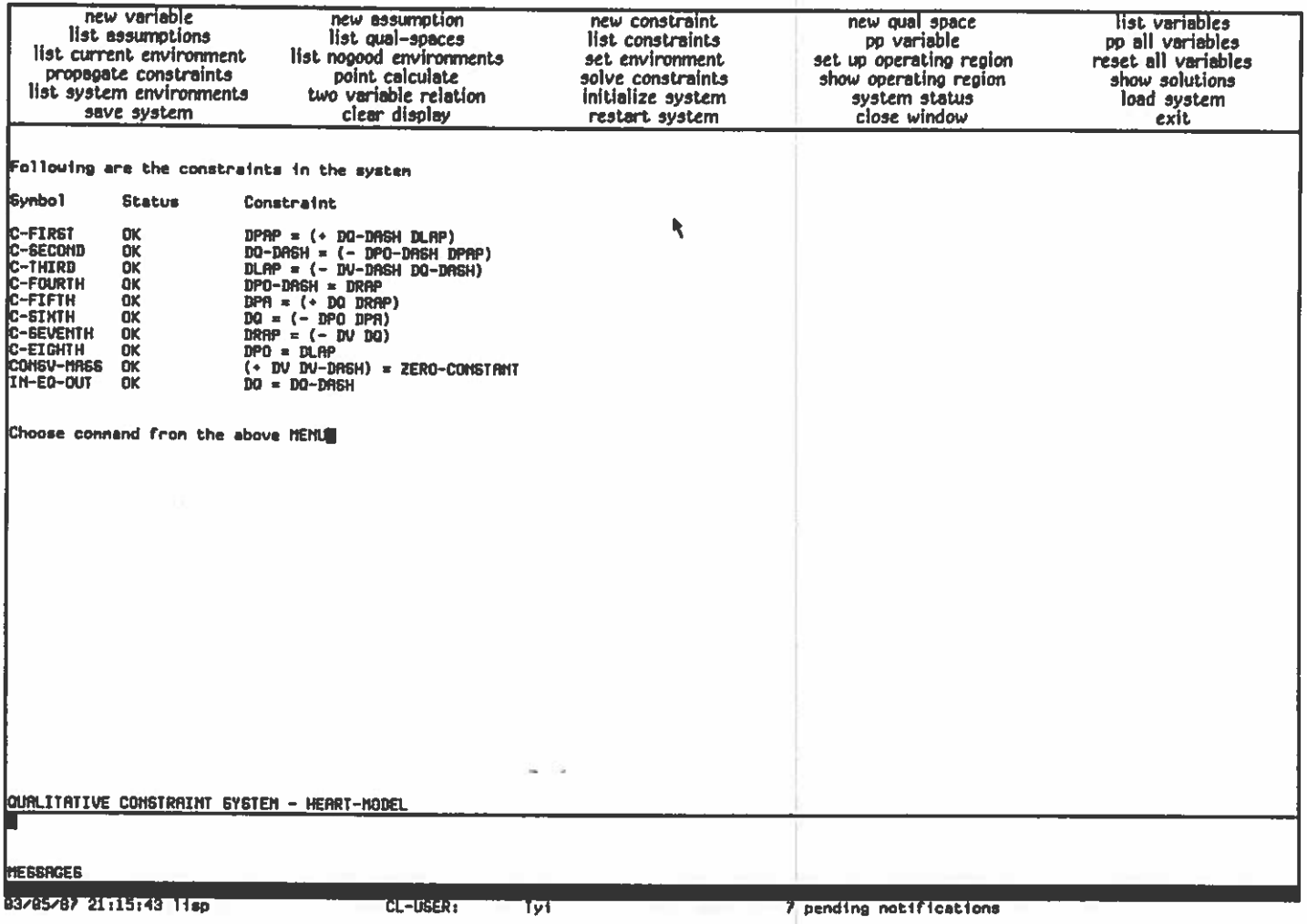

Figure 21. Cardiovascular System: Constraints.

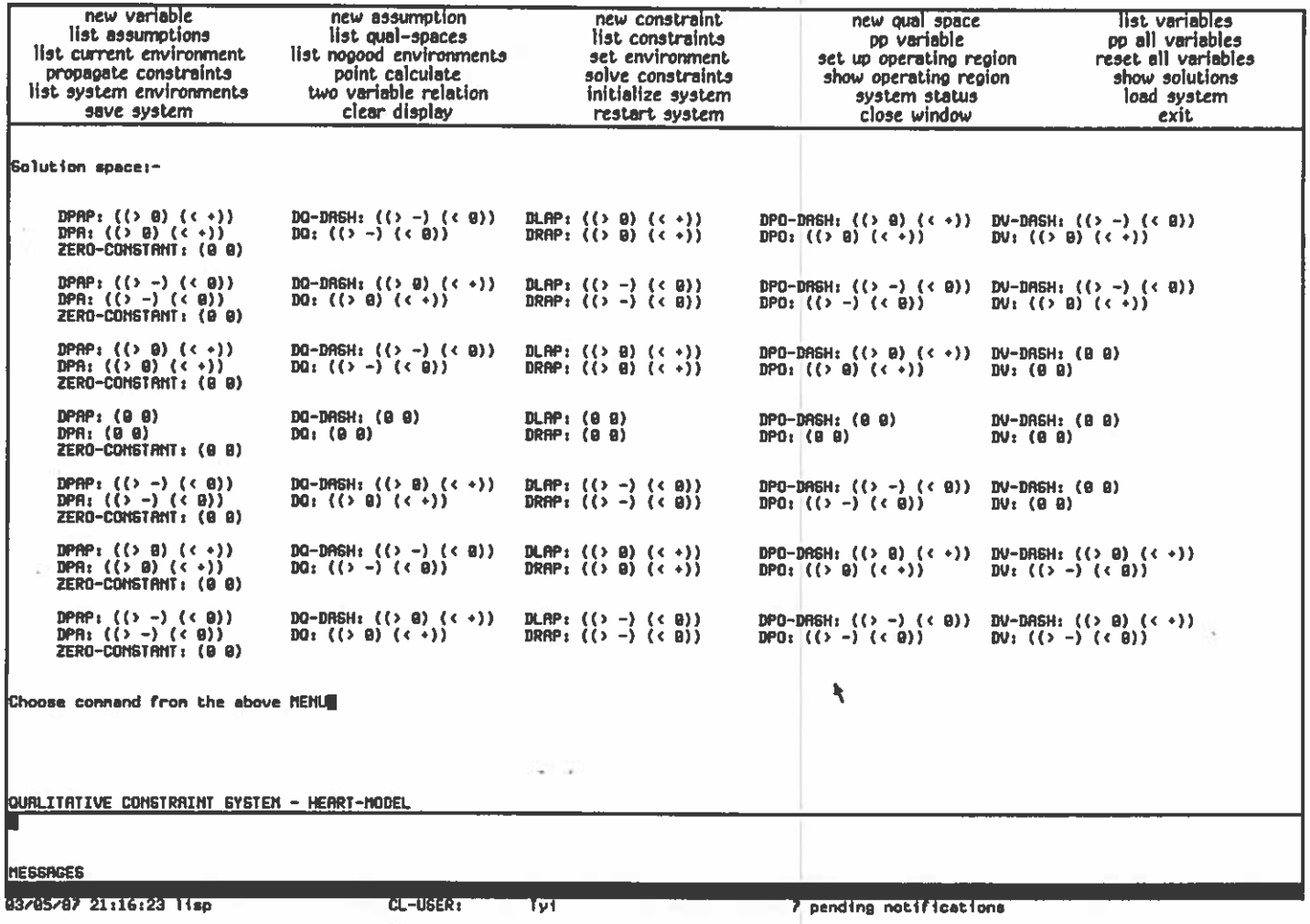

Figure 22. Cardiovascular System: Solution Space.

have same values except DV and DV-DASH. As DV and DV-DASH are constrained by the law of conservation of blood, either there is no change in both or an increase in one is compensated by a decrease in other. The before-heart-pump class solutions explains the building up of pressure inside the heart before pumping the blood out (DPO and DPO-DASH are increasing) and the flow of blood out of ventricles decreasing (DQ and DQ-DASH are decreasing) because the valves of the heart are closed to help build the pressure. The pressure in the atrials should increase (DLAP and DRAP are increasing) to cause the blood to flow into the ventricles form the atrails before the heart pumps. Similarly after-heart-pump class solutions expains the decrease in the internal pressure of the heart after pumping (DPO and DPO-DASH decreasing). The pumping triggers the out-flow of blood from the heart (DQ and DQ-DASH increasing) to the systemic and pulmonary systems.

3. Summary

In modeling the above two examples, we have seen that it is possible to generate the behavior of a system through constraint propagation. Initially one may start with a few constraints due to lack of knowledge about the system. This may result as indicated in our second example in a huge solution space. By examining the solution space one may learn more about the system to add more constraints whereby reducing the solution space. Hence we see that qualitative simulation allows one to start with partial knowledge about the system.

CHAPTER V

CONCLUSION

This chapter gives a summary of the thesis, briefly describing the capabilities of the constraint system. It suggests a possible extension to the present system to make it more robust and efficient and finally the contribution of the research will be discussed.

1. Thesis Summary

The qualitative constraint satisfaction system presented here provides a complete environment with a constraint language to model a system for qualitative simulation purposes. It allows the users to define their own qualitative space over which all the values in the system are defined. In addition the users define the operations: plus and minus for the qualitative space.

The system performs computations on a set of constraints by constraint propagation to satisfy all of them. After each constraint has been satisfied, then local propagation takes place to prune the values of the variables in the constraint. The history of the computation is maintained to indicate which values were derived from which others. This information can be used to explain the computation. An assumption mechanism is provided to be able to create environments and study the behaviors of the system in it, and to record environments whose assumption set is nogood to limit the search and computation time. To avoid recomputation, there is an ATMS to record all the environments and the values of the variables in it.

2. Extension: A Hierarchical Approach

The constraint system presented in this thesis is *flat*. There is no distinction between any two variables or constraints; all are treated by the system at the same level. If we are modeling a small system (that is, one with few variables and constraints) then the present constraint system is sufficient. But if we want to model a complex system, then we find that the present system does not provide any features to help model the system structurally. As the number of constraints increases in the system, there is no mechanism provided in the current constraint system to capture the relationship among the subsets of constraints in the system. By capturing such relationships it would help in a better representation and understanding of the system. This would also lead to a better justification of the variables in the system. To achieve this robustness, we could

decompose the complex system into *components* and try to model each component as a seperate constraint system and finally provide *connectors* to integrate these components. This reflects a hierarchical approach of modeling a system.

A connector states that the two variables, each in different components are the same, that is, the values of the variables at either end of the connector must be equal. Thus each constraint can be treated as a special case of a constraint system or component such that it has only two variables, V_1 and V_2 belonging to C_1 and C_2, respectively; and a single constraint, $V_1 = V_2$. In the present system, local propagation in a constraint assures that if V_1 can be equal to V_2 (that is, the intersection of the values of V_1 and V_2 is non-null) then the values of the variables will be made equal to the intersection of their values. Otherwise, the system is inconsistent.

Each component is treated as an independent constraint system with its own set of variables and constraints. The constraint propagation can be performed on each of the components in the system and the resulting values can be communicated through the connector among the components so that the overall system is consistent. To make the overall system consistent may involve many propagation of constraints in all the components. After each propagation, the values of the variables are communicated among the components which share the variables through connectors.

The advantages of a hierarchical approach are that it provides a natural representation of a complex system; allows to represent generic components, to avoid repetitions; allows to examine a component of a system in more details ignoring other components; and is computationally more efficient than having a single constraint system to represent the whole complex system. We shall now briefly discuss each of the above advantages.

2.1. Natural Representation

The hierarchical representation is almost a one-to-one mapping of the structural representation of the system. Hence it is possible to model the relationships among the components. A change in the value of a variable in a component may in turn change the values of the variables in the same and/or other components. The justification of these changes would be closer to the causal reasoning than that of the present system because of the principle of *locality* and *causality* rule [5]. The principle of locality states that the laws[10] for a component cannot specifically refer to any other component. A component can only act on or be acted on by its immediate neighboring component. Its immediate neighbor components must be identifiable a priori in the structure. The causality rule states that a component will not change state[11] unless it is acted upon, that is, a set of values of the variables is modified externally (outside the component), which in turn triggers constraint propagation for constraint satisfaction within that component.

[10] Laws are represented as constraints in the present constraint system

[11] The state of a component is defined by the values of the variables in it.

2.2. Generic Component

When there are more than one similar type of component in a system, instead of repeatedly defining the same component again and again, one could define a single generic component. This generic component could be used to make any number of component instances. Hence we could have a library of generic components to make any system that could be made out of it.

2.3. Individual Examination

By dividing the system into components, it helps in zooming in a certain component to study its behavior in isolation. This gives a better understanding of the structure and behavior of the component. Due to the neighboring components, certain variables in a component may be forced to have a particular value which may not cause a certain behavior of the component. Thus studying a component in isolation highlights such behaviors of the component.

2.4. Efficiency

As stated in the chapter 3, the complexity of constraint propagation is O(nmp), where n is the number of constraints, m is the number of variables in the system and p is the number of points or landmarks in the qualitative space over which the variables are defined. In a hierarchical system the complexity of constraint propagation would be O(NMp), where N and M are the number of constraints and variables in a component, respectively such that $N * M$ is maximum over all the components in the system. This is a remarkable improvement especially when the variables and constraints are equally distributed. In such a case the complexity will be reduced by a factor of $1/(C * C)$, where C is the number of components in the system.

3. Contribution of this thesis

There has been a considerable research on qualitative reasoning [5,7,9,11,12]. This thesis is also an attempt in this area. Even though the constraint system presented here is able to generate the behaviors of a physical system in qualitative terms, it is not able to backup the behaviors with proper causal reasoning. This is due to the fact that constraints are non-directional and thus need a deeper model to capture the sense of direction of the causal relationships. After the generation of the behavior by the present system, using a deeper model of the system one could give a proper causal reasoning for each behavior.

APPENDIX A

QUALITATIVE SPACE DIFINITIONS

Figure 23 and 24 shows two qualitative space: Integer-space and -0+space.

```
(defflavor integer-space
    ()
    (qual-space)
    :settable-instance-variables
    :gettable-instance-variables)

(defmethod (integer-space :plus) (x y type)
    (cond ((and (listp x) (listp y))
           (cond ((eq (car x) (car y)) (list (car x) (+ (cadr x) (cadr y))))
                 (t (msg N "Warning: adding" x y "result is " (+ (cadr x) (cadr y)) N)
                    (+ (cadr x) (cadr y)))))
          ((listp x) (list (car x) (+ y (cadr x))))
          ((listp y) (list (car y) (+ x (cadr y))))
          (t (+ x y))))

(defmethod (integer-space :minus) (x y type)
    (cond ((and (listp x) (listp y))
           (cond ((not (eq (car x) (car y))) (list (car x) (- (cadr x) (cadr y))))
                 (t (msg N "Warning: subtracting " x y " result is " (- (cadr x) (cadr y)) N)
                    (- (cadr x) (cadr y)))))
          ((listp x) (list (car x) (- (cadr x) y)))
          ((listp y) (cons (if (eq (car y) '>) '<'>)(list (- x (cadr y)))))
          (t (- x y))))

(defmethod (integer-space :add-new-value)(val)
    (if (listp val) (setq val (cadr val)))
    (cond ((listp val) nil)
          ((memq val val-set) nil)
          (t (setq val-set (add-to-qual-space self val val-set)))))

(defun add-to-qual-space (self val val-set)
    (cond ((null val-set) (list val))
          ((¡ (car val-set) val) ;;;(send self :less (car val-set) val)
           (cons (car val-set) (add-to-qual-space self val (cdr val-set))))
          ((¿ (car val-set) val) ;;;(send self :greater (car val-set) val)
           (cons val val-set))))
```

Figure 23: Definition of Integer-space.

```
(defflavor -0+space
    ((-0+-plus-table '(((0 0) 0) ((0 +) +) ((0 -) -)
                       ((+ 0) +) ((+ +) +) ((+ -) *)
                       ((- 0) -) ((- -) -)
                       ((- +) *)))
; Add table is an assoc list where the car of each pair is a list
; representing the qual. values of the two addends and the associated
; list represents the range on the sum. For instance, the assoc pair
; ((+ -) (- +)) means that the addition of a positive and a negative
; integer yields something within the range (- +) (i.e. the whole space).

    (-0+-minus-table '(((0 0) 0) ((0 +) -)
                       ((0 -) +) ((+ 0) +)
                       ((+ +) *) ((+ -) +)
                       ((- 0) -) ((- -) *)
                       ((- +) -))))
    (qual-space)
    :settable-instance-variables
    :gettable-instance-variables)

(defmethod (-0+space :plus) (x y type)
    (cond ((and (listp x) (listp y))
           (cond ((eq (car x) (car y)) (list (car x) (send self :look-plus-table (cadr x) (cadr y) type)))
                 (t (msg N "Warning: adding " x y " result is "
                         (send self :look-plus-table (cadr x) (cadr y) type) N)
                    (send self :look-plus-table (cadr x) (cadr y) type))))
          ((listp x) (list (car x) (send self :look-plus-table y (cadr x) type)))
          ((listp y) (list (car y) (send self :look-plus-table x (cadr y) type)))
          (t (send self :look-plus-table x y type))))

(defmethod (-0+space :minus) (x y type)
    (cond ((and (listp x) (listp y))
           (cond ((not (eq (car x) (car y))) (list (car x) (send self :look-minus-table (cadr x) (cadr y) type)))
                 (t (msg N "Warning: subtracting " x y " result is "
                         (send self :look-minus-table (cadr x) (cadr y) type ) N)
                    (send self :look-minus-table (cadr x) (cadr y) type))))
          ((listp x) (list (car x) (send self :look-minus-table (cadr x) y type)))
          ((listp y) (cons (if (eq (car y) '>) '<'>)(list (send self :look-minus-table x (cadr y) type))))
          (t (send self :look-minus-table x y type))))

(defmethod (-0+space :look-plus-table) (x y type)
    (let (val)
        (setq val (cadr (assoc (list x y) -0+-plus-table :test 'equal)))
        (cond ((not (eq val '*)) val)
              ((eq type 'low) '-)
              (t '+))))

(defmethod (-0+space :look-minus-table) (x y type)
    (let (val)
        (setq val (cadr (assoc (list x y) -0+-minus-table :test 'equal)))
        (cond ((not (eq val '*)) val)
              ((eq type 'low) '-)
              (t '+))))

(defmethod (-0+space :add-new-value)(val)
    (if (listp val) (setq val (cadr val)))
    (cond ((listp val) nil)
          ((memq val val-set) nil)
          (t (setq val-set (add-to-qual-space self val val-set)))))
```

Figure 24: Definition of $-0+$space.

BIBLIOGRAPHY

[1] A. Borning, A Constraint Oriented Simulation Laboratory, PhD dissertation, Dept. of Computer Science, Stanford University, (1979).

[2] K. Campbell, The Physical Basis of the Problem Environment in Cardiovascular Diagnosis, unpublished, (1983).

[3] J. De Kleer, An Assumption-based TMS, Artificial Intelligence, 28 (1985), 127-162.

[4] J. De Kleer, and J. S. Brown, Mental Model of Physical Mechanisms, CIS-3 Cognitive and Instructional Sciences, Xerox PARC, Palo Alto, CA, (1981).

[5] J. De Kleer, and J. S. Brown, A Qualitative Physics based-on Confluences, Artificial Intelligence 24 (1984), 7-83.

[6] J. Doyle, A Truth Maintenance System. Artificial Intelligence 12 (1979), 231-272.

[7] A. M. Farley, Diagnostic Mechanism Modeling, unpublished, University of Oregon, (1986).

[8] Reference Guide to Symbolics-Lisp, 2, Symbolics, Cambridge, MA, (1985).

[9] K. D. Forbus, Qualitative Process Theory, Artificial Intelligence 24 (1984).

[10] J. Gosling, Algebraic Constraints, PhD dessertation, Dept. of Computer Science, Carnegie-Mellon University, (1983).

[11] B. Kuipers, Commonsense Reasoning about Causality: Deriving Behaviour from Structure, Artificial Intelligence 24 (1984), 169-203.

[12] B. Kuipers, Qualitative Simulation, Artificial Intelligence 29 (1986), 289-338.

[13] A. K. Mackworth, and E. C. Freuder, The Complexity of some Polynomial Network Consistency Algorithm form Constraint Satisfaction Problem, Artificial Intelligence, 25 (1985), 65-74.

[14] M. C. Maletz, An Architecture for consideration of Multiple Faults. IEEE Application Conference (1985), 60-67.

[15] G. J. Jr. Steele, The Definition and Implementation of a Computer Programming Language based on Constraints, PhD dessertation, Dept. of Computer Science, MIT, (1980).

[16] G. J. Jr. Steele, Common LISP: The Language, Digital Press, (1984).

[17] R. M. Stallman, and G. J. Sussman, Forward Reasoning and dependency-directed backtracking in a system for computer aided circuit analysis, Artificial Intelligence 9 (1977), 135-196.

[18] G. J. Sussman, and G. J. Jr. Steele, Constraints - A Language for expressing Almost-Hierarchical descriptions, Artificial Intelligence 14 (1980), 1-39.

[19] I. E. Sutherland, SKETCHPAD: A Man-Machine Graphical communication system, PhD Dessertation, Dept. of Computer Science, MIT, (1963).