Wm Leler

Constraint Programming Languages
Their Specification and Generation

Here is the first book to explore constraint languages, a new programming paradigm with applications in such areas as computer-aided design, the simulation of physical systems, VLSI, graphics, typesetting, and artificial intelligence. The book provides an introduction to the subject of constraint satisfaction, a survey of existing systems, and introduces a new technique that makes constraint-satisfaction systems significantly easier to create and extend. Because constraint languages are *declarative,* constraint programs are easy to build and modify, and their nonprocedural nature makes them well suited to execution on parallel processors.

HIGHLIGHTS

- Defines a general-purpose specification language called **Bertrand** that allows a user to describe a constraint-satisfaction system using rules.
- Gives examples of how to use **Bertrand** to solve algebraic word and computer-aided engineering problems.
- Gives examples of how to use **Bertrand** to solve problems in graphics involving computer-aided design, illustration, and mapping.
- Provides precise operational semantics for augmented term rewriting, and presents techniques for efficient execution and compilation.

Constraint Programming Languages: Their Specification and Generation is aimed at researchers and engineers who are investigating declarative languages and how to build useful systems using constraint-satisfaction techniques. The book will also be of interest to computer scientists looking for applications of constraint programming languages in a broad range of areas.

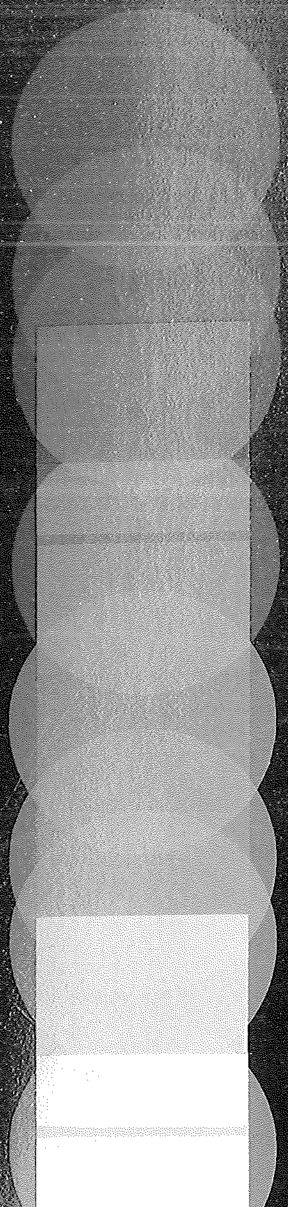
ABOUT THE AUTHOR

Wm Leler received his Ph.D. from the University of North Carolina at Chapel Hill; this book is based on the thesis he submitted there. He is currently a visiting researcher at the University of Manchester, England, and was previously affiliated with the Tektronix Computer Research Laboratory in Beaverton, Oregon.

ISBN 0-201-06243-7

ADDISON-WESLEY PUBLISHING COMPANY

Leler Constraint Programming Languages Their and



Wm Leler

Constraint Programming Languages

Their Specification and Generation

Constraint Programming Languages

Their Specification and Generation

Wm Leler

ADDISON-WESLEY PUBLISHING COMPANY
Reading, Massachusetts • Menlo Park, California • New York
Don Mills, Ontario • Wokingham, England • Amsterdam • Bonn • Sydney
Singapore • Tokyo • Madrid • Bogotá • Santiago • San Juan

This book is in the Addison-Wesley Series in Computer Science
Michael A. Harrison, Consulting Editor

This document was formatted using Troff and PostScript,
and printed using the following fonts:
New Century Schoolbook
Helvetica
Courier
Symbol

Library of Congress Cataloging-in-Publication Data

Leler, Wm (William)
Constraint Programming Languages
Their Specification and Generation
Bibliography: p.
Includes index.
1. Bertrand (Computer program language) I. Title.
QA76.73.B47L45 1988 005.13'3 87-1236
ISBN 0-201-06243-7

Preface

Constraint languages represent a new programming paradigm with applications in such areas as the simulation of physical systems, computer-aided design, VLSI, graphics, and typesetting. Constraint languages are *declarative*; a programmer specifies a desired goal, not a specific algorithm to accomplish that goal. As a result, constraint programs are easy to build and modify, and their nonprocedural nature makes them amenable for execution on parallel processors.

This book is aimed at researchers investigating declarative programming languages and rewrite rule systems, and engineers interested in building useful systems using constraint-satisfaction techniques. It provides an introduction to the subject of constraint satisfaction, a survey of existing systems, and introduces a new technique that makes constraint-satisfaction systems significantly easier to create and extend. A general-purpose specification language called **Bertrand** is defined that allows a user to describe a constraint-satisfaction system using rules. This language uses a new inference mechanism called **augmented term rewriting** to execute the user's specification. Bertrand supports a rule-based programming methodology, and also includes a form of abstract data type. Using rules, a user can describe new objects and new constraint-satisfaction mechanisms. This book shows how existing constraint-satisfaction systems can be implemented using Bertrand, and gives examples of how to use Bertrand to solve algebraic word and computer-aided engineering problems, and problems in graphics involving computer-aided design, illustration, and mapping. It also gives a precise operational semantics for augmented term rewriting, and presents techniques for efficient execution, including interpretation using fast pattern matching, and compilation.

The software described in this document is available for a nominal charge. Inquiries should be directed to the author at the following address:

P.O. Box 69044
Portland, Oregon 97201

Some ideas feel *good* to us. This concept is common enough, although it appeals more to our emotions than our intellect. For example, on the first day of class, the professor of an introductory psychology class I took declared [Nydegger 1973]:

"It will be my task during this semester to convince you that behaviorism is not only correct, but that it is right and good."

Another example appears in the IDEAL user's manual [Van Wyk 1981]:

"To take advantage of IDEAL's capabilities, you must believe that *complex numbers are good*."

IDEAL uses complex numbers to represent two-dimensional graphical points, the advantage being that, because all objects in IDEAL are complex numbers, the same operators and functions can be used on all objects, whether they represent numbers or points.

Why do some ideas feel good? Perhaps there is a measure of beauty for ideas, and some are simply more appealing than others. One may worry, however, that a discussion of aesthetic issues is not compatible with the practice of computer science, and that such arguments belong with other questions of taste, such as those about the right way to indent nested loops or the proper choice of variable names. Indeed, one sometimes hears computer scientists making exhortations resembling those of the psychology professor quoted above. A fairly well-known example of this is the so-called war between the "big-endians" and the "little-endians," concerning whether the bits and bytes of a machine word should be numbered starting from the most or the least significant end.

Should we reject aesthetic considerations as contrary to scientific method? Experience has shown otherwise. Proper attention to the goals of aesthetics leads to measurably better designs. As Fred Brooks says, "Good esthetics yield good economics." [Blaauw & Brooks p. 86, 1986].

The pursuit of good design principles transcends aesthetics. To avoid meaningless arguments about taste, we must give some basis for our aesthetic judgments. Toward this end, Blaauw and Brooks outline four principles of good design: *consistency, orthogonality, propriety,* and *generality.* They apply their principles to computer architecture, but similar principles apply to other areas. In designing computer languages, two key principles are *simplicity* and *generality.* For example, we can say that the use of complex numbers in IDEAL is *good* because it is *simpler* than having two separate data types, and because it is more *general* to allow operators to work on both numbers and points. To paraphrase something a professor of mine once told me, if you find a simple solution to one problem and then find that the same solution simultaneously solves several other problems, then you are probably onto something exciting. Working with constraint-satisfaction systems has been, and continues to be, very exciting.

Acknowledgements

I wish to thank Bharat Jayaraman for his persistent help with this work, and Fred Brooks for his early encouragement and support. I am truly grateful to the wonderful people at the Tektronix Computer Research Laboratory, especially Rick LeFaivre, for allowing me to spend so much time writing this book. Alas, no list of acknowledgements is ever complete—I need to thank the several dozen people and places who somehow found out about this research and either requested copies of this book or invited me to give talks. I especially want to thank several people whose careful reading and comments on this book were invaluable during its preparation: David Maier, Mike O'Donnell, and Chris Van Wyk, and all the people who supplied encouragement, especially Scott Danforth, Marta Kallstrom, Larry Morandi and Philip Todd.

Dedication

Finally, I must step back from this work and admit that it is not really important at all. There are special people out there who have dedicated their lives to ideas that not only *might* make a difference, but *must.* I dedicate this book to Robin and John Jeavons, whose work on Intensive Farming techniques has already made a difference to a hungry world, and to other people like them who have the courage to work long and hard for the important ideas they believe in.

Wm Leler

Contents

Chapter 1

Introduction

1.1 Imperative versus Constraint Programming

In current imperative computer languages, such as C or Pascal, a program is a step-by-step procedure. To solve a problem using these languages, the user must manually construct and debug a specific executable algorithm. This style of problem solving has become so pervasive that it is common to confuse algorithm design with problem solving. The effort required to program using imperative languages tends to discourage programming and thus effectively restricts most users to canned application programs.

To use an algorithmic program to solve different but related problems, the programmer must anticipate the different problems to be solved and include explicit decision points in the algorithm. For example, using a syntax similar to C or FORTRAN one might write the statement:

```
C = (F-32)×5/9
```

to compute the Celsius (C) equivalent of a Fahrenheit (F) temperature. To convert Fahrenheit temperatures to Celsius, however, a separate statement would have to be included in the program; namely,

```
F = 32 + 9/5×C
```

along with a branch (*if*) statement to choose which statement to execute. To be able to convert temperatures to and from degrees Kelvin, even more statements (with the associated branch points) would have to be added:

```
K = C - 273
C = K + 273
K = 290.78 + 5/9×F
F = 523.4 + 9/5×K
```

As new variables are added to this program, the number of statements grows exponentially.

In constraint languages, programming is a *declarative* task. The programmer states a set of *relations* between a set of *objects*, and it is the job of the constraint-satisfaction system to find a solution that satisfies these relations. Since the specific steps used to satisfy the constraints are largely up to the discretion of the constraint-satisfaction system, a programmer can solve problems with less regard for the algorithms used than when an imperative language is used. For the growing number of computer users untrained in traditional imperative programming, this can be a significant advantage. For example, constraintlike spread-sheet languages (such as VisiCalc, Lotus 1-2-3, and others) allow users to solve many different financial modeling problems without resorting to programming in the traditional sense.

In a constraint language, the statement

```
C = (F-32)×5/9
```

is a *program* that defines a relationship between degrees Fahrenheit (`F`) and degrees Celsius (`C`). Given either `F` or `C`, the other can be computed, so the same program can be used to solve at least two different problems, without any explicit decision points. With a typical constraint-satisfaction system, we could also solve for the temperature where the values in degrees Fahrenheit and Celsius are the same (−40 degrees), and so on. The ability to solve many different problems with the same program, even if they were not anticipated when the program was written, is a key advantage of constraint languages.

Constraint programs also are easy to modify and extend — to add the ability to convert between degrees Kelvin (`K`) and Celsius (`C`), only a single additional constraint is required:

```
K = C - 273
```

In addition, a typical constraint-satisfaction system can combine these two relationships in order to convert between degrees Kelvin and Fahrenheit, for example, without requiring any additional statements.

A program in a constraint language consists of a set of relations between a set of objects. In our constraint-language program, `F` and `C` are the objects, which in this case are numbers, and the constraint `C = (F-32)×5/9` is the relationship between these two objects. Given a value for either `F` or `C`, the constraint-satisfaction system can use the problem-solving rules of algebra to solve for the other.

1.1.1 Assignment versus Equality

The difference between imperative languages and constraint languages is highlighted by their treatment of equality. Algorithmic languages require two (or more) different operators for equality and assignment. For example, in FORTRAN the relational operator `.EQ.` returns true or false depending on whether its two arguments are equal, and = is used to assign the value of an expression to a variable, whereas in Pascal = is used as the relational operator and := for assignment.

In a constraint language, equality is used only as a relational operator, equivalent to the corresponding operator in conventional languages. An assignment operator is unnecessary in a constraint language; the constraint-satisfaction mechanism "assigns" values to variables by finding values for the variables that make the equality relationships true.

For example, in the constraint-language statement

```
X = 5
```

the equal sign is used as a relational operator (as in mathematics), but to make this statement true the constraint-satisfaction system will give the value `5` to `X`. Thus the equal sign acts similarly to an assignment operator. Unlike the imperative assignment operator, however, arbitrary expressions can appear as its left argument. For example, the previous statement could be written in many different but semantically equivalent ways:

```
5 = X
X + 1 = 6
3 × X = X + 10
```

Our temperature-conversion program also could be expressed in many equivalent forms. In fact, we might have forgotten the equation for the relationship between `F` and `C`, but remember the following information:

- The relationship is linear (it can be expressed as an equation of the form "$y = m \cdot x + b$").
- 212 degrees Fahrenheit is the same temperature as 100 degrees Celsius (the boiling point of water).
- 32 degrees Fahrenheit is the same temperature as 0 degrees Celsius (the freezing point of water).

This information is easily expressed as the following three constraints:

```
F = M×C + B
212 = M×100 + B
32 = M×0 + B
```

The constraint-satisfaction system will determine the appropriate values for `M` and `B` using the last two constraints, and plug them into the first constraint, yielding the desired `F = 1.8×C + 32`. Constraint languages allow the user greater *expressiveness*, because a program can be stated in whichever way is most convenient.

The treatment of equality in constraint languages also is more natural to nonprogrammers. For example, the statement (here expressed in FORTRAN)

```
X = X + 1
```

has always been a source of confusion to beginning programmers until they learn about the imperative notion of incrementing variables. This statement would be a contradiction in a constraint language because no finite value can be found to satisfy it. Furthermore, a constraint-satisfaction system can detect that this statement is false without knowing the value of X. The rules of algebra can be used to subtract X from both sides of the equation, yielding 0 = 1, which evaluates to false. The equivalent FORTRAN expression

```
X .EQ. X + 1
```

if used in a loop that supplies values for X, will be blindly reevaluated to false over and over for each new value of X. The ability to evaluate an expression containing unknown variables to a constant can be used to advantage by a compiler for a constraint language.

Also note that, in a constraint language, the equal sign expresses an invariant (a *constraint*) between objects that is permanent during the program's execution. In a conventional language, the only time a relation expressed by an assignment statement is guaranteed to hold is just after the statement is executed.

1.1.2 Using Constraints for Computer Graphics

A major advantage of constraint languages is their ability to describe complex objects simply and naturally. As an example, consider the relatively simple task of drawing a regular pentagon using some computer graphics system.

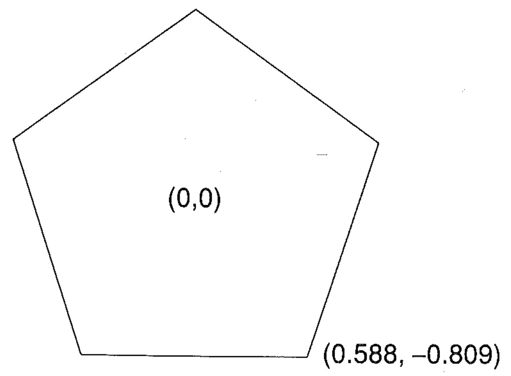


Figure 1.1 Drawing a regular pentagon

This would be very difficult (if not impossible) to do with any degree of accuracy using a typical interactive graphics system such as MacPaint [Kaehler 1983]. With a typical procedural graphics system, such as PIC [Kernighan 1982] or GKS [ISO 1981], this task is reasonable, but the user must specify how to draw the pentagon by specifying the endpoints of the lines. These endpoints must be calculated using

trigonometric functions, and they depend on the location, size, and orientation of the pentagon. For example, if an upright pentagon of unit radius is centered at the point (0, 0), then the lower-right corner is approximately at (0.588, –0.809), as in Figure 1.1. We could possibly specify the endpoints relative to the center of the pentagon, or as a function of the size or even the orientation of the pentagon, but this would be quite a bit of work for a user who just wanted to draw a simple figure.

With a constraint language, a regular pentagon could be specified by drawing any five-sided convex polygon, constraining the vertices to lie on a circle as in Figure 1.2 (or equivalently, to be of equal distance from a point), and constraining the edges to be of equal length.

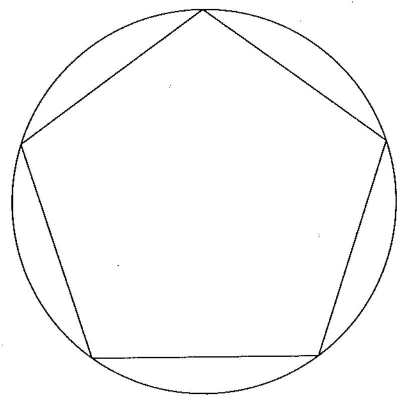


Figure 1.2 Describing a regular pentagon

This not only is easier to do, but also results in a more concise description because it does not depend on any extraneous information, such as the location, size, or orientation of the pentagon. It also does not constrain the order in which the sides of the pentagon are drawn (for example, to allow a program to optimize pen movement for a plotter). Finally, it is much easier to generalize the constraint description to other regular polygons. If desired, constraints can be placed on the location and size of the figure by constraining the location and size of the circle (note that these constraints are independent of the number of sides in the figure). We can also place constraints on the orientation of the figure, for example by constraining the x coordinates of the top point of the figure and the center of the circle to be equal, or by constraining the bottom to be horizontal.

The declarative semantics of constraint languages allow us to describe graphical objects while avoiding extraneous concerns about the algorithms used to draw them. Graphics imagery especially benefits from this because it is inherently spatial and is produced only grudgingly by current procedural languages.

1.2 Scope of the Research

This section defines more precisely some of the terms that have been used informally in the preceeding discussion. A **constraint** expresses a desired relationship among one or more objects. A **constraint language** is the language used to describe the objects and the constraints. A **constraint-language program** is a program written in a constraint language; this program defines a set of objects and set of constraints on these objects. A **constraint-satisfaction system** finds solutions to constraint-language programs. The constraint-satisfaction system uses problem-solving methods, called **constraint-satisfaction techniques**, to find the values of the objects that will make the relationships true.

These definitions are broad and can be interpreted to include a wide variety of systems, from languages that allow some constraintlike statements, to special-purpose systems that satisfy relations between objects. For example, some imperative programming languages (such as Euclid and Common LISP) have an ASSERT statement, and even FORTRAN has an EQUIVALENCE statement that effectively asserts that two variables are always equal. Also included by these definitions would be languages such as the graphics language PIC [Kernighan 1982] and spread-sheet languages which allow relations to be specified, but require that these relations be ordered so that the values of the variables can be calculated in a single pass. At the opposite end of the spectrum allowed by these definitions, symbolic-algebra systems can solve systems of simultaneous equations, and integer programming techniques can be used to find optimal solutions to systems of inequalities.

This book describes an emerging class of (possibly general-purpose) programming languages that use constraint-satisfaction techniques, which I will call **constraint programming languages**. From this designation, I will exclude a vast number of programming languages that use constraint-satisfaction techniques incidentally, or that allow constraints, but require the user to indicate how they are to be solved (typically by ordering them), and will consider only those declarative languages that use constraint satisfaction as their primary computation mechanism. I will also consider only those constraint languages that can reasonably be called programming languages, as opposed to systems that solve constraints but are not generally considered to be programming languages, such as typical symbolic-algebra systems. This distinction may seem arbitrary, but it is analogous to the distinction between logic programming languages such as Prolog, and resolution theorem-proving systems.

I will also concentrate on languages that deal primarily with numeric constraints, and will deal only briefly with languages that use searching techniques to solve logical constraints, such as Prolog. In the future, however, these two classes of languages may not be so distinct. There are already languages being proposed that

may be able to deal with both types of constraints. For the present, however, we will concentrate on techniques for solving numeric constraints.

As is true of most programming languages, a major concern will be the execution speed of constraint programming languages. For example, some constraint-satisfaction techniques will be of interest despite their weak problem-solving abilities because they can be interpreted quickly, or are amenable for compilation. The ability to compile constraint programs will be of major interest in evaluating constraint-satisfaction techniques.

This book will consider only numeric constraint programming languages and constraint-satisfaction systems. Therefore, unless otherwise noted, I will use the shorter terms **constraint language** and **constraint-satisfaction system** to refer to such languages and the systems that interpret them. I also will consider only those constraint-satisfaction techniques that are suitable for implementing these constraint languages. Many potential constraint-satisfaction techniques will not be discussed (or will be only mentioned briefly) simply because they are too slow to be used by a programming language.

1.3 Problem Solving versus Constraint Programming

Because of the high level of specification possible in constraint languages, it is much easier to state constraints than to satisfy them. This is in contrast to conventional imperative languages, where it is relatively easy for the compiler to "satisfy" a correctly specified algorithm. In a constraint language, it is easy to specify correctly problems that any constraint satisfier cannot solve. For example, consider the following constraints:

$$x^n + y^n = z^n$$
$$x, y, z, n \text{ are positive integers}$$
$$n > 2$$

Finding a set of values that satisfies these constraints would constitute a counterexample to Fermat's last theorem. This is obviously an extreme example, but there are many problems, easily solvable by human problem solvers, or even by special-purpose computer programs, that cannot be solved by currently used constraint-satisfaction techniques.

The descriptive nature of constraint languages makes it easy to describe problems, which is one of their major advantages, but it also makes it tempting simply to express a problem to a constraint-satisfaction system and then expect it to be solved automatically. Constraint-satisfaction systems, however, are *not* meant to be general-purpose problem solvers. They are not even as powerful as many mechanical problem solvers, such as symbolic-algebra systems. Constraint-satisfaction systems are meant to solve quickly and efficiently the little, trivial problems that surround and obscure more difficult problems. This frees the user's problem-solving skills for use on the more interesting problems. Thus constraint-satisfaction systems should not be thought of as problem solvers; they are tools to help humans solve problems.

This is not to say that constraint languages cannot be used to solve difficult problems. After all, languages such as LISP have no problem-solving abilities at all, but they can be used to build powerful symbolic-algebra systems and other problem solvers. Constraint languages add a small amount of problem-solving skill, and so reduce the size and difficulty of the task the user must deal with. This is roughly analogous to the way that LISP systems automatically take care of garbage collection, so the user need not be concerned with the management of storage. With LISP, we pay for automatic storage management by giving up some execution speed. With constraint languages, because most problem-solving methods are application specific, we give up some programming generality.

An Example

What a constraint language can do is to make it easier for a human problem solver to describe a problem to a computer, and thus make it easier to apply the computer's abilities to the problem. For example, calculating the voltages at the nodes of a small network of resistors requires the solution of a few dozen simultaneous equations. There are several possible approaches (ignoring constraint languages) to finding the solution for such a problem:

- Set up the equations and solve them by hand. This is what most people would do, but it is a tedious and error-prone task.
- Write a program in an imperative language to calculate the voltages. Unfortunately, writing an imperative program to solve simultaneous equations is more difficult, and just as error-prone, as solving the equations by hand. Writing such a program would be worthwhile only if the user needed to solve many problems of this type.
- Use an existing special-purpose problem solver, such as a symbolic-algebra system, to solve the simultaneous equations. The user would still have to figure out what the simultaneous equations are from the circuit, and each change to the circuit will require a new set of equations.

Using a constraint language, this problem can be described simply as a network of connected resistors, and the constraint-satisfaction system can set up the simultaneous equations automatically (an example of this is given in Section 5.7). This allows the user to concentrate on designing the circuit. While it is performing the calculations for the voltages, the constraint satisfier can also check to make sure that

we do not burn up a resistor by putting too much power through it. A human problem solver should not be bothered with such details.

In general, the issue is not *how difficult* are the problems that a constraint-satisfaction system can solve, but rather *how efficiently* can the constraints of interest be solved. What the constraints of interest are depends on the application. Another issue is how easy is it for the constraints of interest to be stated to the constraint-satisfaction system. If a constraint language is general-purpose and extensible, then the user can tailor the language to the application, making the constraints of interest to the specific application easier to state.

1.4 Limitations of Existing Constraint Languages

Problem-solving systems are typically very difficult to implement, and constraint-satisfaction systems are no exception. Even though constraint languages have been around for over twenty years, relatively few systems to execute them have been built in that time. Graphics researchers are still praising Ivan Sutherland's Sketchpad system [Sutherland 1963], built in the early 1960s, but few have attempted to duplicate it. Furthermore, the constraint-satisfaction systems that have been built tend to be very application specific and hard to adapt to other, or more general, tasks. Consequently, despite the significant contributions of existing constraint languages, they have not found wide acceptance or use. There are several causes of this problem, and existing constraint languages suffer from one or more of them:

- General problem-solving techniques are weak, so constraint-satisfaction systems must use application-dependent techniques. It is usually difficult to change or modify these systems to suit other applications. The few constraint languages that can be adapted to new applications are adaptable only by dropping down into their implementation language. For example, the ThingLab simulation laboratory [Borning 1981] allows an experienced programmer to build simulations (which in effect are small constraint-satisfaction systems for solving limited classes of problems) by defining new objects and constraints, but these new constraints must be defined procedurally, using Smalltalk.*
- The data types operated on by typical constraint languages are fixed. There is no way to build up new data types (such as by using *records* or *arrays* as in conventional languages). For example, in Juno [Nelson 1985], an interactive graphics constraint language, the only data type is a two-dimensional point. In order to use Juno for even a slightly different application, such as three-dimensional graphics, the underlying system would have to be modified extensively.

* Recent enhancements allow some constraints to be defined functionally or graphically [Borning 1985a, 1985b].

- Some constraint languages allow the definition of new data types, but new constraints that utilize these new data types cannot be added. New constraints correspond to procedures in conventional languages. In IDEAL [Van Wyk 1982], another graphics constraint language, the only primitive data type is a point, but new data types such as lines, arrows, and rectangles can be defined. Relations between the nonprimitive data types, however, must be expressed in terms of primitives (points). So, for example, to draw an arrow between two rectangles, separate constraints must be expressed connecting the head and tail of the arrow to the desired points on the rectangles. This is only a limitation on expressiveness, but, like a conventional language without subroutines, it does tend to make a constraint language unwieldy. It also takes away some of the benefit of using a constraint language. For example, it is of little advantage when a constraint language allows us to define a new data type for a resistor if we then have to describe each connection between resistors in terms of primitive constraints between their voltages and currents. We would much rather be able to define constraints for connecting resistors together.

- Many existing constraint languages do not allow any computations to be expressed beyond what can be expressed by a conjunction of primitive constraints. So even if new constraints can be added to the language, these new constraints may be severely limited. In Juno, for example, new constraints can be added as long as they can be expressed as a conjunction of Juno's four primitive constraints. One of Juno's primitives asserts that two line segments are to be of equal length, so we can add a constraint that two line segments are perpendicular by using the standard geometric construction of a perpendicular bisector. Unfortunately, there is no way to express *betweenness* (for example, that a point lies between two other points on a line). This constraint could be expressed if we could only say that the *sum* of two distances is equal to a third distance, but we cannot compute sums. Consequently, many objects cannot be uniquely specified. For example, given the constraints that we used to define a pentagon in Section 1.1.2, Juno might instead produce a pentagram (five-sided star), since Juno does not allow constraints on the relative order of the vertices.

- Even in constraint languages that do allow computation (such as IDEAL, which includes the normal complement of arithmetic operators), few are computationally complete. This is a consequence of the difficulty of adding control structures (such as conditionals, iteration, or recursion) to a nonprocedural language, such as a constraint language, without adding any procedural semantics. Consequently, there are computable functions that these languages cannot compute. For example, without iteration (or recursion) it is impossible to express the general concept of a dashed line (where the number of line segments is not fixed). To solve this problem, IDEAL had to add a new primitive (the *pen* statement) that is

a much restricted form of an iterator. The few constraint languages that are computationally complete are so only because they allow the constraint programmer to drop down into an imperative language (typically LISP or Smalltalk). Unfortunately, this also destroys the declarative semantics of the constraint language. It is possible to add control structures to a declarative language without adding procedural semantics (as in Lucid [Wadge 1985], Pure LISP, and others), so it should be possible to add them to a constraint language.

- Even if we do not require computational completeness, if our language does not have conditionals then constraints that depend on other constraints cannot be expressed. Such constraints (called **higher-order constraints**, discussed in Section 2.2) allow us to tailor the solution of a set of constraints to different circumstances. For example, we might wish to express a constraint that centers some text inside a rectangle, unless the width of the text is too wide, in which case the text is to be broken onto multiple lines.

- Many constraint-satisfaction systems use iterative numeric techniques such as relaxation. These techniques can have numerical-stability problems; a system using these techniques might fail to terminate even when the constraints have a solution, or might find one solution arbitrarily for constraints with more than one solution. For example, Juno uses Newton–Raphson iteration for satisfying constraints and so for the pentagon example in Section 1.1.2 it will arbitrarily return either the desired regular pentagon or a pentagram depending on the shape of the initial polygon. This can lead to unexpected changes to the result when some only slightly related constraint is modified. Also, this means that the answer might depend on the order in which the constraints are solved, which effectively destroys any declarative semantics.

In summary, systems to execute constraint languages are difficult to implement, and once one is implemented we are typically stuck with a special-purpose language that, suffering from one or more of the above problems, is just as difficult to modify to apply to other applications. What is needed is an easier way to implement constraint languages, which also avoids all of the above problems. We would like the languages so implemented to be computationally complete (while retaining purely declarative semantics, of course), so we can handle any constraint whose solution is computable, including higher-order constraints. In addition, something like abstract data types would allow new data types and constraints to be defined. And, of course, it must be fast.

One possible approach would be to generate constraint-satisfaction systems using a rule-based specification language — similar to the way parsers can be built by parser generators that accept specifications in the form of grammar rules. Of course, in order to specify a constraint-satisfaction system, not only must we specify the syntax of the constraint language (as for a parser), we must also specify its semantics (what the constraints mean), and, even more difficult, we must give rules that specify how to satisfy the constraints. In order to have abstract data types, we must also be able to define new data types, and be able to control the application of rules to objects based on their type.

1.5 Proposed Solution

This book presents a general-purpose language called **Bertrand** (after Bertrand Russell), which is a solution to the problem of building constraint-satisfaction systems. Bertrand is a rule-based specification language — a constraint satisfaction system is specified as a set of rules and is automatically generated from those rules. Bertrand allows new constraints to be defined, and also has a form of abstract data type.

The major goal of this book is to show that Bertrand makes it easier to build constraint-satisfaction systems. In order to demonstrate how easy it is to build constraint-satisfaction systems using this language, we will examine how existing constraint languages would be used to solve some example problems,* and then generate a constraint language using Bertrand to solve the same problems. These examples will also serve to show that the constraint languages generated using Bertrand are as powerful as existing constraint languages.

The mechanism used to interpret the rules specifying a constraint-satisfaction system is a form of term rewriting. Term rewriting [Bundy 1983] has been used to build interpreters for languages other than constraint languages. For example, the equational interpreter, a term rewriting system developed by Hoffmann and O'Donnell at Purdue, has been used to build interpreters for LISP and Lucid [Hoffmann 1982]. Bertrand uses an extended form of term rewriting, which I call **augmented term rewriting**. Within the framework of term rewriting, augmented term rewriting includes the ability to bind values to variables, and to define abstract data types. These extensions make term rewriting powerful enough that it can be used to build interpreters for constraint languages.

Augmented term rewriting shares with standard term rewriting several desirable properties: it is general-purpose and has a simple operational semantics, which makes it easy to execute. In addition, augmented term rewriting has properties that make it possible to take advantage of well-known optimizations, so the same mechanism also helps solve the execution-speed problem. It can be implemented efficiently as an interpreter using fast pattern-matching techniques, or compiled to run on a conventional processor or even a parallel processor such as a data-flow computer.

* In most cases, the problems will be substantial examples taken from the thesis or other document describing the existing constraint language.

Constraint-satisfaction systems can be easily described using Bertrand and efficiently implemented using augmented term rewriting.

The remainder of this book is divided as follows:

- Chapter 2 discusses existing constraint-satisfaction techniques. These techniques will be used to build constraint-satisfaction systems using Bertrand.
- Chapter 3 describes augmented term rewriting, especially how it differs from standard term rewriting. This chapter also introduces the Bertrand programming language, shows its connection to augmented term rewriting, and gives some examples of its use.
- Chapter 4 describes existing constraint languages, and presents example problems for them to solve. In Chapters 5 and 6, constraint languages will be built using Bertrand to solve these same problems.
- Chapter 5 uses Bertrand to build an equation solver based on algebraic transformation techniques. This algebraic constraint-satisfaction system is used as a base on which the other constraint languages are built to solve the example problems.
- Chapter 6 discusses how graphic input and output can be added to Bertrand, and uses the resulting language to solve constraint problems involving graphics.
- Chapter 7 discusses how augmented term rewriting is amenable for efficient execution, including showing how parallelism can be detected and utilized.
- Appendix A gives further examples of constraint languages built using Bertrand, including listings of the rules for a simultaneous equation solver and a graphics library.
- Appendix B presents a formal operational semantics for augmented term rewriting, and discusses its properties.
- Appendix C gives the code for a working interpreter for an augmented term rewriting system.