INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

UMI

University Microfilms International
A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
313/761-4700 800/521-0600
Programming using a constraint satisfaction system for robotics

Raja, Basalat Ali, M.S.

Rice University, 1992
RICE UNIVERSITY

Programming Using a Constraint Satisfaction System for Robotics

by

Basalat Ali Raja

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE Master of Science

APPROVED, THESIS COMMITTEE:

Rui J.P. de Figueiredo, Chairman
Professor of Electrical and Computer Engineering
and of Mathematical Sciences

Peter J. Varman
Associate Professor of Electrical and Computer Engineering

J. Barlett Sinclair
Associate Professor of Electrical and Computer Engineering

Houston, Texas
April, 1991
ABSTRACT

Programming robots is a difficult task, especially for autonomous robots that must operate in an open, unstructured environment. A programmer building such a system should have a number of sophisticated tools at his disposal. In this thesis, we provide the basic framework for the development of a programming language that can be used to program an autonomous robot. The programming language is based on a constraint system. We describe our implementation of the constraint system, as well as our research done into developing a language based on this system.
Table of Contents

1. Introduction ................................................................. 1
2. Background ............................................................... 6
   2.1 Pattern Matching Systems ........................................... 7
   2.2 Temporal Constraint System ...................................... 9
   2.3 A Plan Generator For Robots .................................... 11
3. Constraint Propagation .................................................. 16
   3.1 Constraint Networks ............................................... 16
   3.2 Constraint Connectors ............................................. 19
   3.3 System Description ................................................ 20
       3.3.1 Environments .................................................. 20
       3.3.2 Constraint Satisfaction Algorithms ...................... 22
       3.3.3 Constraint Propagation Mechanism ....................... 25
4. Integration With Other Systems ....................................... 29
   4.1 Motivation ................................................................ 29
   4.2 A Model by Temporal Constraints System ...................... 30
       4.2.1 Temporal Constraint Satisfaction ......................... 30
       4.2.2 Temporal Constraint Relationships ....................... 31
   4.3 Dynamic Modeling and Visualization of Down-line Effects .... 36
       4.3.3 Deductive Reasoning in Logos ............................... 40
       4.3.4 State Descriptors ............................................. 40
       4.3.5 Constraint System in a Deductive Framework ............ 43
   4.4 Encoding Expertise In Constraint Relaxation Strategies ...... 45
   4.5 Summary ................................................................... 50
5. A Constraint Language .................................................... 51
   5.1 Necessary Extensions to the Constraint System ................. 51
   5.2 Implementing Recursion ............................................. 52
   5.3 Logical Connectors ................................................... 55
   5.4 Using the Language .................................................... 56
   5.5 Trapping Constraint Violation Errors ............................ 58
6. Conclusions .................................................................... 62
7. Bibliography ................................................................... 65
Chapter 1
Introduction

Our ultimate goal is to provide a comprehensive planner to guide the Rice-rob bot I, an autonomous testbed designed and developed at Rice. The robot is expected to be able to operate in an unstructured environment and respond intelligently to input from a variety of sensors. This is a very difficult problem to handle, and modules that deal with various situations must be built piecemeal over a period of time. This thesis discusses some of the primary steps that we took in achieving our goal.

Given the complexity of the situation, we did not attempt the creation of the task-planner in a single step. Rather, we decided to design a powerful and expressive programming language that will then be used to build several of the modules of the task-planner. This represents an important step towards our overall goal. We had available a constraint system that had been partially completed at the time we made the initial thesis proposal. We decided to complete this constraint system, and to enhance the capabilities of this system in a direction suitable to design the language that we desired. It is expected that the work presented in this thesis will provide the programming language that will be used to develop several of the tools that will constitute the path-planner for the robot.

The specific contribution of this research is to provide the basic framework for the creation of a new programming language that can be used to pro-
gram an autonomous robot. Over the past years, as computers have grown larger and much more complex, the difficulties involved in programming them and using them to the maximum of their potential capacity have also grown. The field of robotics includes all of these aforementioned difficulties, as well as including problems specific to itself. Within a computer, the environment is very well-defined. Strange events do not happen. However, a robot needs to operate in an environment where unplanned events can occur. Our own experience has shown that except for the most trivial cases, even within the confines of a well-defined environment such as a laboratory, unexpected events can occur. This adds a further dimension of difficulty in programming a robot. Current programming languages have not been able to provide a framework in which an autonomous robot can be totally programmed with ease, thus it is important to conduct research into new approaches that might allow us a better grasp of this immense complexity.

Within the scope of this thesis, our primary focus is language design, and in this thesis we will focus on the development of a language similar to several other declarative languages, for example, Prolog. In the following chapters, we will describe the system that we have developed and compare and contrast it with other systems of a similar type. We will also discuss our own system in detail. The organization of this thesis is as follows:

I. Background: Chapter 2 will discuss different methods of constraint propagation and various ways in which they can be implemented. This chapter is intended to provide a background against which the system we have developed
can be evaluated. It details some of the other work done in this area, and gives pointers as to where else in this thesis other work is described.

II. Development of a Constraint System: We have an implementation of a constraint propagation system based on the Waltz algorithm [Davis87]. Chapter 3 focuses on the development of the system and discusses the issues involved in constraint propagation and in the design of such a system. The main thrust of this work was to demonstrate that such a system is indeed viable and to apply it to real-world applications. Once we have a constraint system, we can then apply it to problem-solving. We have also made several enhancements and additions to the Waltz algorithm; for example, multiple interval values, point spaces and constraint and variable priorities. This constraint system has been integrated into a powerful and expressive deductive reasoning tool. As such, we have a sophisticated error reporting mechanism. This provides for a facility for constraint relaxation, which is useful in handling constraint violations. This capability will be initially used in Chapter 4 and further developed in Chapter 5.

III. Integration with other systems: The constraint system that we have developed is not a stand-alone tool. It is intended to be part of a set of tools that can be used in various AI applications. In Chapter 4 we briefly describe some of the other tools that the constraint system has been interfaced with, and we will discuss the features that make this system useful. Chapter 4 addresses three themes. First we will show that the precedence graph upon which popular analysis tools in operations management such as PERT/CPM depend is a special case of the temporal constraint satisfaction (TCS) problem. Moreover, TCS
provides a richer set of temporal relationships than the precedence relationship. Secondly, a temporal constraint system provides a metaphor (constraint propagation) to model the down-line effects of perturbations that occur in dynamic operations. Logos-TCS, as a language to express expert problem-solving, provides a declarative framework to define TCS problems. Logos-TCS implements an innovative scheme by embedding a constraint system within a deductive reasoning system (Logos). The third theme of this chapter is on encoding expertise in relaxation strategies when constraint violations occur (situations that cause current operation to fail). Consider the case of an expert who would know when to apply different operations management techniques to solve a problem, e.g., heuristics such as task rearrangement, priority changes, first-in-first-out (FIFO) queue management, shifting start/finish times, or retracting constraints. Logos-TCS provides a simple and straightforward rule-based framework to encode the strategies used by this expert in its reasoning rules.

IV. A Constraint-based Programming Language: Chapter 5 focuses in particular on developing a concise language based on constraint propagation. This language is similar to many pattern matching languages; we will show how a powerful declarative language can be built using the constraint system we developed in Chapter 3. Just as unify is the basis of logic-programming languages, we show how to build a programming language based on constraint satisfaction.

V. Conclusions: This work is not designed to exist in a vacuum. Chapter 6 presents the conclusion of this thesis, in which we discuss future paths
that might be taken in achieving our original goal of providing a planner for the Rice-obot I.
Chapter 2
Background

The work presented in this thesis has several fields as its background. We will briefly describe some of the major work done in each field. The first field that we will describe is that of pattern-matching systems.

There is one very well-known way to solve a problem. Assume that we are presented with a start state and an end state and we wish to solve the problem of reaching the end state from the start state. One way is for us to specify a number of steps that must be taken in order to go from the start state to the end state. This is the procedural method. However, there are other ways we can solve this problem, e.g., we can present the start and end states to a system, and the system can automatically design an algorithm to go to the end state from the start state. This is certainly easier for us, because in such a case all that is necessary for us is to specify our desired goals. We do not need to go to the extra effort of coming up with an algorithm.

As computer systems become more and more complex, it becomes more and more difficult to program them. Thus if we can design and use a system such as this, this will make our task in programming easier. Now, a description of a state can be thought of as a pattern, and our system must be able to understand patterns, and also match them with other patterns or states as necessary. This is the motivation behind the study of pattern-matching systems. In this sec-
tion, we will present a basic pattern-matching algorithm that is used in many logic-programming languages.

A pattern-matching algorithm is a special case of a constraint algorithm. In the case of the pattern-matching algorithm, the type of the constraint is also defined, i.e., two patterns are being constrained to be equal, and the algorithm must derive the conditions under which such an equality holds. Davis [Davis87] presents several constraint propagation algorithms. This is a facet on which we have focused in great depth, and therefore we devote Chapter 3 to this topic. Chapter 3 will describe this algorithm, as well as our enhancements and modifications in detail. Allen [Allen83] also describes a constraint algorithm. Whereas Davis presents a general algorithm, Allen's algorithm is designed specifically to handle the domain of events occurring on a time line. Allen's algorithm is described briefly in this chapter, and we go into detail in Chapter 4.

2.1 Pattern Matching Systems

The most widely used constraint system is to be found in logic programming languages, for example, in Prolog [Shapiro]. The heart of this language is based on the function unify. Unify takes two patterns as arguments and returns an mgu [most general unifier]. A unifier is defined as a substitution. A substitution is a set of bindings. A binding is a pair; it contains a variable and a value that the variable can assume. Substitutions can be thought of as dynamic objects. One presents a pattern to a substitution and receives another pattern as output. The substitution will replace as many variables as possible in the pattern. For example, let us assume that we have the following substitution:
\{A=1, B=2, C=3\}

Furthermore, we are presented with the following pattern:

\((+ A B C D)\)

Then, applying the substitution to the pattern will give us the following result:

\((+ 1 2 3 D)\)

The substitution replaced as many variables as possible in the pattern; however, the substitution was not aware of the value of D, so it left the variable as it is. We will say that D is an unbound variable, given the current substitution.

The unify algorithm operates on patterns. Patterns can consist of variables and/or constants. We will use the notation that any symbol preceded with a "?" is a variable. Thus, ?X is a variable. X, 1 and (+ X 1) all represent constants. To demonstrate the workings of unify, we present the following examples.

<table>
<thead>
<tr>
<th>No.</th>
<th>Pattern 1</th>
<th>Pattern 2</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>[1]</td>
<td>?x</td>
<td>a</td>
<td>?x=a</td>
</tr>
<tr>
<td>[3]</td>
<td>(?x a)</td>
<td>(b ?x)</td>
<td>fail</td>
</tr>
<tr>
<td>[6]</td>
<td>?x</td>
<td>(a ?x)</td>
<td>fail</td>
</tr>
</tbody>
</table>

Unifying a variable with a pattern binds the variable to that pattern. This is the result of Ex. 1. In Ex. 2, ?x is being unified with y, and y is being unified with ?x. However, since there is only one variable in question, there is only one entry in the list of substitutions. In Ex. 3, an attempt is being made to bind ?x
with a and with b at the same time. A variable cannot be bound to more than one value at a time, so this will produce an error; unification is not possible. In Ex. 4, ?x is bound to a. Furthermore, ?y is bound to ?x. ?x already has a value, which ?y will assume as well. In Ex. 5, ?x is bound to ?y; neither one of the variables has a value, so the result is simply the binding ?x=?y. Ex. 6 is somewhat more interesting. An attempt is being made to bind ?x to (a ?x), which itself contains ?x. No possible value of ?x exists to unify ?x with (a ?x), so the result is an error.

Unify can be thought of as a pattern matching mechanism. Any pattern matching mechanism has certain properties. The universe of discourse explicitly contains two disjoint sets, variables and constants. There is also one other set, which can be either implicitly or explicitly defined. This set is a set of connectors, which are used to construct complex patterns from the set of variables and the set of constants. A constraint mechanism will take patterns constructed in this fashion and attempt to match them together. Pattern matching mechanisms can be further thought of as being constraint propagation mechanisms. Two patterns are being constrained to being equal, and the system will propagate a constraint stating that the two patterns are equal. Given this constraint, the system then attempts to infer what conditions must hold on the values of constraint variables for the constraint to be true.

2.2 Temporal Constraint System

The pattern matching mechanism described above makes inferences on the values of variables involved in the system. There are two basic ways in which
one can operate on a constraint system. One way is to operate on variables and assume that the constraints are fixed. This describes our basic constraint system, which makes inferences about the values of variables and changes these values as it sees fit. However, inferences can be carried out in another manner. The other way is to assume that the variables are unchangeable and operate on the constraints. Allen [Allen83] describes a temporal constraint system in which constraints are inferred. The universe of discourse is events that occur on a timeline. Events can be related to each other by temporal relationships. These relationships are described in detail in Chapter 4. We will only mention the names of the relationships here, which are before (<), equal (=), during (d), contains (c), overlaps (o), meets (m), starts (s) and finishes (f). These relationships, along with their inverses, cover all possible relationships between two events. Chapter 4 also discusses in further detail how these relationships can be transformed into the constraint system that we have developed.

Allen describes an algorithm to infer further relationships from the relationships currently given. Assume that A, B and C are two events, and we have the following relationships:

\[ A < B, B < C \]

We show how an algorithm for computing the closure of these relationships. The algorithm maintains a transitivity table of all the relationships. For example, it knows that when it is presented with the above case, then it is possible to infer that \( A < C \). This particular piece of information is maintained in an entry of a table \( T \) as follows:
Each pair of relationships has an entry in the table. This table is hardcoded into
the system. Think of the events as nodes in a graph, and relationships as links in
the graph. Stated simplistically, the algorithm computes all possible paths in this
graph based on $T$; i.e. it will compute the closure of $T$. In this fashion, this type
of constraint system will make inferences about constraints in the network, and
decide on what constraints must hold true, based on the information that it cur-
rently possesses.

2.3 A Plan Generator For Robots

Chapter 9 of [Charniak75] describes a plan generator for robots. In this
section we will describe this plan generator. We have built a working model of
this plan generator, and made some modifications. We will provide descriptions
of the modifications that we made when implementing it, and the reasons behind
these modifications.

Plans can be expressed as a problem in deduction; i.e., a planner can be
thought of an algorithm which finds an answer for the deductive problem: (to-do
task action ?way) that is, to find a ?way to do a task task whose action is action.
For instance, a robot working in a nuclear plant might know that:

(to-do ?task (disable-reactor) (reset switch101)).

Hence, the goal (to-do task73 (disable-reactor) ?way) will deduce that :

?way = (reset switch101).
The general form of a plan is shown in Fig. 2.1. We will then present an example, and discuss a planning procedure using that example.

```
(plan
  objects:  name1 formula1
             name2 formula2
             .......
  steps:   name1 act1
            name2 act2
            .......
  order:   (seq name name)
            (overlap name name)
            .......
  protect: state1 init1 fin1 ; (init and finish
           state2 init2 fin2 ; fields)
           ........)
```

**Figure 2.1**

Fig. 2.2 shows a concrete example of a plan.

```
(to-do ?task (achieve (on ?x ?y))
(plan
  steps:  clearit (achieve (clear ?x))
          findspace: (achieve (space-for ?x ?y))
          do-move: (move ?x ?y)
  order:  (seq clearit do-move)
           (seq findspace do-move)
  protect: ((clear ?x) (end clearit) (begin do-
              move))
            ((space-for ?x ?y)
             (end findspace) (begin do-move))
)
```

**Figure 2.2**

The above is a blocks world example, showing how to get one block of wood ?x on top of another block of wood ?y. The steps clause shows the steps
that must be taken in order to carry out this task. First, we must clear the top of ?x. Then, we must find space on ?y for ?x. Then, we can do the actual move, achieved by the do-move clause. The order clause tells us in which sequence the steps can be initiated. Both clearit and findspace have to occur before do-move. The two protect clauses tell us the protections for various sub-tasks. These protections last from the end of their respective preparatory steps to the beginning of the do-move step. Thus, the expression:

\[
((\text{clear } ?x) \ (\text{end clearit}) \ (\text{begin do-move}))
\]

tells us that the clearit action must be concluded before the do-move action.

In the above we described a plan. In the following, we will describe a plan manager which takes a plan and reduces it to its primitive components. A primitive task is defined as a task that need not be reduced further, i.e., the system already knows how to execute the plan in a single step. Fig. 2.3 describes an algorithm for a simple plan manager that will continue reducing tasks until all tasks have been reduced to their primitives.

Loop: Select a task that is ready to execute (i.e., all predecessors have been executed)
If it is a primitive already then
  execute it.
else find a ?way such that
  (to-do task task-action ?way).
If ?way is a plan then
  create subtasks as necessary.
else
  create a subtask with the action ?way.

Figure 2.3
We have implemented this system, with some modifications. We have replaced the order and protect clauses. Instead, we use our implementation of the temporal constraint system (TCS) we have mentioned in Section 2.2. Thus, instead of the clauses:

\[
\begin{align*}
\text{order:} & \quad (\text{seq clearit do-move}) \\
& \quad (\text{seq findspace do-move}) \\
\text{protect:} & \quad ((\text{clear ?x}) \ (\text{end clearit}) \ \text{(begin do-move)}) \\
& \quad ((\text{space-for ?x ?y}) \\
& \quad \quad \quad \text{(end findspace}) \ \text{(begin do-move)})
\end{align*}
\]

we use the following:

\[
\begin{align*}
\text{sequence:} & \quad (\text{BEFORE} \ \text{(do-move ?x ?y)} \ \text{(clear ?x)}) \\
& \quad (\text{BEFORE} \ \text{(do-move ?x ?y)} \\
& \quad \quad \quad \text{(achieve (space-for ?x ?y))})
\end{align*}
\]

All of the events that occur in the planning session can be regarded as ranges on a timeline. The purpose of the order and protect clauses is to ensure that these events occur in a certain sequence. We have used TCS to describe the sequence and compute the appropriate start and end times during which these events must occur.

This planner that we have seen represents a language in which we can program robots. However, this language does require that every plan, along with its sub-plans, be entered by the programmer into the robot, as well as the sequence of the sub-plans and so on. As such, this language is still procedural in nature. The Rice-obot I is expected to be autonomous and to operate in an
unstructured environment. We believe that it is virtually impossible to anticipate every action that must be taken by the robot in such an environment. Thus, while the implementation of this planner represents an important and useful step in our experimentation, we do not believe that it achieves our final goals.
Chapter 3
Constraint Propagation

There are many problems in artificial intelligence that can be stated in terms of constraints on a set of variables. This chapter describes in detail the constraint propagation algorithm and the development work that went into building a system based on this algorithm.

3.1 Constraint Networks

A constraint network is a system that consists of nodes and constraints imposed upon these nodes. Each variable or node represents a single parameter in the constraint network. Nodes can take values or ranges of values from a given domain. A constraint variable $X$ can take its value from a particular domain, e.g., the natural numbers. This set can be finite, discrete, and contain continuous numeric values and boolean or enumeration types. $X$ is unstrained if it can arbitrarily choose any value from all of the domain.

A constraint is a relationship usually a mathematical or logical relationship among a set of constraint variables. Satisfaction of a constraint satisfaction problem (CSP) can be stated as a process that derives the subset of its legal values (the constraint values) such that the constraints can be satisfied. Solutions to a CSP are derived through the process of constraint propagation. Constraints are used to express relationships between variables which assume these values. The values
of nodes can be either known or unknown. In the cases that they are not known, they can be arbitrarily defined to be either the whole domain or a particular subset of the domain. In such a case, the node is said to be unconstrained. A constraint network is said to be in a stable state when the nodes take on values consistent with all the constraints expressed upon them. It is possible that a node has no value that will satisfy all constraints. In this case, the system is said to be overconstrained. When one or more nodes have more than one possible value, the system is underconstrained.

There are different ways in which constraint propagation can be carried out. One technique is to infer additional constraints from the values of variables that are currently available. The other technique is to infer values for variables that satisfy the current constraints. These different techniques have been discussed in detail in Chapter 2. We have used the value inference technique in our system.

Nodes are implemented as Lisp expressions in our system. Each node has three values associated with it. In our system, these are maintained as property lists. The legal values of a node refer to the complete range of values that the variable can take. These are supplied by the user and are never modified internally by the system. The constrained values of the node are a subset of the legal values. These are the values that the node can assume while the program is attempting to satisfy a set of constraints. The current value is a scratch register maintained by the system. It is used in bindings and environments, which are explained later.
Constraints are expressions that represent logical relations among their sub-expressions. The constraint system recognizes two types of connectors which can be used to build constraint expressions. Relational connectors represent one type, e.g. ":="", "<" or ">=". Other connectors are expressional connectors, such as ":+", ":-" and ":*". Constraints are defined as being relational expressions. They express relationships between their sub-expressions. Thus every constraint must contain a relational connector which will express a relationship between certain expressions. An arbitrary number of expressional connectors can occur in every constraint. These will be tied together by a relational connector. The following examples demonstrate the nature of constraints accepted by our system:

\[(= x y)\]

The above is a constraint that says that \(x\) has to be equal to \(y\). This constraint contains no expressional connectors. It contains the relational expression ":=".

\[(<= (+ w z) (+ y z))\]

This constraint contains the expression connector ":+" twice and states that the sub-expressions \((+ w z)\) has to be lesser than or equal to \((+ y z)\).

\[(+ x y)\]

While this is a legal sub-expression for a constraint, it is not a legal constraint itself because it does not contain any relational expressions.

\[(= (= x y) true)\]
This is not a legal constraint expression because it contains more than one relational expression, i.e., ":=".

A symbol is defined as being overloaded if it can appear in different contexts. For example, in several languages the following are legal statements:

\[ x := 1+1; \quad x := \text{"aa" + "bb"}; \]

The symbol "+" is said to be over-loaded because it can be used in both of the above contexts, i.e. for integer addition and string concatenation. In our system, over-loading of symbols is not allowed, so a connector cannot be both relational and expressional at the same time. The system is expandable. New connectors of both the expressional or relational type can be added as considered necessary. Temporal expressions, discussed in the next chapter, use relational connectors of this nature. There are some restrictions on what type of function can be translated into a connector that the constraint system can actually use. These restrictions will be enumerated here, and the reasons for these restrictions will be discussed after the constraint propagation mechanism is described.

3.2 Constraint Connectors

**Relational Connectors:** Assume that the function associated with a relational connectors takes \( n \) arguments. The system requires that if the function is called with \( n \) arguments then it return a *true* or *false* value. Consider the case of the natural numbers, and the function \(<\). For every pair of natural numbers \( a \) and \( b \), it is known whether \( a < b \). However, there are several predicates and domains which cannot be described in this manner. For example, consider the
set of pairs of natural numbers. For some two pairs, it is not known whether (a, b) < (c, d), unless one designs an acceptable definition of <.

**Expressional Connectors:** Consider an arbitrary function of the type:

\[ f(x_1, x_2...x_n) = F. \]

The function \( f \) takes \( n \) arguments and returns a value \( F \). The restriction made by the constraint system is that if any \( n-1 \) arguments and \( F \) are provided, then another function must exist to calculate the missing argument. The constraint system has to have all of these functions, as well as \( f \), before this connector is considered acceptable. Fortunately, these conditions are easy to satisfy for many important connectors, like arithmetic addition and subtraction.

### 3.3 System Description

In the following, we will discuss implementation of the constraint system that we have developed. We will give a background detailing the data structures that we maintain, and then discuss the algorithm used.

#### 3.3.1 Environments

A binding is a relationship between a variable and certain values. A variable can take its values from a subset of the domain we are addressing. A binding is defined as being a mapping from the variable domain to the value range. An environment is a set of variable bindings. As such it can be better stated to be a mapping from a set of variables to a set of values. Environments are not complete mappings. They are used to represent only the variables associated with a
particular constraint system. In the implementation bindings are maintained as extended properties of the expressions in question.

Environments can be very similar to each other, perhaps differing in that only one or two variables are bound to different values. The concept of sets of environments is a useful one in this context. We define a complete set of environments for a system. Consider the case where we have the following three variables, along with their associated values:

\[
\begin{align*}
A \text{ in } \{1, 2\} & \quad B \text{ in } \{\text{cat, dog}\} & \quad C \text{ in } \{3, 4\}
\end{align*}
\]

Following is a complete set of environments over \{A, B, C\}:

\[
\begin{align*}
((A \ 2) \ (B \ \text{dog}) \ (C \ 3)) & \quad ((A \ 2) \ (B \ \text{dog}) \ (C \ 4)) \\
((A \ 2) \ (B \ \text{cat}) \ (C \ 3)) & \quad ((A \ 2) \ (B \ \text{cat}) \ (C \ 4)) \\
((A \ 1) \ (B \ \text{dog}) \ (C \ 3)) & \quad ((A \ 1) \ (B \ \text{dog}) \ (C \ 4)) \\
((A \ 1) \ (B \ \text{cat}) \ (C \ 3)) & \quad ((A \ 1) \ (B \ \text{cat}) \ (C \ 4))
\end{align*}
\]

\textbf{Figure 3.1} A complete set of environments

A complete set of environments is a set of environments defined on a set of variables. Within each environment, all variables take on certain values, and each environment has a combination of values which is unique within that set. A complete set is one which contains all possible environments. To compute a complete set, a simple combinational algorithm is sufficient, as environments do not have to be ordered sets. We should note that in the case where every variable in a system has exactly one value, the size of the complete set of environments is exactly one.
We now have a sufficient background to be able to discuss the constraint satisfaction mechanism.

3.3.2 Constraint Satisfaction Algorithms

A constraint can be considered to be satisfied when the evaluation of the expression that represents it yields a value of TRUE. For example consider the constraint \( A+C=5 \), with \( A \) and \( B \) having values as described above. This can be satisfied either by choosing the bindings \( \{A=1, C=4\} \) or \( \{A=2, C=3\} \). Any environment that contains these two bindings will satisfy this constraint.

Consider the case where only discrete values are allowed in the system. In this case, we already have an easy way to describe the constraint satisfaction process. All we have to do is to examine each environment individually and determine whether the variable bindings satisfy the constraints in the expression or not. The following presents an off-line constraint satisfaction. An off-line system is one which does not have any input. The system is started with an initial state and eventually produces some output and halts. For our example, the system is initiated with a set of constraints and legal values of constraint variables. It eventually halts and produces a result. This has the disadvantage that the user cannot interactively assert a new constraint into a system without having to re-compute the effects of the new constraint from scratch. It is often the case that the programmer wishes to find the effects of a small number of constraints on a stable system. In such a case, the off-line algorithm is computationally much more expensive.
To satisfy a constraint system $S$
foreach environment $E$ in the complete environment set of $S$
begin
    foreach variable binding $(Var, Val)$ in $E$
    Var := Val;
    end foreach variable binding

    foreach constraint $C$ in the constraints list of $S$
    Evaluate expression for $C$.
    if the evaluation does not return TRUE
    go on to next environment.
    end foreach

    mark $E$ as a satisfactory environment
end

Output: List of all satisfactory environments.

Figure 3.2: Off-line constraint satisfaction algorithm

An *on-line* system is one which can deal interactively with a user. The user can examine the state of a system and incrementally assert constraints to determine their effects. There are also other reasons why an on-line system is desirable. For example, the system may be part of a client/server pair. In this pair, the constraint system will assume the role of the server and accept constraints produced by the client.

The algorithm works in the following fashion. We will assume that we already have a consistent system $S$. A consistent system is one in which all constraints are satisfied with the values of all the variables, and thus there is no conflict between different constraints in the system. In our particular case, an on-line system is presented with one constraint at a time. As it is possible for a system to have no constraints at all, this algorithm can be used to inductively derive any
system that could be built using the off-line algorithm. The system will compute
the effects of the constraint and then come to a halt.

To add constraint C to a constraint system S
Let T = a stack of variables, initially empty
Let P = a stack of constraints, initially empty

foreach variable V in the parameter list of C
    add C to the constraint list of V
endforeach variable V

while P is not empty
    let X be the constraint popped off P

    attempt to satisfy X
    let L = the list of variables constrained
        while satisfying X
            foreach variable V in L
                add V to T
            endforeach variable V

            foreach variable V in L
                foreach constraint N in the constraint
                    list of V-C
                        add N to P
                endforeach constraint N
            endforeach variable V
    endwhile P is not empty

Output: A complete set of environments on T.

Figure 3.3: On-line constraint satisfaction algorithm

In the cases of both algorithms, if no environments exist, and the output
is the empty set, then the system is in an error state; there is a constraint violation.
If one or more environments exist which do satisfy all the constraints in the
system, then the system is considered to be satisfiable.
We should note the conceptual similarity of the unify algorithm presented in Chapter 2 with these constraint satisfaction algorithms. The unify algorithm takes as input two expressions. The output of the algorithm is the bindings for the variables in the expressions that will make the two expressions equivalent. As such, the following two expressions can be regarded as equivalent.

\[(\text{unify } A \ B) \quad \text{ (constrain } (= A \ B))\]

Thus, unify can be thought of as a special case of constrain.

3.3.3 Constraint Propagation Mechanism

We use the on-line algorithm in our system. Constraints are asserted into the system in interactive mode; i.e., a constraint is asserted, and the system attempts to achieve a quiescent state before proceeding to a state where it is ready to accept more input. When a constraint is asserted, the system analyzes the expression and extracts its parameters. These are the constraint variables of that particular constraint. The set of variables, along with its current set of values represents the working set for the system. The constraint is also inserted into the list of constraints that the system associates with each of the constraint's parameters (the system uses this technique to maintain information about all the constraints that mention any particular parameter). Next, the system will be attempt to satisfy its constraints. The constraint variables represent the initial working set of variables for the program. A complete set of environments is created out of the values associated with these variables.
Consider the following system:

**variables:**
- \( x \{1, 10\}, y \{3, 8\}, z \{2, 7\} \)

**constraints:**
- \( x + y = z, y \leq x \)

The values of the variables \( x, y \) and \( z \) together violate the constraint \( x+y=z \). We define range addition and subtraction in the following manner:

\[
\begin{align*}
[a, b] + [c, d] &= [a+c, b+d] \\
[a, b] - [c, d] &= [a-d, b-c]
\end{align*}
\]

Thus, \( x+y \) will evaluate to \([1, 10] + [3, 8] = [4, 18]\). The value of \( z \) is defined as being \([2, 7]\). The system will now attempt to equate \( x+y \) with \( z \). The intersection of the ranges represented by \( x+y \) and \( z \) is the only value that the "=" predicate finds acceptable, i.e., \( \text{Intersection}([4, 18], [2, 7]) \) is \([4, 7]\). These values can be now propagated back down the network. They are propagated down to \( z \), whose values are restricted to the intersection of its current values and \([4, 7]\). Coming to the expression \( x+y \), \([2,7]-[1,10]\) are propagated to the \( y \) node and \([2,7]-[3, 8]\) is propagated to the \( x \) node.

The system is not satisfied yet, as changes have been made to \( x, y \) and \( z \). The modified variables become the new working set for the system. The system finds all constraints that have parameters which belong to the working set. In our particular example, these are the constraints \( y \leq x \) and \( x+y=z \). The system knows that \( x+y=z \) can be satisfied with the current values (it was the one that caused the current propagation). Therefore, it will only check \( y \leq x \). This checking entails propagating constraint effects using \( y \leq x \) in the same fashion as was done with \( z=x+y \). Checking will continue until all constraints are satisfied with
the values of their parameters, or until it is known that some constraint is unsatisfiable.

The system tries to restrict the values of variables as little as possible. For any final value of any particular variable in the system, it is guaranteed that some value exists for each of the other variables that satisfies the constraints in the system, unless the system is over-constrained. It is not guaranteed that all values of the other variables will satisfy these constraints.

We can now explain the restrictions we have imposed on our system. We mentioned that expressional combinators must be invertible. Consider the case of propagating values in the following constraint:

\[(= (+ x y) z)\]

The "=" predicate forwards values to the sub-expressions of the constraint. Thus \((+ x y)\) receives a value that it must propagate further. We know the value of the sub-expression \((+ x y)\) and the values of \(x\) and \(y\). However, we must compute the values that must be propagated to \(x\) and to \(y\). For the sake of this example, let us assume that the result is 7 and the current values of \(x\) and \(y\) are 3 and 4, respectively. In this case, we have to propagate 4 and 3 to \(x\) and \(y\). We know that we have to propagate 4 to \(x\) because we know that \(x\) has to take the value 4 in the expression 7=3+x. To generalize, any function appearing instead of the "+" must be invertible; i.e., if the result of the function is known and values of all but one of the arguments is known, then there must be a way to compute a value for the unknown argument.
We further introduce the concept of *constraint violations*. In the above example, we were attempting to propagate the value 4 to the variable $x$. The original value of $x$ was also fortunately 4, so nothing untoward happened. However, there are cases when a constraint cannot be satisfied. Assume that we attempted to satisfy the following system:

\[
\begin{align*}
\text{variables:} & \quad x = 1, \ y = 2, \ z = 4. \\
\text{constraints:} & \quad x + y = z
\end{align*}
\]

It is impossible to satisfy the system because $1 + 2$ does not equal 4. The system discovers an error when it attempts to propagate the value 2 to $x$ and the value 3 to $y$. $x$ and $y$ have the values 1 and 3, respectively, which cannot match 1 and 2. On discovering this error, the system signals a *constraint violation*. This signaling can take the form of informing the user, or executing some code that the user of the system considers to be appropriate.
Chapter 4
Integration With Other Systems

4.1 Motivation

Operations management problems such as project planning and scheduling, resource allocation and optimization are common in both industrial and administrative domains [Buffa87]. In operations research (OR) [Hiller86], there are valuable analysis tools such as Performance Evaluation and Review Techniques (PERT), Critical Path Methods (CPM), and the Gantt chart. Optimization procedures such as linear programming solve some important aspects of these problems. However, these tools assume a static problem definition and, for large-scale operations, take an excessive amount of time to compute the optimal solution. In real life operations, dynamic changes, updates or perturbations, from minor to serious, are a large part of the problem. Before re-running expensive optimization procedures, it is preferable to analyze the down-line effects of these perturbations and reason about what-if scenarios, and check if simple heuristic rules are available to patch the problem. For decisions that have to be made in a short period of time and with limited computational resources, we may not even want an the optimal solution because it may be too expensive to compute.

Logos-TCS is a system designed for modeling dynamic perturbations in the operation management domain. It provides a computational metaphor (con-
straint propagation) to model the down-line effect of changes. Logos-TCS implements the constraint system in a deductive framework and provides a state maintenance mechanism allowing the user to hypothetically visualize the future scenarios by asking what-if questions.

This chapter is organized around three themes. First, what can temporal constraint systems do in operations management? Would the constraint system provide a rich model to represent the operation management problems? Secondly, why use a temporal constraint system? How does it model the dynamic perturbations? Third, how does the temporal constraint system work? How do we encode heuristic rules (constraint relaxation) in Logos-TCS?

4.2 A Model by Temporal Constraints System

The first theme of this chapter is to show that the popular PERT/CPM or Gantt chart in operations management is a special case of the temporal constraint satisfaction (TCS) problem. The precedence relationship, represented in a precedence graph in PERT/CPM, is a subset of temporal constraint relationships. In Logos-TCS, we adopt with some semantic modifications the 13 temporal relationships defined by Allen. We will show that the precedence relationships in PERT/CPM are exactly the BEFORE and MEETS relationships in TCS.

4.2.1 Temporal Constraint Satisfaction

Constraint Satisfaction has been used in circuit layout/analysis and simulation applications, and lately in scheduling and resource allocation problems. Depending on the domain and the constraints in question, a general constraint
system can be inefficient. For certain types of constraints, efficient algorithms do exist. However, the metaphor of constraint propagation (discussed in the preceding chapter) provides a powerful problem-solving paradigm.

4.2.2 Temporal Constraint Relationships

Allen listed 13 possible relationships in 7 categories to specify the temporal relationships between two events as shown in Fig. 4.1.

<table>
<thead>
<tr>
<th>x before y</th>
<th>x</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td>x meets y</td>
<td>x</td>
<td>y</td>
</tr>
<tr>
<td>x equals y</td>
<td>x</td>
<td>y</td>
</tr>
<tr>
<td>x during y</td>
<td>x</td>
<td>y</td>
</tr>
<tr>
<td>x overlaps y</td>
<td>x</td>
<td>y</td>
</tr>
<tr>
<td>x starts y</td>
<td>x</td>
<td>y</td>
</tr>
<tr>
<td>x finishes y</td>
<td>x</td>
<td>y</td>
</tr>
</tbody>
</table>

Figure 4.1 Allen's temporal relationships

These temporal relationships can be expressed as a constraint satisfaction problem by the following translation. An event or activity is specified by three constraint variables: "event-begin, event-end, event-duration" and one constraint:
"event-begin = event-end + event-duration". Fig. 4.2 shows the table of the constraint expressions mapped. We say that r1 covers r2 if two events related by the relationship r2 can also be said to be related by r1. The relationships as defined by Allen are disjoint sets. We have not preserved this property. Thus we have made slight semantic modifications to the language. Note that in our mapping, BEFORE covers MEETS, DURING covers EQUALS, OVERLAPS covers MEETS, STARTS covers EQUALS, and FINISHES covers EQUALS.

<table>
<thead>
<tr>
<th>x before y</th>
<th>x-end ≤ y-begin</th>
</tr>
</thead>
<tbody>
<tr>
<td>x meets y</td>
<td>x-end = y-begin</td>
</tr>
<tr>
<td>x equals y</td>
<td>x-begin = y-begin, x-end = y-end</td>
</tr>
<tr>
<td>x during y</td>
<td>x-begin ≥ y-begin, x-end ≤ y-end</td>
</tr>
<tr>
<td>x overlaps y</td>
<td>x-begin &lt; y-begin, x-end &lt; y-end, x-end ≥ y-begin</td>
</tr>
<tr>
<td>x starts y</td>
<td>x-begin = y-begin, x-end ≤ y-end</td>
</tr>
<tr>
<td>x finishes y</td>
<td>x-begin ≥ y-begin, x-end = y-end</td>
</tr>
</tbody>
</table>

Figure 4.2 Temporal constraint expressions
Consider Fig. 4.3, depicting a PERT chart representing a house-building example from the book [Hiller86]. All PERT-type systems use a network of nodes and edges to portray graphically the precedence relationships among the activities. An activity is represented as an edge while a node indicates either the beginning or the end of that activity. With each activity, there is a number indicating the duration of the activity. There are dummy (dashed) edges that represent no activity but only precedence relationships. The pair of numbers associated with each node indicates the earliest time (left), and the latest time (right) of the beginning/end of the activity without delaying the completion of the whole project. The slack time of a node is defined as the difference between the latest and the earliest time.

4.3 The PERT chart for building a house
In temporal terminology, if the slack time of a node is zero, then the previous activity, in the precedence sense, MEETS the next activity. If the slack time is greater than zero, the previous activity is BEFORE the next activity. Using Logos-TCS, the house-building example can be expressed declaratively as follows:

\[
\begin{align*}
\text{:EVENT Excavate BEGIN (0) DURATION (2))} \\
\text{:EVENT Foundation DURATION (4))} \\
\text{:EVENT Rough-Wall DURATION (10))} \\
\text{:EVENT Roof DURATION (6))} \\
\text{:EVENT Rough-Exterior-Plumbing DURATION (4))} \\
\text{:EVENT Rough-Interior-Plumbing DURATION (3))} \\
\text{...} \\
\text{:TEMPORAL-CONSTRAINT Excavate BEFORE Foundation) } \\
\text{:TEMPORAL-CONSTRAINT Foundation BEFORE Rough-Wall) } \\
\text{:TEMPORAL-CONSTRAINT Wall-Board BEFORE Flooring) } \\
\text{:TEMPORAL-CONSTRAINT Rough-Wall BEFORE Roof) }
\end{align*}
\]

In Logos-TCS, an activity node in PERT/CPM corresponds to a temporal event which in turn is translated into three constraint variables (-begin, -end and -duration). Figure 4.4 shows the Logos-TCS display for the house-building example. In this figure, an event is displayed by its start and finish time. The start time is displayed first, followed by the finish time overlayed in the same row. A "+" sign indicates that the event has a fixed start or finish time. In other words, the constraint value of the -begin or -end variable of the event is a point value (i.e., not an interval), and it is tightly constrained. A "horizontal bar" next to a variable name indicates the range of possible start and finish times of the variable. The size of the interval is exactly the slack time calculated by the CPM algorithm, that is, the constraint value of the variable is an interval, and the variable is loosely constrained.
Figure 4.4 Logos-TCS display of house-building example I
Besides BEFORE and MEETS, the temporal constraint system provides richer concepts such as DURING, OVERLAPS, EQUALS, STARTS and FINISHES. It greatly increases the expressive power of possible relationships between two temporal events. For example, if multiple resources are available, we now can express parallel activities as follows:

(:TEMPORAL-CONSTRAINT Interior-Painting STARTS Exterior-Siding)
(:TEMPORAL-CONSTRAINT Interior-Plumbing DURING Exterior-Painting)
(:TEMPORAL-CONSTRAINT Wall-Board OVERLAPS Exterior-Fixtures)
(:TEMPORAL-CONSTRAINT Roof OVERLAPS Exterior-Plumbing)

Furthermore, through the constraint system, the -begin, -end and -duration of event can be loosely quantified as follows:

(:EVENT Excavate BEGIN ((0 2)) DURATION ((2.5)))
(:EVENT Rough-exterior-plumbing END (10) DURATION ((4 7)))

The first statement denotes that Excavate can begin between now (day 0) to 2 days later; it will take between 2 to 5 days to finish. The other statement indicates that Rough-exterior-plumbing must be finished by the 10th day and it will take 4 to 7 days to finish.

4.3 Dynamic Modeling and Visualization of Down-line Effects

What will happen to the completion time if the rough-exterior-plumbing activity has to be postponed for 5 more days in our house-building example? When can the house be finished if an extra activity has to be inserted? Figure 4.5
shows the new situation based on the change (rough-exterior-plumbing activity has to be postponed for 5 more days). As shown in this figure, this event causes the slack time to be stretched out as compared with the previous example. In operations management, analyzing the down-line effects of a perturbation and visually displaying the propagation of the effects is highly useful.
# Logos-TCS: Temporal Constraints System

<table>
<thead>
<tr>
<th>Task</th>
<th>Start</th>
<th>End</th>
</tr>
</thead>
<tbody>
<tr>
<td>EXCAVATE</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>FOUNDATION</td>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>ROUGH-WALL</td>
<td>5</td>
<td>16</td>
</tr>
<tr>
<td>ROOF</td>
<td>16-23</td>
<td>22-31</td>
</tr>
<tr>
<td>ROUGH-EXTERIOR-PLUMBING</td>
<td>16-23</td>
<td>25</td>
</tr>
<tr>
<td>ROUGH-INerior-PLUMBING</td>
<td>23-25</td>
<td>29</td>
</tr>
<tr>
<td>WALLBOARD</td>
<td>30-38</td>
<td>38</td>
</tr>
<tr>
<td>FLOORING</td>
<td>38-45</td>
<td>42-49</td>
</tr>
<tr>
<td>INTERIOR-PAINTING</td>
<td>38-45</td>
<td>42</td>
</tr>
<tr>
<td>INTERIOR-FIXTURES</td>
<td>43-49</td>
<td>49</td>
</tr>
<tr>
<td>EXTERIOR-EXTERIOR-PAINTING</td>
<td>22-31</td>
<td>29-38</td>
</tr>
<tr>
<td>EXTERIOR-PAINTING</td>
<td>29-38</td>
<td>28-47</td>
</tr>
<tr>
<td>EXTERIOR-FIXTURES</td>
<td>30-47</td>
<td>40-49</td>
</tr>
<tr>
<td>ROUGH-ELECTRIC-WORK</td>
<td>16-27</td>
<td>19-30</td>
</tr>
<tr>
<td>FINISH</td>
<td>49</td>
<td>55</td>
</tr>
</tbody>
</table>

**Figure 4.5 Logos-TCS display of house-building example II**
In our constraint system, there are only two ways to activate the constraint propagation process: (i) a change in the set of constraints, i.e., a constraint being deleted, added or modified, or (ii) a change of the legal values of the constraint variables. The constraint system will ensure that the constraint values of the variables, at the time of any query, always satisfy the set of outstanding constraints and legal values given at that time, unless a constraint violation occurs. Consider the following example:

\[\text{:EVENT Excavate BEGIN } \langle 0 \ 2 \rangle \text{ DURATION } \langle 2 \ 5 \rangle\]
\[\text{:EVENT Rough-exterior-plumbing BEGIN } \langle 8 \rangle \text{ DURATION } \langle 4 \ 7 \rangle\]
\[\text{:TEMPORAL-CONSTRAINT Excavate Overlaps Rough-exterior-plumbing}\]

It states that the event Excavate begins between 0 and 2, and takes between 2 and 5 to complete. The Rough-exterior-plumbing event begins exactly at 8 and takes between 4 and 7 to complete. The constraint states that Excavate must overlap Rough-exterior-plumbing. However, this is not possible because the last that Excavate can possibly finish is 7 and Rough-Exterior-Plumbing does not start until 8. As a consequence, there is a constraint violation, and an error is signalled. This signalling is similar to the constraint violation signalling that we described in the previous chapter. As a consequence, the same error handling mechanisms can be invoked.

We will now briefly describe Logos-TCS as a language to define a CSP (constraint satisfaction problem). Logos-TCS integrates our constraint system into a deductive framework (Logos). This innovative method allows us to embed
the constraint system inside the reason maintenance system (RMS) of Logos. In this framework, the consequences of constraint propagation can be maintained, explained and retracted as logical deductions.

4.3.3 Deductive Reasoning in Logos

Logos implements a framework similar to the system in Duck [McDermott83]. As with Duck, Logos is a deductive language based on predicate calculus. It consists of valid predicate calculus sentences and rules such as (-> antecedent consequence) in forward deduction (Modus Ponens), or (<= consequence antecedent) in backward deduction. However Logos provides a state descriptor construct to model the concept of state and state changes of an event.

4.3.4 State Descriptors

In Logos, facts denote the static facts of a state. For example,

(inst Fido dog)
;; Fido is a dog. This will not change. Fido will always be a dog.

A state descriptor denotes the current state that is expected to change:

(:= (location-of Fido) in-the-house) ;; Fido is in the house now
(:= (status-of M-1) in-use) ;; the current state of M-1 is in-use

The symbol "=" can be interpreted as "IS", as in the last example: "the status of M-1 IS in-use". The "=" has stack-based semantics that model the concept of the current state and the previous states. Depending on the order of the assertions, proper state information is maintained by pushing onto a stack. Consider the following sequence of assertions, queries and retractions:
Assert that \( x \) has a value which is a range from 1 to 10.
\[
\mathtt{assert:} \; (\mathtt{:\text{legal-value} \; x} \; (\mathtt{:interval \; 1 \; 10}))
\]

Assert that \( y \) has a value which is a range from 1 to 100.
\[
\mathtt{assert:} \; (\mathtt{:\text{legal-value} \; y} \; (\mathtt{:interval \; 1 \; 100}))
\]

Assert that \( x \) and \( y \) have to be equal.
\[
\mathtt{assert:} \; (\mathtt{:\text{constraint} \; (= \; x \; y)})
\]

Now we try to find out the value of \( y \):
\[
\mathtt{query:} \; (\mathtt{:\text{legal-value} \; y} \; ?\mathtt{value})
\]
We find that \(?\mathtt{value}=\mathtt{(interval \; 1 \; 10)}\); i.e., the value of \( y \) is the same as \( y \).

We change the value of \( x \):
\[
\mathtt{assert:} \; (\mathtt{:\text{legal-value} \; x} \; (\mathtt{:interval \; 15 \; 16}))
\]

Now the value of \( y \) has also changed:
\[
\mathtt{query:} \; (\mathtt{:\text{legal-value} \; y} \; ?\mathtt{value})
\]
We find that \(?\mathtt{value}=\mathtt{(interval \; 15 \; 16)}\); i.e., the value of \( y \) is different.

If we retract our above assertion about the value of \( x \):
\[
\mathtt{retract:} \; (\mathtt{:\text{legal-value} \; x} \; (\mathtt{:interval \; 15 \; 16}))
\]

and we attempt to find out the values of \( x \) and \( y \):
\[
\mathtt{query:} \; (\mathtt{:\text{legal-value} \; x} \; ?\mathtt{value-x})
\]
\[
\mathtt{query:} \; (\mathtt{:\text{legal-value} \; y} \; ?\mathtt{value-y})
\]

we find that the values of \( x \) and \( y \) are still the same. However, the values have now both changed back to the interval \([1, \; 10]\). The value of \( y \) has followed the value of \( x \) because of the constraint that requires both to be equal. The constraint
used the state descriptor semantics to maintain information about the proper values of y.

This demonstrates the implementation of a powerful mechanism to maintain *hypothetical reasoning*. Consider the case of a robot which has certain information about the world. The robot wants to carry out a certain action; e.g., it is trying to cross a bridge. Assume that the robot has observed that the bridge is made of wooden planks and that the planks are rotten. The robot should now attempt to compute all the consequences of stepping on that bridge. Internally, the robot has a structure of information that we can refer to as the *world model*. To compute the consequences of stepping on the bridge, the robot makes changes to its internal world model. The position of the robot can be represented as a state descriptor. To model the action of stepping on the bridge, we make the assertion:

\[(\equiv (\text{robot position}) \text{ bridge})\]

Now, assume that the robot knows the consequences of stepping on an unstable bridge. This piece of information is encoded as a *rule* as follows.

\[\rightarrow (\text{and} (\equiv (\text{robot position}) \text{ bridge}) (\equiv (\text{state bridge}) \text{ unstable})) (\equiv (\text{robot state}) \text{ wet})\]

The fact \[(\equiv (\text{state bridge}) \text{ unstable})\] is already a part of the world model. Asserting \[(\equiv (\text{robot position}) \text{ bridge})\] will *satisfy* the above rule, and a new assertion is made: that the robot's new state is wet. The fact \[(\equiv (\text{robot state}) \text{ wet})\] is called a *down-line effect* of the fact \[(\equiv (\text{robot position}) \text{ bridge})\]. The robot can examine what has now become the current world model and decide whether it really wishes to proceed on the intended course of action. Assume that it consid-
ers becoming wet to be an undesirable consequence. The robot can then simply retract the assertion that it had made previously, i.e., retract the following fact:

\[(:= \text{(robot position)} \text{bridge})\]

When the above fact is retracted, all facts that depended on this fact are also retracted, and we are left with a world model where the robot is contemplating the bridge, which is our initial state (the point of time at which we originated this example). To make sure that this computation is not wasted, we have to mark the fact as being an undesirable fact. This is quite simple and only entails modifying a (possibly global) variable that exists outside the data structure which contains the world model.

In the above, we have demonstrated how to carry out hypothetical reasoning using a state mechanism. This state mechanism can be used to ask what-if questions of the nature "what will happen if I do this" and allows us to make intelligent decisions on how to deal with undesirable down-line effects of actions that we might consider.

4.3.5 Constraint System in a Deductive Framework

In this section we will discuss integration of the constraint system within our deductive framework and state descriptor mechanism. As we have already shown, constraints and constraint variables are declared in the following forms:

```
;; declaring constraint variables and their legal values
(:= (:LEGAL-VALUE X) ((:INTERVAL 1 10)))
(:= (:LEGAL-VALUE Y) ((:INTERVAL 3 8)))
(:= (:LEGAL-VALUE Z) ((:INTERVAL 2 7)))
```
;; declaring constraints
(:CONSTRAINT (<= Y X))
(:CONSTRAINT (= Z (+ X Y))

Using state descriptors in declaring the constraint variables and their legal values allows results from different constraint propagations to be maintained on a stack. Moreover, the results of the constraint propagation process are maintained under the RMS as consequences. For example, from the above constraints and variables, the following are three conclusions (the constraint values) that are inferred from the constraint propagation:

(:= (:CONSTRAINT-VALUE X) (:INTERVAL 3 4))
(:= (:CONSTRAINT-VALUE Y) (:INTERVAL 3 4))
(:= (:CONSTRAINT-VALUE Z) (:INTERVAL 6 7))

More precisely, Logos defines a dummy rule denoting that the constraint values are inferred based on the current set of constraints and the constraint variables (Modus Ponens). In this way, the constraint values are maintained through the RMS; results can be recorded, explained and retracted as a deduction.

- a dummy constraint propagation rule
(-> (AND (CONSTRAINS) (LEGAL-VALUES))
 (CONSTRAINT-VALUES))
antecedents:
(:= (:LEGAL-VALUE X) (:INTERVAL 1 10))
(:= (:LEGAL-VALUE Y) (:INTERVAL 3 8))
(:= (:LEGAL-VALUE Z) (:INTERVAL 2 7))
(:CONSTRAINT (<= Y X))
(:CONSTRAINT (= Z (+ X Y)))
Consequences:
(= (:CONSTRAINT-VALUE X) ((:INTERVAL 3 4)))
(= (:CONSTRAINT-VALUE Y) ((:INTERVAL 3 4)))
(= (:CONSTRAINT-VALUE Z) ((:INTERVAL 6 7)))

In summary, Logos embeds the constraint propagation within the deductive framework as a forward deduction. The result of constraint propagation (the constraint values) are consequences of the constraints and variables. Being able to record, trace and explain the down-line effects of perturbation of world models, modeled by constraint propagation through the same RMS chains is a unique feature of Logos.

4.4 Encoding Expertise In Constraint Relaxation Strategies

For daily operation management tasks, simple heuristics such as rearrangement based on priority, delay, cancel, or start early are often the major source of expertise. For most of the minor perturbations, such heuristic rules are adequate to patch the problem. However, when the number of affected factors grows, the solutions are not intuitively simple.

In the constraint system's terminology, a constraint violation occurs when a set of constraints can not be satisfied. For complex systems, a perturbation may cause a chain of changes in a system and eventually lead to a constraint violation. When this happens, the action that causes the constraint propagation will be rejected and Logos asserts the following two clauses:

(= (:CONSTRAINT-VARIABLE :ERROR) -var-)
(= (:CONSTRAINT-VIOLATION :ERROR) (:CONSTRAINT -expr-))
These two assertions indicate that a constraint violation occurs in checking the legal values of variable -var- against the constraint expression -expr-. Again, the state descriptors allow different states of constraint violations to be maintained. For example,

\[ (\text{:= } (:\text{LEGAL-VALUE Y}) (\text{(:INTERVAL 10 18)})) \]
\[ (\text{:= } (:\text{LEGAL-VALUE X}) (\text{(:INTERVAL 5 9)})) \]
\[ (:\text{CONSTRAINT} \ (\leq \ Y \ X)) \]

will cause assertion of the following:

\[ (\text{:= } (:\text{CONSTRAINT-VIOLATION} :\text{ERROR}) \ (\text{(:CONSTRAINT} \ (\leq \ Y \ X)))) \]
\[ (\text{:= } (:\text{CONSTRAINT-VARIABLE} :\text{ERROR}) \ X) \]

There are two ways to relax the constraint system, either by relaxing the legal values of constraint variables or relaxing constraints (retracting or modifying the constraints). Logos provides the same declarative framework so that users can encode their constraint resolution strategies in rules as follows.

For the purpose of this example, we assume that we know that x has values in the interval [2, 5]. We are also given a fact that the boundaries of the range of x should be extended by 3 should a constraint violation occur (these two pieces of information represent the domain-dependent knowledge in our system). These can be encoded as the following:

\[ (\text{:= } (:\text{LEGAL-VALUE X}) (\text{(:INTERVAL 2 5)})) \]
\[ \text{(relax-variable X by 3)} \]
\[ \ldots \text{ insert other constraint variables and constraints} \ldots \]
The following is a relaxation strategy rule. There are four facts that must be satisfied, before this rule can come into play. These are: [i] A constraint violation must occur, and be caused by \(?\text{var}\). [ii] The constraint violation must involve a constraint \(?\text{ex}\). [iii] There must be domain-dependent knowledge that the system can use to find out what strategy to invoke in relaxation. [iv] The legal values of \(?\text{var}\) must be in the interval \([?\text{lb}, ?\text{ub}]\).

```lisp
;; relaxation strategy - relaxing variable
(\(\rightarrow\) (AND (= (:CONSTRAINT-VARIABLE :ERROR) \(?\text{var}\))
 (= (:CONSTRAINT-VIOLATION :ERROR)
  (:CONSTRAINT \(?\text{ex}\))
  (relax-variable \(?\text{var}\) by \(?n\))
  (= (:LEGAL-VALUE \(?\text{var}\) ((:INTERVAL ?lb ?ub))))
  ; antecedents

  (= (:AND (= (:LEGAL-VALUE \(?\text{var}\))
             ((:INTERVAL (- ?lb ?n) (+ ?ub ?n))))
      (:CONSTRAINT \(?\text{exp}\)))
  ; consequents
)
```

Should the above rule be satisfied, two things will occur. [i] The range of values that the variable \(?\text{var}\) can take will be expanded to \([?\text{lb}-3, ?\text{ub}+3]\). [ii] The constraint \(?\text{ex}\) will be re-asserted into the system. The following states the above rule in a more concise form:

... explanation of above strategy
IF a constraint violation occurred on \(?\text{VAR}\), and the constraint violated is \(?\text{EXPR}\), and the strategy to relax \(?\text{VAR}\) is to relax its legal values by \(?\text{N}\), and the current legal values of \(?\text{VAR}\) are from \(?\text{LB}\) to \(?\text{UB}\)

THEN assert the new legal values as from \((- ?\text{LB} ?\text{N})\) to \((+ ?\text{UB} ?\text{N})\), and re-assert the constraint \(?\text{EXPR}\)

As we did in the above example, the user will write relaxation rules to intercept the two assertions when a constraint violation occurs. These can be thought of as
interrupts that occur when an error condition is signalled. In general, the relaxation strategies are domain-dependent.

Let us consider another example. Assume that we are interested in assigning priorities to variables. When a constraint violation occurs, we want to be able to select a specific variable that will be relaxed. To illustrate our example, let us examine more closely the mechanism in the constraint violation above. Assume we are attempting to assert a constraint of the form (= x y). The system above chooses a variable and expands its range. The system does not know of any preferences about which variable to choose, so it will arbitrarily pick one of x and y and relax the variable. However, there may be cases where we have priorities on our variables. Consider the case where we have two conflicting events, e.g., attend-class and watch-tv. Furthermore, let us assign numeric priorities of -1 and 10 to the variables, respectively. We have the constraint and the domain-dependent knowledge encoded below:

```
(:= (relaxation-strategy
    (:CONSTRAINT (<= attend-class and watch-tv)))
   (relax-lowest-priority-variable by 5))
```

We use the following rule. We introduce a LISP function called get-lowest-priority-variable which will find the variable with the lowest priority. We assume that this function has the appropriate domain-dependent knowledge and can decide which variable is more important. In the example below, the other clauses are used to find out what the constraint is, and to re-assert it into the system.

```
(-> (AND (:= (:CONSTRAINT-VIOLATION :ERROR) ?con)
         (:= (relaxation-strategy ?con)
         ```
(relax-lowest-priority-variable by ?N)
(:BIND ?var (:FUNCALL get-lowest-priority-variable ?con))
(= (:LEGAL-VALUE ?var) ((:INTERVAL ?lb ?ub))))
(= (:LEGAL-VALUE ?var) ((:INTERVAL (- ?lb n) (+ ?ub n))))

Following is an algorithmic description of the relaxation process.

... explanation of above strategy
IF a constraint violation occurred on ?CON
the strategy is to relax
the variable of the lowest priority by ?N
the variable ?VAR is found
by an external function call
the current legal values of ?VAR are
from ?LB to ?UB

THEN assert the new legal values as
from (- ?LB ?N) to (+ ?UB ?N)

Instead of relaxing constraint variables, another option available to us is to
retract the constraint causing the error. The following states that our preferred
method of resolving a constraint violation is to retract the constraint.

(= (relaxation-strategy (:CONSTRAINT (<= Y X)))
  retract-constraint)

The following rule will fire if our preferred method of dealing with constraint vi-

lations is to retract the constraint causing the violation if a violation occurs.

(-> (AND (= (:CONSTRAINT-VIOLATION :ERROR) ?con)
    (= (relaxation-strategy ?con)
        retract-constraint))
    (:RETRACT ?con))

... explanation
IF a constraint violation occurred on ?CON and
the strategy is to retract ?CON
THEN retract ?CON
4.5 Summary

In this chapter, we presented three themes: First, the precedence graph, on which analysis tools in operations management such as PERT/CPM depend, is a special case of the temporal constraint satisfaction (TCS) problem. TCS provides a richer set of temporal relationships (or vocabulary) for expressing relationships between two events. It allows for several more relationships than the BEFORE/AFTER in PERT/CPM. Secondly, constraint propagation provides a sound metaphor to model the down-line effects of perturbations in managing real operations. Logos-TCS provides an innovative scheme embedding our constraint system within a deductive framework. Thus, the user can specify a constraint satisfaction problem in a declarative form as a logic deduction problem. The third theme, relaxation strategies, in the constraint satisfaction context, encodes the heuristics that most human managers use when the original plan fails and a constraint violation occurs. Logos-TCS provides the same deductive framework in which heuristic knowledge (or constraint relaxation strategies) can be expressed in a simple and straightforward way.
Chapter 5
A Constraint Language

In this chapter, we will address the issues of usability and expressiveness with regards to the constraint system that we developed in Chapter 3. Chapter 4 showed the applicability of constraint propagation to scheduling management and operations management problems. As such the constraint mechanism was regarded as a tool that interfaced with other tools. This chapter will deal with the constraint system as a language. We will take a bottom-up approach; initially, we will discuss the constructs used in task-planning, and the issues involved in implementing these constructs. We will then proceed to see how these constructs can be useful in programming.

5.1 Necessary extensions to the Constraint System

The first extension to the constraint system that we must make is to allow the ability to name constraints. This is necessary in order to be able to carry out recursion. Given that we have the ability to name constraints, we next incorporate an ability to recursively define constraints (we define operators that let us build new constraints out of previous ones). Constraints are currently described in terms of relational connectors and expressional connectors. A relational connector can and must appear exactly once in any constraint. Sub-expressions built out of expressional connectors can as complicated as desired. Thus the sub-expressions built using only expressional combinators are indeed recursively defined. Since we have the limitation that a relational expression can exist only
once inside a constraint, constraints as a whole cannot be recursively defined. However, it is still desirable to have recursive definition as part of our language because it is well-understood how many problems might be tackled using recursion. We introduce another type of connector, a *logical* connector, which will allow us to recursively define constraints. It will also give us an added benefit of incorporating decision-making facilities in our language. Finally, we will incorporate parameter-passing.

All of the facilities we have mentioned above are necessary in a general-purpose programming language. If we implement all of them, we have a language that can be used to compute anything that is computable.

### 5.2 Implementing Recursion

In almost any programming language, it is necessary to be able to execute a fixed body of code while some condition holds true. Several languages have the **while** construct. Other languages use recursion to implement a construct similar to **while**. There is one feature that both techniques have in common. Both have a fixed point or a label that represents a *re-entry point*. A re-entry point is a piece of code that names a part of a program; i.e., it provides a mechanism to refer to the code that it represents. For the **while** loop construct, the label is not explicitly named. It is implicitly declared just before the execution point of the first statement in body. For recursive languages, the label is explicitly named. It is the name of the function which calls itself recursively. As such, the recursive model is somewhat lower-level and easier to implement because it does not require the compiler or interpreter to implicitly generate labels
during compilation. We will take the recursive approach in our implementation. Having made this decision it is now necessary for us to provide a facility for naming constraints.

Let us consider a simple constraint as described in Chapter 3. It is a relational expression between two or more sub-expressions and describes how exactly these sub-expressions relate to each other. To provide more expressive power to our language, we will extend our definition of what constitutes our constraints. First we will give constraints names. The following represents an example of such a construct:

\[(:\text{constraint-relation c1 (= x y)})\]
\[(:\text{constraint-relation (relation ?queen-1 ?queen-2 in-x)}\]
\[\text{(not (= (x-pos ?queen-1) (x-pos ?queen-2)))})\]

The first example states that (= x y) is a constraint that will be asserted into the constraint system. Furthermore, we have added the facility to address the constraint with a symbolic name. In the first case, this is c1, in the second, it is the following expression:

\[(\text{relation ?queen-1 ?queen-2 in-x}).\]

We will discuss later how this naming is implemented and how we will make appropriate use of the naming facility. The second example illustrates a more interesting use of the naming facility. It says that a relationship exists between any two queens such that their x co-ordinates cannot be the same (this is a fragment of the eight queens puzzle, where one is required to place eight queens on a chessboard such that no queen can attack another queen). With the facility
that we have implemented, it is possible to assert constraints in the system, such
that the names of the constraints are themselves patterns in a pattern-matching
language.

The main reason for naming constraints is that it allows labels in the
system so that we can implement recursion. We implement a facility to build
new, larger constraints out of sub-constraints, as the following illustrates:

\[
(:\text{macro-relation} \textit{c1-2} \\
\quad (= z (+ x y)) \\
\quad (\geq x y))
\]

This construct states that \textit{c1-2} is composed of the following two sub-
relations:

\[
(= z (+ x y)) \quad (\geq x y).
\]

Let us also consider the following system:

\[
(:\text{constraint-relation} \textit{c1} (= z (+ x y))) \\
(:\text{constraint-relation} \textit{c1-2} \\
\quad \textit{c1} \\
\quad (\geq y x))
\]

The two systems described above are equivalent. A simple environment
can be used to maintain information about whether a constraint expression repre-
sents a name or a relation (we should recall that names can also be complex con-
structs). Thus, the expression \((= z (+ x y))\) is legitimately a name for an expression as well as a legal constraint expression. With this construct, we can now
define recursive constraints. However, there is still one further extension that is
desirable in our system. In the constraint system described in Chapter 3, all
variables used in the system had the same scope. There was no concept of
stacked environments as used in conventional programming languages. To be
able to make full use of the recursive capability we have implemented, it is necessary to have a concept of parameter passing. Consider the following example:

\[
(:\text{constraint-relation} \ c1 \ (x \ y \ z) \\
\quad \quad \quad \quad \quad \quad \quad \quad \quad \quad (= \ ?z \ (+ \ ?x \ ?y)))
\]

The above states that \(c1\) is a constraint of three parameters, which are related by the expression \(= z (+ x y)\). The following call will invoke the constraint-checking mechanism:

\[(c1 \ 1 \ 2 \ 3)\]

This expression can now be asserted into the constraint system just as we would assert any other constraint.

5.3 Logical connectors

In a previous section, we showed how to recursively define constraints. Constraints could be expressed in terms of sub-constraints. This can be thought of as implicitly defining a connector. Consider the following:

\[
(:\text{constraint-relation} \ X \\
\quad \quad \quad \quad \quad \quad \quad \quad \quad \quad (b) \\
\quad \quad \quad \quad \quad \quad \quad \quad \quad \quad (c))
\]

The above states that \(X\) is a constraint that is asserted by asserting \(b\) and then asserting \(c\). However, \(X\) might be built using \(b, c\) and some other relationship. For example, we might have a condition where one of either \(b\) or \(c\) have to be asserted, but not necessarily both. In this case, we would have:

\[
(:\text{constraint-relation} \ X \\
\quad \quad \quad \quad \quad \quad \quad \quad \quad \quad (\text{or} \ (b) \ (c)))
\]
We have implemented the logical connectors **and, or** and **not**. The semantics of these are as one would intuitively expect. The **and** connector requires that both of its sub-parts be asserted; **or** requires at least one of its sub-parts, and **not** requires that it not be possible that its one argument be assertable.

### 5.4 Using the language

We will now walk through an example using the features that we have described above. Consider the classical example of appending two lists together. We describe this in the following manner:

\[
\text{(:constraint-relation append (A B C)}
\]

\[
\text{OR (and (= ?A nil) (= ?B ?C))}
\]

\[
\text{(and (= (car ?A) (car ?C))}
\]

\[
\text{(append (cdr ?A) ?B (cdr ?C))))}
\]

append is defined as a constraint of three parameters. It is expected that each of the parameters will be a list (the system will generate an error if it is not). append is further defined as being an **or** combination of two clauses. One of the sub-expressions has to be a constraint that does not cause a constraint violation. Consider the following:

\[
(\text{and (= ?A nil) (= ?B ?C)})
\]

It states that ?A must be the empty list, and that ?B and ?C must be equal. This is the base case of our definition of append; the result of appending an empty list to a second list is the second list itself. The second part of the **or** is more interesting.

\[
(\text{and (= (car ?A) (car ?C)) (append (cdr ?A) ?B (cdr ?C)))}
\]
The above states that the *car* or *head* (i.e., first element) of `?A` and `?C` have to be equal, and we recursively call append on the *cdr* or *rest* (i.e., the remaining elements) of the list.

We will now take a closer look at the propagation mechanism involved here. The result of appending `A` to `B` will be stored in `C`. The above system works fine if `A`, `B` and `C` are all bound variables in the system. However, most logic programming languages allow for a case where unbound variables can be bound based on some relationships. Thus, we would like to be able to go both ways. When `C` is not bound, and `A` and `B` are properly bound, we would like `C` to be bound to the value of `append(A, B)`. The key lies in assertion of the two branches of the second *and*. These are:

\[
(= \text{ (car A) (car C)}) \ (\text{append (cdr A) B (cdr C)})
\]

We will assume that either `A` or `C` is bound, but not both. This is the case in which we are particularly interested. Without loss of generality, let us assume that `C` is the unbound variable. The working of the system depends on the definition of *car* - more precisely the propagation function that propagates values down the network. When values are propagating upwards, the upward propagation function for *car* propagates the value of the *car* of `A` up the network. However, on the downward propagation, we are missing one piece of information. We do not know what the *cdr* value of `A` is expected to be. However, this will be bound when the other branch of the *and* is executed. We will introduce the concept of structured variables. These are variables that can have parts of themselves bound at one time, while other parts are unbound. Thus a cons-cell in this context will
consist of a structured variable of two parts. We can recursively assume that append works. It is obvious that it works for the base case when A is nil.

Logic programming languages allow constructs similar to the ones we have described and used above. They allow for assertions of patterns in a database. Our equivalent for this is an assertion into our constraint system. One can express recursive definitions of patterns which we have also shown how to implement. Patterns can have variables in them which are implemented as parameters to constraints in our system. We also have assertions of the nature of and, or and not, which serve as the basis for decision-making in these languages.

Thus, we have shown how to use the constraint system to design a language similar to a logic programming language. The results we have seen so far have been interesting. We have taken a novel approach for creating a tool that can be used to fulfill the tasks of a logic programming language. This tool is based on constraint propagation instead of the usual unification algorithm [Shapiro]. However, we should now investigate how this language can be made more expressive by using other capabilities of the constraint system.

5.5 Trapping Constraint Violation Errors

One way to enhance a logic programming language is to build error reporting mechanisms into the language. In conventional logic-programming languages, when a goal fails unexpectedly, the programmer has to go through extra effort to understand why it has failed. Ideally, there should be a heuristic that the
programmer should be able to code that will deal with this failure. In this section, we will further describe the construct :relaxation-strategy that was introduced in Section 4.4. The following depicts the fields used in this construct.

(:relaxation-strategy <name-of-strategy>
   <constraint-causing-error>
   <variable-causing-error>
   <preconditions>
   <assertions>
   <constraints>)

The name of the strategy is a LISP symbol. The <constraint-causing-error> and the <variable-causing-error> are similar to the ?con and ?var in these patterns that we described in the previous chapter:

(:= (:CONSTRAINT-VARIABLE :ERROR) ?var)
(:= (:CONSTRAINT-VIOLATION :ERROR) (:CONSTRAINT ?expr))

The <variable-causing-error> serves the same function as ?var. The same hold for ?expr and <constraint-causing-error>. The :relaxation-strategy construct is a specialized rule that is invoked when the above assertions are made. Furthermore, all the facts included in <preconditions> must be true before the rule can be invoked. Once the rule is invoked, the facts in <assertions> become true, and the constraints in <constraints> are asserted into the constraint system.

An example of using :relaxation-strategy is the following:

(:relaxation-strategy for-c1
c1
?x
(:constraint-variable ?x :legal-values ((?lb ?ub)))
(:constraint-variable ?x :legal-values ((- ?lb 3) (+ ?lb 3)))
c1)
The above states that should a violation occur in attempting to satisfy c1, bind ?x to the offending variable. Then, attempt to shift the legal-values of that variable by 3. The following statement in the <preconditions> field is a query statement:

(:constraint-variable ?x :legal-values ((?lb ?ub)))

It is matched with the knowledge base to find out what the range of ?x is. The following statement in the assertions field is an assertion statement:

(:constraint-variable ?x :legal-values (( (- ?lb 3) (+ ?lb 3))))

Now that the values of ?lb and ?ub are bound properly, the assertion is made with all the variables in the pattern being bound properly. The statement c1 instructs the system to attempt re-asserting the constraint c1 when a violation occurs. Should a violation occur when this happens, relaxation strategies are then re-invoked as appropriate.

We thus have a mechanism that can be programmed to automatically deal with failures. Most logic programming languages work in the following method. The system is presented with a pattern to be matched. The system will look in its knowledge base and attempt to match the pattern with all possible clauses in its database. All matching patterns in the knowledge base are presented to the user. However, there are cases when a pattern will almost, but not quite, match some of the patterns in the knowledge base. The user can invoke certain functions to do certain things when a match is made, but only for the case of patterns that do match. It is also desirable to take care of patterns that do not match, and at that point decide whether it is possible to invoke an exception handler. In the follow-
ing section we will discuss using :relaxation-strategy in an example that shows how a programmer can deal with the cases of patterns that almost match.

Let us consider the classical blocks world example. Let us assume that we are trying to stack a red block on top of a green block. This can be done in the following manner:

(:constraint stack (b1 b2)
   (= (colour ?b1) red)
   (= (colour ?b2) green)
   (on ?b1 ?b2)))

However, suppose that the system is out of green blocks. In such a case, we might be willing to compromise and accept blue or yellow blocks. We can then assert the following strategy into the system:

(:relaxation-strategy colour-mismatch
   (= (colour ?b1) green)
   ?colour
   (or (= ?colour blue)
       (= ?colour yellow)))

This strategy says that blocks are now allowed to be yellow or blue as well, and the variable ?colour is tested against appropriate values.
Chapter 6
Conclusions

In Chapter 3, we showed the development of the constraint network. We described the constraint propagation mechanism that we had used in detail. Chapter 4 discussed the integration of the constraint network into our reason maintenance system (RMS). The user is able to specify a constraint satisfaction problem (CSP) in a declarative framework. Rules are declared that model the behaviour of the system. These rules represent the domain-dependant knowledge for the application that the system is being used in. Given a framework for expressing knowledge about a system, we showed the basis of developing a mechanism for handling constraint violations. Chapter 5 showed strengthening of the constraint language. With the results of Chapter 5, we now have a powerful expressive language, which was the stated goal of this thesis.

Where do we go from here? This work is not designed to exist in a vacuum, but rather, it is expected that it will be part of set of co-operating tools that will be used to guide the Rice-obot. Our basic purpose is to provide a comprehensive task-planner for the Rice-obot. This task-planner needs to possess a world model that gives it a realistic and knowledgeable view of the environment that it operates in. We have made two assumptions about the methodology we are adopting to solve the problem. Our first assumption is that the robot environment can be effectively modelled by describing it as a set of constraints. This
seems to be a realistic assumption. Constraints can be used to express relationships about objects in the world, and represent a powerful modelling technique.

Not only does the task-planner have to be able to model the world knowledgeably, but it also has to be able to propose plans that will allow a robot to be able to demonstrate its knowledge by behaving intelligently. Constraint violations can be thought of as hooks. In this context, constraint violations can be thought to represent hooks that are expected to be developed later as we learn more about the domain that we are dealing with. These programs attached to these hooks should be able to deal with the violations that occur. The second assumption we made is that it is possible to install a rich enough set of constraint violation hooks to allow the robot to deal with all the problems that arise in task-planning. The natural consequence of this assumption is that we also believe it possible to write programs to fit in these hooks that will allow the robot to demonstrate knowledgeable behaviour. Implementation of the relaxation-strategy construct has been our first step at providing a set of hooks for the robot programmer.

We expect that future work will focus on using the constraint language that we have implemented to build a viable world model. Furthermore, we believe that it is worthwhile to continue the investigation of using constraint violations as a technique to program the robot. Constraints can be thought of as knowledge representation. Constraint violations can be thought of as actions using the knowledge representation to behave intelligently. We feel that given the
combination of the two, we can construct a comprehensive path-planner of the sophistication that we need.
Bibliography


[Ciscon90] Lawrence A. Ciscon, *Hierarchical Robot Path Planning Using a Distributed Blackboard*, Rice University, Houston, TX, 1990.


