

**CONSTRAINED OBJECTS
FOR
MODELING COMPLEX SYSTEMS**

by

**Pallavi Y Tambay
October 2003**

A dissertation submitted to the
Faculty of the Graduate School of The State
University of New York at Buffalo
in partial fulfillment of the requirements for the
degree of
Doctor of Philosophy

Department of Computer Science and Engineering

Copyright by
Pallavi Yashwant Tambay
2003

“Shree”

Acknowledgements

I take this opportunity to express my sincere thanks to my advisor Dr. Bharat Jayaraman. I am grateful for his guidance throughout my work, and for his patience in letting me work at my own pace. He has, in the true sense of the Sanskrit word, been my *Guru*. I thank my committee members Dr. Li Lin, Dr. Rohini Srihari and Dr. Shapiro for their valuable comments. I thank Dr. Lin for his guidance in the case study on modeling gear trains. I thank Dr. Shapiro for his careful review of every page of my dissertation and his comments. I also thank Dr. I. V. Ramakrishnan for agreeing to be my external reader.

I thank Abhilash Dev, Kapil Gajria, Narayan Menon, Prakash Rajamani and Palaniappan Sathappa for their contribution in the development of the domain-specific modeling tools. Special thanks to Abhilash Dev for his contribution in the modeling tool for circuits that paved the way for similar tools in other domains. I thank Rachana Jotwani for her contribution in the development of the domain-independent modeling tool. I thank Ananthraman Ganesh and Anuroopa Shenoy for their contributions in the development of the tool for interactive execution. I thank Jason Chen for providing me with material on understanding gears and for his explanation of this material. I thank Pratima Gogineni for her contribution in the development of the model of a flowsheet. I thank Kevin Ruggiero, Yang Zhang, and those students of cse 505 and cse 605 who programmed in Cob and gave me valuable feedback for making the language more robust. I thank Gill Porter whose interest in this project helped secure a grant from Xerox Corp.

I thank the past and present secretaries of the Computer Science Department at UB, Ellie Benzel, Sally Elder, Peggy Evans, Joann Glinski, Jodi Reiner, Jaynee Straw for taking care of all my paper work, always with a warm smile. I thank Jennifer Braswell for providing me with an ergonomically sound workdesk. I thank Helene Kershner for keeping a benevolent eye on me through the years. I also thank Ken Smith and the staff on cse-consult for providing quality technical services.

I also thank my friends in Buffalo who made my stay here pleasant and memorable. I thank Pavan Aduri, Fran Johnson, Shama Joshi, Jiangjiang Liu, Girish Maheshwar, Vani Mandava, Sanu Mathew, Sangeetha Raghavan, John Santore, and Ankur Teredesai. I espe-

cially thank my former housemates Meera Gupta, Mrs. Gupta, Vani Mandava, and Manika Kapoor, for helping me in my daily chores and making my diet restrictions their own. I am deeply moved by the friendship that Pavan, Fran, and John have shared with me over the years. I also thank Sanu Mathew for his incredible patience and his quiet words of encouragement.

I wish to thank Vani Verma and Deepti Vaidya for our long standing friendship. I thank Hina Naseer, Sanjay Suri, and my friends from PUCSD who've always shown confidence in me. I wish to thank my cousin Varsha Apte, and Jiten Apte for providing a home for holidays. I thank Fran and Tom Johnson for putting me up and letting me use their car in my last few weeks in Buffalo. I thank the members of the Buffalo Amma Satsang who have always had kind words for me. I also express my appreciation for the many meals Mrs. Sandhya Deshpande cooked for me during my years in Fergusson College.

I take this opportunity to also thank all my teachers from my high school, Kendriya Vidyalaya (S.C.), Department of Mathematics, Fergusson College, Pune, and the Computer Science Department of the University of Pune. Their teaching, and encouragement have been, for me, a source of confidence in myself.

I express my deep sense of gratitude, for the services of Vaidya Chandrashekhar Joshi, Hakim Totawala, Dr. Wacker and Dr. Carnevale. I also thank P.T. Daniel Brown and my yoga teacher Mrs. Apte.

I also take this opportunity to thank all my cousins, aunts, uncles, and my grandmothers who have always showered me with love, shown undying confidence in me and been the kind of supportive family that one feels lucky to have. I especially express my appreciation for my grandmother, Vahini, who put in long nights playing solitaire beside me, giving me company as I prepared for my exams. I express my appreciation for the love, encouragement, and support that my sister-in-law has given me. I thank my brother for the encouragement and inspiration he has provided, and last but not least, for his sense of humor. I thank my parents for the inner peace they bring me. They are an example of hard work, dedication, and sacrifice for me.

It is with a deep sense of satisfaction and gratitude that I dedicate this dissertation to all of the above.

Contents

1	Introduction	1
1.1	Motivation and Significance	1
1.2	Objectives and Results	3
1.2.1	Technical Approach	3
1.2.2	Scope of the Dissertation	4
1.3	Summary of Dissertation	7
2	Background	8
2.1	Constraint Programming	8
2.1.1	Constraint	8
2.1.2	Constraint Satisfaction	9
2.1.3	Programming with Constraints	12
2.2	Constraint Logic Programming	17
2.2.1	The CLP(R) Language	18
2.2.2	Applications	22
2.3	Preference Logic Programs	29
2.3.1	PLP Syntax	30
2.3.2	Examples	31
2.4	Summary	33
3	A Constrained Object Language	34
3.1	Syntax of Cob Programs	34
3.1.1	Attributes and Terms	35

3.1.2	Constraints and Predicates	37
3.2	Examples	42
3.2.1	Conditional Constraints (Date Object)	42
3.2.2	Constraints over Arrays (Heat Plate Object)	45
3.2.3	Constraints over Aggregated Objects (Simple Truss)	48
3.2.4	Constraints with Inheritance and Aggregation (DC Circuits)	51
3.3	Cob Modeling Environment	55
3.3.1	Domain Independent Interface	55
3.3.2	Domain-Specific Interfaces	60
3.4	Related Work	63
3.4.1	Constrained Objects Languages	63
3.4.2	Constraints, Logic and Objects	67
3.4.3	Constraint-based Specifications	71
3.5	Summary	72
4	Semantics of Constrained Objects	74
4.1	Declarative Semantics	75
4.1.1	Constraint Logic Programs	76
4.1.2	Translation of Constrained Objects to CLP	79
4.1.3	Constrained Object Programs	85
4.1.4	Cob Programs with Conditional Constraints	87
4.2	Operational Semantics	90
4.2.1	Constraint Logic Programs	90
4.2.2	Constrained Object Programs	93
4.2.3	Soundness and Completeness	95
4.2.4	Operational Semantics of General Conditional Constraints	98
4.3	Summary	100
5	Implementation of Constrained Objects	102
5.1	Cob Compiler	107
5.1.1	Translation of Cob to CLP	107

5.1.2	Classification of Predicates	115
5.1.3	Properties of Translated Programs	117
5.2	Partial Evaluation	119
5.2.1	Strategy	120
5.2.2	Optimization	125
5.2.3	Conditional and Non-linear Constraint Solving	129
5.2.4	Queries that can be Partially Evaluated	130
5.3	Interactive Execution	131
5.3.1	Interactive Solving	134
5.3.2	Interactive Debugging	135
5.4	Comparison with Related Work	136
5.5	Summary	137
6	Constrained Objects with Preferences	138
6.1	Cob Programs with Preferences	139
6.1.1	Syntax	139
6.1.2	Declarative Semantics	143
6.1.3	Operational Semantics	145
6.2	Relaxation in Cob	150
6.2.1	Syntax	152
6.2.2	Motivation	153
6.3	Operational Semantics	158
6.3.1	Naive Approach	158
6.3.2	Improved Approach	161
6.4	Related Work	165
7	Case Studies	167
7.1	Electric Circuits	168
7.1.1	Cob Model of RLC Circuits	168
7.1.2	Cob Diagrams of Electrical Circuits	172
7.1.3	Uses of Cob Model	176

7.2	Product Design: Gear Train	177
7.2.1	Structure of Gear Train	178
7.2.2	Cob Model of Gear Train	180
7.2.3	Uses of Cob Model	182
7.3	Documents as Constrained Objects	184
7.3.1	Structured Documents	184
7.3.2	Book as a Constrained Object	187
7.3.3	Creating and Formatting a Simple Book	193
8	Conclusions and Future Work	199
8.1	Summary and Contributions	199
8.2	Directions for Future Work	202
8.2.1	Language Design	202
8.2.2	Modeling Environment	203
8.2.3	Semantics	204
8.2.4	Interactive Execution	205
	Bibliography	206
A	Complete Syntax of Cob Programs	218
B	Compiling and Running Cob Programs	219
C	Sample Truss Code	224
D	Cob Model of Heatplate using Inheritance	226

List of Figures

2.1	Simple Electrical Circuit	27
3.1	A Heat Plate	46
3.2	A Simple Truss	49
3.3	Simple Electrical Circuit	54
3.4	Class Diagram for Circuit Example	56
3.5	Snapshot of CUML tool with Class Diagram for Circuit Example	57
3.6	Popup window for entering/editing constraints and other specifications of a Class	57
3.7	Popup Window through which the constructor of a class can be defined/edited.	58
3.8	Snapshot of CUML tool showing the Cob code generated from the Class Diagram of Figure 3.5.	59
3.9	Snapshot of the Truss Drawing Tool showing a Sample Truss	61
3.10	Popup Window for Entering Values for Attributes of a Beam	62
4.1	Algorithm for renaming variables	81
4.2	Cob Class Definitions	83
4.3	Translated CLP Program	84
5.1	Cob Computational Environment: <i>build</i> phase	103
5.2	Cob Execution Environment: <i>solve</i> phase	104
5.3	Cob Interactive Execution Environment	105
5.4	Cob Execution Environment: <i>query phase</i>	106
5.5	Cob class definitions	113
5.6	Translated CLP(R) code	114

5.7	Scheme for Partial Evaluation	122
5.8	Constraints obtained by partial evaluation of samplecircuit	124
5.9	Object Graph of Samplecircuit	132
6.1	A Separation Flow Sheet	141
6.2	Sample Graph.	157
6.3	Naive evaluation of a relaxation goal.	160
7.1	Snapshot of the Circuit Drawing Tool	173
7.2	Snapshot of the input window for a resistor in the circuit drawing tool . . .	174
7.3	Snapshot of the Circuit Drawing Tool showing input and output windows and details of resistor	175
7.4	Snapshot of the Analog Circuit Drawing Tool	176
7.5	Gear Train as Constrained Objects	181
7.6	Structure of a Simple Book	189
8.1	A Simple Truss	225

Abstract

This dissertation investigates the theory, design, implementation and application of a programming language and modeling environment based on the concepts of objects, constraints and visualization. We focus on modeling complex systems that can be described as an assembly of interconnected, interdependent components whose characteristics may be governed by laws, or constraints. Often, a natural visual representation can be associated with each of the components and their assembly. Such complex systems occur in a wide variety of domains such as engineering, biological sciences, ecology, etc. Modeling such systems therefore involves the specification of their structure and behavior, and also their visualization. In modeling structure, we can think of each component as an *object*, with internal attributes that capture the relevant features. In modeling behavioral laws or invariants, it is natural to express them as *constraints* over the attributes of an object. When such objects are aggregated to form a complex object, their internal attributes might further have to satisfy interface constraints. The resultant state of a complex object is deduced by satisfying the internal and interface constraints of the constituent objects. This paradigm is referred to as *constrained objects*.

We describe a principled approach to the design of a constrained object programming language built on rigorous semantic foundations. Our proposed language, Cob, provides a rich set of modeling features, including object-oriented concepts such as classes, inheritance, aggregation and polymorphism, and also declarative constraints, such as symbolic, arithmetic equations and inequalities, quantified and conditional constraints, and disequations. We define the semantics of constrained objects based upon a translation to constraint logic programs (CLP). A Cob class is translated to a CLP predicate and the set-theoretic semantics of the class are given in terms of the least model of the corresponding predicate. Such semantics also pave the way for a novel implementation of constrained objects.

However, due to the limitations of CLP, such an implementation cannot handle conditional constraints and may not give satisfactory performance for large-scale models. To overcome these limitations, we have developed partial evaluation techniques for generating

optimized CLP code. These techniques also allow us to use our own handler for evaluating conditional constraints, and employ a more powerful existing solver for handling complex non-linear constraints. Based on partial evaluation, we have developed novel techniques for interactive execution of Cob models and fault detection in over-constrained structures.

Often a constrained object model of a complex system may have more than one solution. We allow the modeler to state preferences within a Cob class definition and the resultant state of the system is obtained by constraint solving and optimization. The semantics of Cob programs with preferences are based upon a translation to preference logic programs (PLP). We also investigate different forms of relaxation of preferences to obtain suboptimal solutions and propose a scheme for the operational semantics of relaxation that accounts for recursively defined PLP predicates.

We give several examples from different domains to illustrate that the paradigm of constrained objects provides a principled approach to modeling complex systems and is amenable to efficient and interactive execution.

Chapter 1

Introduction

1.1 Motivation and Significance

This dissertation investigates the theory, design, implementation and application of a programming language and modeling environment based on the concepts of objects, constraints, and visualization. Our focus is on modeling systems that consist of an assembly of interconnected, interdependent components that have the following characteristics:

1. They are compositional in nature, i.e., a complex structure is made of smaller structures or components which are further composed of smaller components and so on.
2. The behavior of a component by itself and in relation to other components is generally governed by some laws or rules.
3. There is a natural visual representation that can be associated with each component.

Examples of such systems occur in different domains: engineering, organizational enterprises, biological systems, ecosystems, business processes, information warehouses, and program execution. One of the main goals of our research is to provide a modeling environment that facilitates a principled approach to modeling such complex systems. Such an environment should facilitate intuitive creation and manipulation of models that are easy to understand, modify and debug. We also require that, where appropriate, a model should be given a visual representation. Thus the modeling environment for complex systems should

facilitate compositional specification of structure, declarative specification of behavior, and visual development and manipulation of models (where appropriate).

In modeling structure, it is natural to adopt a compositional approach since a complex engineering entity is typically an assembly of many components. In programming language terms, we may model each component as an *object*, with internal attributes that capture the relevant features that are of interest to the model. The concepts of classes, hierarchies and aggregation found in object-oriented (OO) languages, such as C++, Java, and Smalltalk, are appropriate to model the categories of components and their assembly. However, in modeling the behavior of complex engineering entities, the traditional OO approach of using procedures (or methods) is inappropriate because it is more natural to think of each component as being governed by certain laws, or invariants. Using methods to represent behavioral laws places the responsibility of enforcing them on the programmer. Moreover, these laws become implicit in the procedural code, instead of being explicitly declared.

From a programming language standpoint, it is more natural to express behavioral laws as constraints. A *constraint* is a declarative specification of a relation between variables/attributes. Constraint programming languages are declarative in that the programmer specifies *what* constraints define the problem without specifying *how* to solve them. The constraints and their solving techniques are appropriate to model, test and enforce (solve) the behavioral constraints of a system. Constraints thus facilitate a declarative specification of the behavior of a complex system. However, a pure constraint programming language does not provide an adequate representation for the structure of a system. It lacks the modularity of object-oriented languages, and a complex system appears as a large collection of constraints with little correspondence to the structure of the system being modeled.

A more direct and intuitive approach to modeling is through a visual representation of the system in the form of two or three dimensional drawings. A visual representation can be easier to understand since it is less abstract and bears close physical resemblance to the real system. For example, engineering drawing tools provide a convenient way for a modeler to visualize and design a structure. Such visualization of a structure though proportional and drawn to scale, is however a purely geometric representation. It captures the geometric attributes of the structure and but does not have any correlation to the underlying semantics.

For example, one may draw a sketch of a bridge, but there is no way to infer its load bearing capacity etc. from the drawing.

Thus, while the paradigms of objects, constraints, and visualization each offer features suitable for modeling certain aspects of a complex system, they are individually inappropriate for modeling both the structural as well as behavioral aspects of the system. In this dissertation we investigate the theory, design and implementation of a programming language and modeling environment based on the notion of *constrained objects*. This approach combines the modular aspects of objects with the declarative nature of constraints, and, together with visualization, provides a powerful modeling environment for complex systems.

1.2 Objectives and Results

1.2.1 Technical Approach

A constrained object is an object whose attributes are governed by laws or declarative constraints. When such objects are aggregated to form a complex object, their internal attributes might further have to satisfy interface constraints. In general, the resultant state of a complex object can be deduced only by satisfying both the internal and the interface constraints of the constituent objects. This paradigm of objects and constraints is referred to as *constrained objects*, and it may be regarded as a declarative approach to object-oriented programming.

To illustrate the notion of constrained objects, consider a resistor in an electrical circuit. Its state may be represented by three variables V , I , and R , which represent respectively its voltage, current, and resistance. However, these state variables may not change independently, but are governed by the constraint $V = I * R$. Hence, a resistor is a *constrained object*. When two or more constrained objects are aggregated to form a complex object, their internal states may be subject to one or more interface constraints. For example, if two resistor objects are connected in series, their respective currents should be made equal. Similarly, in the civil engineering domain, we can model the members and joints in a truss as objects and we can express the laws of equilibrium as constraints over the various forces

acting on the truss. In the chemical engineering domain, constrained objects can be used to model mixers and separators, and constraints can be written to express the law of mass balance.

In the paradigm of constrained objects that we present, a complex object may also have a visual representation, such as a diagram. This visual representation is compositional in nature; that is, each component of the visual form can be traced back to some specific object in the underlying constrained object model. An end-user, i.e., modeler, can access and modify the underlying model using the visual representation. This capability may be contrasted with currently available tools for engineering design such as AutoCAD, Abaqus, etc., where the visual representation contains only geometric information but does not provide access to the underlying logic or constraints. We expect to have different visual interfaces for different domains but a common textual language (described in Section 3) for defining the classes of constrained objects.

We expect a modeler to first define the classes of constrained objects, by specifying their attributes and their internal and interface constraints. These classes may be organized into an inheritance hierarchy. Once these definitions have been completed, the modeler can *build* a specific complex object, and execute (*solve*) it to observe its behavior. This execution will involve a process of logical inference and constraint satisfaction [32, 33, 40, 49, 59]. A modeler will then need to go through one or more iterations of *modifying* the component objects followed by re-execution. Such modifications could involve updating the internal states of the constituent objects, as well as adding new objects, replacing existing objects, etc. The complex object can be queried to find the values of attributes that will satisfy some given constraints in addition to the ones already present in the constrained objects.

1.2.2 Scope of the Dissertation

We describe below the major areas of investigation along with a brief summary of our contributions within these areas.

Language Design. We develop a novel programming language, Cob, based on the notion of constrained objects. The language facilitates declarative specification of a wide variety of constraints including arithmetic and symbolic. We provide quantified constraints for compact specification of a relation whose participants may range over enumerations, i.e., indices of an array or the elements of an explicitly specified set. Similarly, we also provide a notation for iterative terms appearing in arithmetic constraints. Such a declarative syntax brings the constraint specification closer to its mathematical equivalent. We also provide conditional constraints, which are a powerful and expressive means of stating different constraints depending on the state of a constrained object. We also allow user-defined predicates within a constrained object class definition and give several examples illustrating the use of these constructs and the application of Cob to modeling problems from various domains.

Modeling Environment. We provide two types of visual interfaces for the creation of constrained object models, their modification and execution. A domain independent visual interface is provided for authoring constrained object class diagrams. The definition of a constrained object class (i.e., its attributes, constraints, constructors etc.) and its relation (e.g. inheritance, aggregation, etc.) to other classes can be specified through a graphical user interface. The interface can be used to run (solve) the constrained object model and query it for values of attributes. We also provide different domain dependent visual interfaces for drawing constrained object models of engineering structures such as electrical circuits and trusses. These diagrams of engineering structures can themselves be thought of as programs since they can be created, modified and executed through this visual interface which also displays answers (values of attributes) to queries.

Formal Semantics. We define formal declarative and operational semantics for constrained objects. The formal semantics of a programming language provide a mathematical model for its programs and can be used for its analysis e.g. computability, verification, etc. A constrained object can be understood as an abstract datatype whose characteristics are described axiomatically through constraints. Informally, the meaning of a constrained object is the set of values for its attributes that satisfy its constraints. We define a set-theoretic

semantics of constrained objects that is based on a mapping between constrained objects and constraint logic predicates [59, 60]. The operational semantics are given as rewrite rules and serve as a basis for an implementation of the constrained object paradigm.

Execution Environment. We provide an implementation for constrained objects based on a translation to the constraint logic programming language CLP(R). A class is mapped to a CLP(R) predicate such that the constraints of the class form the body of the predicate. In this way the predicate serves as a procedure that enforces the constraints. However, due to the limitations of CLP(R), such a translation alone cannot satisfactorily handle conditional and non-linear constraints and may also be inefficient for large models. Therefore, we have developed techniques for partial evaluation of constrained object programs that generate optimized code and facilitate handling of conditional constraints, and deployment of more powerful constraint solvers for non-linear systems of constraints. Partial evaluation also facilitates efficient re-solving when a model is modified. Using partial evaluation, we have developed techniques for visual interactive execution and debugging of a constrained object program based on its object structure. For a given instance of a complex constrained object, its object structure represents the relation between the instances of its components.

Preferences and Relaxation. When a system of constraints has more than one solution, there may be a preference for one solution over another. For such cases, we extend the syntax and semantics of constrained objects to allow the specification of constraint optimization problems. Two forms of optimization are provided: maximization (or minimization) of an arithmetic function and the more general form of a preference clause [38]. The semantics of constrained object programs with preferences are based on a mapping to preference logic programs (PLP) [38]. When preferences are specified in a constrained object class definition, we determine the set-theoretic semantics of the class from the set of *preferential consequences* of the corresponding predicate.

Often, one may be interested in the optimal as well as suboptimal solutions to an optimization problem. This requires a relaxation of the preference criteria and we explore different forms of relaxation: with respect to the underlying preference criterion or with respect to an extra constraint. We address the computational problems arising from relaxation

and provide an intuitive operational semantics for relaxation goals in PLP.

Applications. We present several examples throughout the dissertation to illustrate that the paradigm of constrained object is appropriate for modeling complex systems from various domains. We present three case studies that describe the constrained object methodology for modeling complex systems which gives a compositional specification of the structure and declarative specification of the behavior of the system. The ability to specify, run, query, and debug partial models, textually as well as through graphical user interfaces, makes the constrained object paradigm useful in general and also for pedagogical purposes in the engineering domain.

This dissertation presents a novel programming language and modeling environment called **Cob** based on the notion of *constrained objects* and their visualization that facilitates a principled approach to the modeling of complex systems.

1.3 Summary of Dissertation

The remainder of this dissertation is organized as follows. Chapter 2 gives an overview of constraints and constraint programming techniques. The paradigms of constraint (logic) programming and preference logic programming are described in detail along with several examples. Chapter 3 gives the syntax of the constrained object programming language named **Cob**, and describes its modeling environment. Several examples are given to illustrate the use and application of the various constructs of this language and environment. Chapter 3 also gives a comparison with related work. Chapter 4 describes formal declarative and operational semantics of constrained objects. Chapter 5 discusses implementation techniques for constrained objects. This includes the translation to CLP(R) and a scheme for partial evaluation. Chapter 6 gives a description of the extension of the constrained object paradigm to handle optimization problems and discusses the computational issues of relaxation in preference logic programs. Chapter 7 presents case studies illustrating the development and application of constrained object models in different domains. Chapter 8 gives conclusions and our current and future research plans.

Chapter 2

Background

This chapter presents an overview of some basic concepts and computational techniques relating to constraints and constraint programming languages. Two constraint programming paradigms are discussed: constraint logic programming and preference logic programming. The work on constrained objects presented in this dissertation is closely related to these two programming paradigms. The overview presented in this chapter provides the necessary background for understanding the rest of the dissertation.

2.1 Constraint Programming

The overview of constraint programming presented in this section is summarized from [81, 8, 9, 70] and from [51] which gives a comprehensive survey of constraint programming.

2.1.1 Constraint

A constraint is a *relation* among variables and serves to specify the space of possible values for these variables. For example, the equation $X + Y > 0$ is a constraint that restricts the values of variables X and Y to add up to a positive number. A constraint may involve arithmetic, relational, boolean and set operations. Following are the main characteristics of constraints.

- A constraint is a *declarative* specification of a relation between variables, but does not specify any algorithm or procedure to enforce the relation.

- Constraints are *non-directional*, i.e., they do not have input/output variables. The same constraint $X = Y + Z$ may be used to compute both the constraints on X given the constraints on Y and Z as well as the constraints on Z given the constraints on X and Y .
- Due to their declarative and non-directional nature, the order of constraints does not matter.
- A constraint may give *partial information* relating a variable to other variables or values, or it may specify its exact value(s).
- Constraints are *additive*, i.e., a constraint may be added to an existing store of constraints, and the conjunction of all existing constraints is required to hold simultaneously. It is possible that the addition of a constraint makes the constraint store inconsistent. This is referred to as an over-constrained system and is discussed later in Section 2.1.3.
- A constraint domain refers to the data domain of each variable and the operations and relations defined on these domains. Different variables can have different domains.

Problems in the areas of engineering modeling, operations research, combinatorics, databases, information retrieval, user interface construction and document formatting are often characterized by relations such as: laws of physics (e.g. balance of forces, conservation of mass and energy), functional specifications (e.g. input-output torque, power), performance requirements (e.g. efficiency, speed), scheduling restrictions (e.g. availability, feasibility, preference), data integrity (e.g. an employee's salary should be consistent with his grade), etc. A declarative specification of constraints facilitates a natural and intuitive description of such problems.

2.1.2 Constraint Satisfaction

A **constraint satisfaction problem (CSP)** consists of a finite set of variables, a domain (of values) for each variable, and a finite set of constraints involving these variables. A

solution to a CSP is a *valuation*, or mapping, of its variables to values in their respective domains that satisfy all the constraints of the CSP.

The goal of a constraint satisfaction algorithm is to determine whether a solution exists, and to find one or more or all solutions to the constraints. In the case of multiple solutions, the goal might be to find an optimal solution with respect to a given objective (cost) function. Constraint satisfaction algorithms can be broadly divided into two categories based on the domains of the variables.

Finite Domain CSP. Satisfaction techniques for CSP over *finite domains* are combinatorial in nature and we describe some of these techniques here. The following description is summarized from [51, 8, 9] and [70] which gives a comprehensive survey of constraint satisfaction algorithms for finite domains. Algorithms for finite domain CSP include those based on search, inconsistency detection or stochastic methods. A simplistic search algorithm that can solve any finite domain CSP is the generate-and-test (GT) scheme. This scheme generates combination of values for the variables, and tests the constraints for these values. This scheme is inefficient since it may generate every possible combination. A relatively efficient search technique uses backtracking (BT). Here, variables are sequentially instantiated with values from their domain. Each time a variable is instantiated the pertinent constraints are tested. If a constraint fails, the system backtracks to pick an alternate value for the most recently instantiated variable having alternatives. Thus, when a constraint fails, this scheme prevents the generation of a subset of combination of values that are guaranteed to fail. Although more efficient than GT, for most non-trivial problems, backtracking takes exponential time.

Constraint satisfaction algorithms based on inconsistency detection use the notion of a *constraint graph* to reduce backtracking while searching for a solution. The nodes of this graph represent distinct variables and there is an edge between two variables if they appear in a single constraint. A unary constraint is represented as an edge from a node to itself. Since any finite domain CSP can be mapped to an equivalent CSP in which every constraint has at most two variables [94], it suffices to discuss only binary CSP. Some basic consistency based algorithms use: (i) node consistency: remove those values from a variable's domain that violate any unary constraint involving the variable; (ii) arc

consistency: remove those values from a variable's domain that violate a constraint on that variable; (iii) path consistency: remove pairs of values that satisfy a constraint (edge) but do not satisfy a series of constraints that form a path between the variables. Neither node nor arc consistency for a set of constraints however, implies their solvability.

A stronger notion of *k-consistency*, based on path consistency, has been defined which ensures that any partial solution/instantiation of $k - 1$ variables which does not violate any constraints can be extended by the instantiation of any k^{th} variable such that the constraints are still not violated. A constraint graph is *strongly k-consistent* if it is *j-consistent* for all $j \leq k$ [70]. Node consistency is equivalent to strong 1-consistency; arc consistency is equivalent to strong 2-consistency; and path consistency is equivalent to strong 3-consistency.

Algorithms based on k-consistency take a partial solution and try to extend it to a full solution. If a constraint graph with n variables is strongly n -consistent then it is solvable without any backtracking. But making an n -node constraint graph strongly n -consistent can take exponential time in the worst case. Hence, more practical approaches try to make the constraint graph strongly k -consistent for $k < n$. The aim of these algorithms is to make the search for a solution backtrack-free. These algorithms combine simple backtracking with arc consistency, i.e., in every step of the backtracking algorithm, they try to make the graph arc-consistent. Every time a variable is instantiated, consistency check is performed on uninstantiated variables. There are different algorithms depending upon the extent of this checking: forward checking (FC), partial lookahead (PL), full lookahead (FL) and really full lookahead (RFL). These algorithms are called constraint propagation algorithms since they propagate information about one constraint to another.

We have discussed only some of the interesting techniques for finite domain CSP above. Other techniques include stochastic methods [86], algorithms that exploit the structure of the problem [21], and identifying tractable classes of problems [31, 78, 22].

Non-finite Domain CSP. The solving techniques for CSP in which the variables may have an *infinite domain* are usually mathematical or numeric-based, e.g., the Gauss-Jordan elimination method, Newton method, Taylor series, Kachian and Karmarkar, Simplex algorithm or automatic differentiation. Constraint solving techniques for infinite domain are often specialized for a particular domain (e.g. real or rational numbers) as well as cate-

gory of constraints (e.g. linear, non-linear or differential equalities or inequalities). The generality of the solving algorithm is a tradeoff with its efficiency.

Typically, a constraint solving system keeps the constraints in a standard canonical form ,e.g., $X > 0$, $X \geq 0$ or $\sum_{i=1}^n c_i X_i = d$. Equations are maintained in parametric solved form such as $X_i = b_i + c_{i1} * T_{i1} + \dots + c_{in} * T_{in}$, where X_i is a non-parametric variable and T_{ij} s are parametric variables [62]. Common steps in a basic algorithm for solving linear equality constraints such as Gaussian elimination involve substitution of parametric variables by equivalent parametric expressions. When sufficient information about (parametric) variables is obtained, the values of the non-parametric variables can be computed. Inequality solvers may also maintain constraints in a solved form and determine implied equalities from the set of inequalities. When used in a constraint programming system such algorithms are adapted to be incremental, performing substitution and pivoting operations to get back the canonical form after constraints are added. Some constraint solving techniques for linear equality and inequality constraints are described in [63, 75, 48, 56].

2.1.3 Programming with Constraints

Computing with constraints involves the specification, generation, manipulation, testing and solving of constraints. A computational system based on constraints provides a syntax for specifying, combining and manipulating constraints and its underlying computational engine provides mechanisms for processing, testing and solving constraints. The computational engine may use inference, backtracking, search, constraint propagation, entailment and/or optimization techniques for constraint satisfaction.

A majority of constraint programming (CP) systems are based on the syntax and control mechanism of logic programming, e.g., constraint logic programming [59, 61] and concurrent constraint programming [99]. Other CP systems combine constraints with imperative programming, functional programming and term rewriting. We give a brief overview of these major categories of constraint programming languages which are described in [81].

Constraint Logic Programming. Constraint logic programming (CLP) merges constraints into the framework of logic programming. A CLP system is characterized by its constraint

domain and the solver for that domain. Unification between terms is generalized to constraint solving over the data domain. CLP languages can be used to model a wide variety of problems including arithmetic, scheduling, as well as combinatorial problems and constraints can be used to prune the space of solutions. There are numerous constraint logic programming languages for modeling problems and solving constraints over different domains. For example, the CHIP [24] language provides boolean, linear constraints and finite domain constraints. The clp(FD) solver [16] provides finite domain constraints and is used for discrete optimization and verification problems like scheduling, planning, etc. The CLP(R) [58] language provides constraints over real numbers but solves only linear constraints. Our work on constrained objects is directly related to the CLP paradigm. In Section 2.2, we describe the CLP paradigm and the CLP(R) language in some detail.

Concurrent Constraint Programming. Concurrent constraint programming merges constraints into the framework of concurrent logic programming [100]. A rule defines a “process” and comprises of guard conditions, or constraints, and a body which is a sequence of literals. A rule can be used to resolve a goal if its guard conditions are enabled. The guard conditions can be of two kinds: *ask* and *tell*. The former checks for entailment while the latter (checks for compatibility and) adds the constraint to the current constraint store. The evaluation mechanism CCP programs differs from CLP programs in that if more than one rule can be used to reduce the goal, then any one is picked randomly. The evaluation scheme does not consider any other rules (enabled or not enabled) at any later stage. Concurrent constraint programming is used for modeling reactive systems, e.g. incremental constraint solvers. Some concurrent constraint programming languages are cc(fd) [50], Oz [105], AKL [65], CIAO [52].

Constraint Handling Rules. Constraint handling rules (CHR) [35, 36] are a high-level language designed specifically for writing constraint solvers. They consist of multi-headed guarded rules which are used to rewrite a constraint store (a set of constraints). Constraints that match the left hand side of a rule can be replaced by the constraints on the right hand side of the rule if the guard is implied by the constraint store. Applying a CHR can result in simplification of constraints or propagation. Simplification replaces constraints with

simpler equivalent constraints while propagation adds a new redundant constraint that may cause further simplification. The constraint store is manipulated by repeatedly applying the constraint handling rules until further application does not change the constraint store. CHRs can be used to build new constraint solvers, extend existing ones with new constraints, specialize constraint solvers for a particular domain or application or combine constraint solvers. CHRs can be combined with a high-level host language (e.g. CLP) to define user-defined constraints and constraint solvers.

Constraints + Object-Object Programming. Languages such as Kaleidoscope [76, 75], Siri [55], ThingLab [10], Modelica [34] integrate constraint into an object-oriented framework. In ThingLab, each constraint is accompanied by methods for solving it and the system arrives at a constraint satisfaction plan using the appropriate method. Siri simulates imperative constructs through “constraint patterns” [54]. In Modelica there is a clear separation between constraints and imperative constructs which can both exist in a class definition. Kaleidoscope simplifies user-defined constraints until a primitive constraint solver can be used to solve them. We give a detailed comparison of our work with these languages in Section 3.4.

Constraints + Term Rewriting. Term rewriting is a declarative paradigm that uses rewrite rules along with pattern matching to simplify terms. If an expression matches the left hand side of a rewrite rule, it can be replaced by the expression on the right hand side of the rule. For example, one can define rewrite rules to express any Boolean formula in conjunctive normal form. Augmented term rewriting in Bertrand [72] combines constraints with term rewriting. Augmented term rewriting extends term rewriting by providing substitution of a variable by an expression and the introduction of new local variables on the right hand side of a rule. In an augmented term rewriting system, there is no global constraint store or solver, since the rewrite rules themselves are used to define constraint solving. Another approach to combining constraints with term rewriting provides conditional rules or rules with guard conditions: rules that are applicable only when their constraint or guard condition is true. Such systems are typically used to implement automatic theorem provers [57, 6] and are considered closer to CCP than to CLP [81].

Constraints + Functional Programming. There have been three major approaches to combining constraints with functional programming [81]. A simple approach is to treat constraints as functions that return answers. However, these are not constraints in the declarative sense since the programmer must provide the functions for solving the constraints. Another approach is to embed functions in a constraint language like CLP. Although built-in functions such as $+$, $-$, etc. are provided in CLP, the system should also allow user-defined functions. Treating functions as relations with an extra argument representing the result is not sufficient since the system does not have enough knowledge to reason about such functions when they appear in constraints. By knowledge, we mean properties like associativity, commutativity, etc. Also multi-directional evaluation of functions might not always be possible. A third approach treats functions and constraints to be at the same level and extends lambda calculus with a global constraint store. One such approach called *constrained lambda calculus* [79] uses the constraint store to determine value of variables. Another approach in the language Oz [105] (which has constraint solvers for tree constraints and finite domain constraints) associates guard constraints or conditions with lambda expressions and variables may range over lambda expressions (functions).

Constraint Solving Toolkits and Symbolic Algebra Packages. Constraint solving toolkits such as JSolver (Java solver), ILOG solver [91] or 2LP [5] embed a solver into an imperative host language. The programmer can access the solver only through an interface. Such toolkits are usually specialized for a particular domain of problems. Symbolic algebra packages such as Matlab [109], Maple [112], MACSYMA [43], Mathematica [115] are mathematical languages typically used by scientists and mathematicians to solve simultaneous linear, non-linear or differential, equations, inequalities, etc. These are powerful tools that can solve or simplify a system of mathematical constraints. Some of them have some programming facilities but little by way of controlling the solving.

Over-Constrained Systems. A system of unsatisfiable constraints is called an over-constrained system. The study of over-constrained systems includes developing techniques for detecting an over-constrained system, locating the cause of the error and relaxing some constraints in order to get a solution. Over-constrained systems are less likely to occur when a

constraint may be tagged as a “soft” constraint [113], i.e., it is not required to be satisfied. One approach to specifying a soft constraint is to associate a strength or weight with each constraint and to specify an error function which associates an error with a constraint and a valuation and can be used to compare alternate valuations [113]. Intelligent backtracking in CLP detects independent subgoals in a query. These goals can be run in parallel [7] because an error cannot be caused by their interaction. Among other reasons, the study of over-constrained systems is motivated by problems of belief revision in artificial intelligence, where a new belief may contradict an existing belief set [37, 83]. A technique for detecting a contradictory set of constraints is given in [18]. This technique is for linear equalities and inequalities, and it is based on introducing an extra error variables into constraints and then solving to minimize their sum. The constraint whose error variable gets a non-zero value is removed because it is considered unsatisfiable and the process is redone. In the end a minimal set of unsatisfiable constraints is obtained. A collection of articles on over-constrained systems appears in [64].

Constraint Optimization. It is often the case that a set of constraints has more than one solution. In such cases there may be a criterion based on which one solution can be declared to be *better* than another. This criterion is referred to as the optimization or preference criterion. The problem then is to determine the optimal solutions from a set of feasible solutions based on this criterion. The preference criterion may be used to rank multiple solutions in decreasing order of preference and the top ranking solutions are optimal. In general, however, the preference criterion may not be able to place all the solutions in a total order.

The most well-known optimization technique is linear programming. A set of linear constraints over some variables is given which may include the range of the variables. Also given is a mathematical function involving some or all of these variables. This function is called the objective function, and the goal is to find a solution to the constraints that minimizes or maximizes the objective function as the case may be.

Another way to state an optimization problem is to specify a set of constraints and indicate which ones must be satisfied and which ones may be violated if it is not possible to satisfy them all. The hierarchical constraint logic programming (HCLP) language [11, 113]

incorporates the notion of constraint hierarchies into the CLP paradigm. Constraints can be tagged `required`, `strong`, `medium`, `weak`, etc., and the computational engine tries to first find a solution that satisfies all the required constraints, and then as many of the strong constraints, and subsequently as many of the medium constraints, and so on as possible. A more general language for expressing constraint hierarchies and constraint optimization within the realm of constraint logic programming is the preference logic programming (PLP) paradigm [38]. We discuss it in more detail in section 2.3.

2.2 Constraint Logic Programming

The Constraint logic programming (CLP) [59, 61] scheme is a merger of two declarative paradigms: logic programming and constraint programming. Logic programming emerged from research in resolution theorem proving and artificial intelligence. A basic logic program is a set of Horn clauses (a subset of first order logic) and the goal or query to be proved is also a special kind of Horn clause (headless). The observation that logic programming is basically a kind of constraint programming where the constraints are equalities over terms and constraint-solving is term-unification led to the development of constraint logic programming. CLP extends logic programming with constraint solving capabilities and generalizes unification to solvability over the constraint domain.

At the core of a CLP scheme lies a structure. Informally, a structure is a domain comprising a set of values along with functions and relations on this set. The relations can be thought of as constraints. The structure must be *solution compact*, i.e., it should be possible to describe every value in the domain by a conjunction of a (possibly infinite) set of constraints and the complement of every constraint should be definable as a disjunction of a (possibly infinite) set of constraints [61]. Given the CLP scheme and a constraint domain, we get an instance of a CLP language. For example, `clp(FD)` represents constraint logic programming over finite domain constraints; `CLP(R)` is the constraint logic programming language over the real number domain; `clp(B)` is a constraint logic language with boolean constraints. `CLP(Term)`, where `Term` is the traditional logic terms with unification, represents the Prolog-like logic programming paradigm. Our paradigm of constrained

objects bears close relations with the CLP(R) scheme and hence we give an overview of the CLP(R) language next.

2.2.1 The CLP(R) Language

The syntax and computational model of the CLP(R) language bear close resemblance to those of traditional logic programming languages such as Prolog. We describe CLP(R) by highlighting the generalizations it makes to logic programming. This description summarizes a more detailed discussion given in [44, 63].

Syntax. We describe the syntax of CLP(R) terms, constraints and rules.

- *Term.* A term can be a variable, constant, or an arithmetic expression involving variables, real number constants or functions such as `sin`, `cos`, etc. As in logic programming, terms are built from constants and uninterpreted functors, i.e., if t_1, \dots, t_n are terms and if f is an uninterpreted functor, then $f(t_1, \dots, t_n)$ is a term.
- *Explicit Constraint.* The relational operators ($=, >, <, \geq, \leq$) over the real number domain are used to form explicit constraints. Note that explicit constraints cannot involve uninterpreted constants or functors. Equality between uninterpreted functor terms is treated as unification.
- *Goal.* A goal is similar to a Prolog goal except that it can have explicit constraints. Thus, a goal is a sequence of literals and explicit constraints. A literal is a positive or a negative atom or explicit constraint. If p is an n -ary predicate symbol and t_1, \dots, t_n are terms then $p(t_1, \dots, t_n)$ is an atom [73].
- *Rule.* A rule is similar to a Prolog rule except that the atoms in the body of a rule can be explicit constraints. Thus, a CLP(R) rule is of the form $A : - B$ where A is an atom and B is a goal. A fact is a rule with an empty goal as the body.

A CLP(R) program is a sequence of rules. A CLP(R) query is a goal and it is evaluated with respect to a CLP(R) program. We describe goal evaluation next.

Goal Evaluation. As mentioned earlier, unifiability amongst terms in logic programming is generalized to solvability of constraints over \mathbb{R} , the real number domain, in CLP(\mathbb{R}). The evaluation of a goal with respect to a CLP(\mathbb{R}) program works along the lines of goal resolution in a logic programming context. We provide an informal description here; a formal description of a CLP(\mathbb{R}) program, a query and resolution with the declarative and operational semantics is given in Section 4.1.

The evaluation of a goal G_0 with respect to a CLP(\mathbb{R}) program P can be described in terms of a *derivation* or a finite sequence of states S_0, \dots, S_n . A *state* S_i is represented as $\langle G_i \parallel C_i \rangle$ where G_i is a goal and C_i represents a collection of constraints or a *constraint store*.

- *Solution.* A set of constraints is solvable if there exists a mapping from their variables to values in \mathcal{R} such that the constraints evaluate to true. The mapping is called a solution to the constraints.
- *Constraint Solver.* A constraint solver tests whether a given set of constraints can be solved. It may also find a solution to the constraints and/or simplify them. Typically, a constraint solver is built for a specific domain. In the CLP(\mathbb{R}) context, the constraint solver is used to solve the conjunction of arithmetic constraints present in the constraint store over the real number domain. Typically such a constraint solver will perform *incremental constraint solving* since the constraint store is expected to change (add or delete constraints). The properties of the domain (in this case the real number domain \mathbb{R}) are built into the constraint solver so that it does not have to perform generate-and-test in order to solve/test constraints. For example, the CLP(\mathbb{R}) constraint solver can deduce that the conjunction of the constraints $X > Y$ and $X < Y$ is false without testing the constraints for any particular values. In other words, the solver has the ability to reason about one or more constraints using its knowledge about the domain and the built-in operations on this domain. The solver uses an adaptation of the Two-Phase Simplex algorithm and [103] for deciding linear inequalities and Gaussian Elimination for linear equalities.
- *Derivation.* Given a query G_0 and a CLP(\mathbb{R}) program P , a derivation of G_0 with

respect to P is a sequence of states $S_0 \equiv \langle G_0 \parallel true \rangle, S_1, S_2, \dots$ with the following properties.

- *Start State.* The start state (S_0) of a derivation consists of the goal G_0 and an empty constraint store.
- *Derivation Step.* A state $S_{i+1} \equiv \langle G_{i+1} \parallel C_{i+1} \rangle$ is *derived* from $S_i \equiv \langle G_i \parallel C_i \rangle$ if the following holds. Let $G_i \equiv A_1, \dots, A_m$ and suppose A_j ($j \in 1..m$) is selected for resolving. There are three cases:
 1. A_j is an atom: Suppose it is of the form $p(t_1, \dots, t_n)$. If there is a rule in P of the form $p(t'_1, \dots, t'_n) :- B$ then $G_{i+1} \equiv A_1, \dots, A_{j-1}, t_1 = t'_1, \dots, t_n = t'_n, B, A_{j+1}, \dots, A_m$ and $C_{i+1} \equiv C_i$. Note that before this step is applied, the variable of the pertinent rule of P are renamed to remove any common variable names between the goal and the rule. If there does not exist such a rule in P , then $S_{i+1} \equiv \langle \phi \parallel false \rangle$.
 2. A_j is an explicit constraint: It is added to the constraint store and hence $G_{i+1} \equiv A_1, \dots, A_{j-1}, A_{j+1}, \dots, A_m$ and $C_{i+1} \equiv C_i \cup \{A_j\}$.
 3. A_j is a unification constraint (term equality): Such equations are between non-arithmetic terms and may or may not involve functors. This is similar to the term equality in standard logic programming and is handled by a unification algorithm. Note that this may give rise to explicit (equality) constraints if arithmetic subterms are present. Equality between an arithmetic and non-arithmetic term results in the state $S_{i+1} \equiv \langle \phi \parallel false \rangle$.
- At every step of the derivation, the constraint store is checked for consistency. If $S_i \equiv \langle G_i \parallel C_i \rangle$ and $solv(C_i) = false$, i.e., the constraint store is inconsistent, then $S_{i+1} \equiv \langle \phi \parallel false \rangle$. If the constraint store is consistent, then the derivation proceeds to the next step.
- *Successful Derivation.* A derivation is said to be successful if it is finite and its final state consists of an empty goal and a consistent constraint store. There can be more than one successful derivation.

- *Answer Constraint.* The concept of answer substitution in logic programming is generalized to an answer constraint in CLP(R). The constraints in the constraint store at the end of a successful derivation form the solution to the goal. Specifically, the restriction of these constraints to those containing variables from the initial goal are the *answer constraints*, or the *computed answer*. Multiple successful derivations could give rise to multiple computed answers. Every instance of the answer constraint is a solution.
- *Finitely Failed.* If all the derivations for a goal are finite and end in failure, then the goal is said to have *finitely failed*, i.e., there is no computed solution.

CLP(R) System. We now give some practical details of a CLP(R) system [58].

- The literal selection strategy in the basic CLP computational model is impartial, making solutions independent of goal ordering. However, a practical implementation invariably deviates from this strategy in the following ways:
 1. The atoms in a goal are selected in a left-to-right order. A programmer can write more efficient programs by placing subgoals that test constraint satisfaction before those that generate solutions. We explain this point later with an example.
 2. The goal selection between a parent subgoal and child subgoal follows a depth first order, and a clause from a program is selected for matching in a top-to-bottom order. These strategies may in general result in loss of completeness. But this tradeoff is usually made in logic programming systems.
 3. Constraint solvability of non-linear constraints is not tested. A delay mechanism (explained later) is employed for handling non-linear constraints.
- *Backtracking.* When a derivation fails, the CLP(R) engine backtracks to find an alternate rule to match the most recently resolved atom. As the system backtracks, it undoes all the variable bindings and changes (addition of constraints) in the constraint store made after the most recent choice point. This backtracking mechanism is similar to that of traditional logic programming systems.

- *Delay Constraints.* The constraint solver for CLP(R) does not attempt to solve non-linear constraints. It delays their solving until they become sufficiently linear, i.e., until a sufficient number of their variables become ground. For example, solving of the constraint $\sin(X) = Y$ is delayed until either X or Y becomes ground. The constraint $\text{pow}(X, Y) = Z$ ($X^Y = Z$) is not tested for solvability by the constraint solver until at least two of X , Y or Z become ground or X becomes 0 or Y becomes 0 or Z becomes 1 or X becomes 1.

Hence, if an answer constraint contains non-linear constraints, then it actually represents a solution only if the non-linear constraints can be satisfied.

As mentioned above, a limitation of the CLP(R) system is its inability to solve non-linear constraints. In our constrained object paradigm which is based on the CLP(R) system, we overcome this limitation by employing powerful solvers such as Maple [112] to solve non-linear constraints. Chapter 5 describes our scheme for partial evaluation of constrained object programs that facilitates such handling of non-linear constraints.

2.2.2 Applications

Constraint logic programming finds use in a wide variety of computationally intensive applications, e.g., scheduling, mathematical modeling, engineering modeling, optimization. We present examples drawn from [46, 81] to illustrate the syntax and application of the CLP methodology.

Recursive Predicates with Constraints (Mortgage Example)

The CLP predicate `mortgage` below gives the relationship between a principal amount (P), the duration of the loan in months (T), the rate of interest (I), the monthly installment (MP) and the balance at the end of a period (B). The interest is compounded monthly.

```
mortgage(P, T, I, B, MP) :-
    T = 1,
    B = P + (P*I - MP).
mortgage(P, T, I, B, MP) :-
    T > 1,
    mortgage(P*(1 + I) - MP, T - 1, I, B, MP).
```

There are two rules defining the mortgage predicate. The first one states that after one monthly payment of MP, the outstanding balance B is obtained by first taking the sum of the principal (P) and the interest for one period ($P \cdot I$), and then subtracting the monthly payment (MP). The second rule is recursive and relates the mortgage attributes between successive time periods. The outstanding balance B on a principal P after T monthly payments of MP each, can be obtained as the outstanding balance for a principal of $P + P \cdot I - MP$ at the same interest rate and monthly payments but after $T-1$ months. This relation holds only if the life of the loan is more than one month (expressed as the constraint $T > 1$).

Suppose a sum of 1000000 is to be paid back in 30 years at a monthly interest rate of 0.01. The monthly payment can be computed by the goal G_0 below.

`?- mortgage(1000000, 360, 0.01, 0, M).`

A 0 in the argument corresponding to the outstanding balance indicates that the loan is to be fully repaid at the end of 30 years. The evaluation of goal G_0 with respect to the above mortgage program proceeds as follows. The goal matches the first clause or rule of the program and results in the goal G_1 which is a collection of explicit constraints shown below:

$P = 1000000, T = 360, I = .01, B = 0, MP = M,$
 $T = 1, B = P + (P \cdot I - MP).$

Since each of the above subgoals is an explicit constraint, by multiple applications of the second derivation step given in Section 2.2.1, they are added to the constraint store which is tested for consistency by the solver. Clearly this constraint set is not satisfiable (is not consistent), and the system backtracks and picks the second clause. Matching the goal to its head and performing goal reduction results in goal G_1 as follows:

$P = 1000000, T = 360, I = .01, B = 0, MP = M,$
 $T > 1, \text{mortgage}(P \cdot (1+I) - MP, T-1, I, B, MP).$

By multiple applications of the derivation steps described in the Section 2.2.1, we obtain the goal G_n as follows:

$\text{mortgage}(P \cdot (1+I) - MP, T-1, I, B, MP)$

and the constraint store C_n :

$$P = 1000000, T = 360, I = .01, B = 0, MP = M, T > 1.$$

Since the constraint store C_1 is consistent (solvable), the constraints are solved and the goal G_n is simplified to

$$\text{mortgage}(1000000*(1+.01) - MP, 359, .01, 0, MP).$$

This goal is again matched with the first clause but fails and the computation backtracks to pick the second clause. Resolving the goal using the second clause leads to the following goal:

$$P' = 1000000*(1+.01) - MP, T' = 359, I' = .01, B' = 0, MP' = MP, T' > 1, \\ \text{mortgage}(P'*(1+I) - MP', T'-1, I', B', MP').$$

which on further derivation becomes

$$\text{mortgage}((1000000*(1+.01) - MP')*(1+.01), 358, .01, 0, MP').$$

and the constraint C_3 $P' = 1000000*(1+.01) - MP, T' = 359, I' = .01, B' = 0, MP' = MP, T' > 1.$

are added to the constraint store. Since the constraint store $(C_2 \cup C_3)$ is consistent, the derivation proceeds to pick another atom from the goal. This process continues until a goal is successfully resolved using the first clause. At that point sufficient information is available to solve the constraints to obtain the answer constraint

$$M = 10286.1$$

As another query, suppose only the interest rate and the life of loan are known. The query

$$\text{mortgage}(P, 720, 0.01, B, M)$$

generates the answer constraint

$$M = -7.74367e-06*B + 0.0100077*P$$

which relates the principal, balance and monthly payments. This example illustrates the expressive power of the CLP(R) language. There is no need to explicitly distinguish input or output variables, as is expected in a true declarative constraint language.

As mentioned earlier, in the CLP scheme, the query itself can have constraints. Suppose that an affordable monthly payment must lie between \$1000 and \$1200. Now, if the loan is to be paid back in 30 years at a monthly interest rate of 1%, how much loan can be borrowed? The following query represents this question

$$\text{mortgage}(P, 360, 0.01, 0, M), M > 1000, M < 1200.$$

The answer is also in the form of a constraint:

```
P < 116662
97218.3 < P
```

The above program and queries illustrate several of the aspects of constraints mentioned earlier in section 2.1: constraints are declarative, non-directional, additive and may give partial information.

Test-then-Generate (Combinatorics). The next example illustrates the use of constraints for a test-then-generate strategy (instead of the usual generate-then-test strategy) of searching for a solution. Consider the classic crypt arithmetic puzzle:

```
      S E N D
+     M O R E
-----
= M O N E Y
```

The problem is to determine a unique digit between 0 to 9 for every letter in the above formula so that when each letter is replaced by its corresponding digit, the formula holds true. The CLP(R) program below models this problem [81] and the top level predicate is `solve`.

```
solve([S, E, N, D, M, O, R, Y]) :-
    constraints([S, E, N, D, M, O, R, Y]),
    gen_diff_digits([S, E, N, D, M, O, R, Y]).

constraints([S, E, N, D, M, O, R, Y]) :-
    S >= 0, S <= 9,
    ... similar constraints on variables
    ... E, N, D, M, O, R, Y ...
    S >= 1, M >= 1,
        1000*S + 100*E + 10*N + D
        + 1000*M + 100*O + 10*R + E
    = 10000*M + 1000*O + 100*N + 10*E + Y.
```

The predicate `gen_diff_digits` iteratively goes through all permutations of assignment of different digits from 0 to 9 to the letters. The predicate `constraints`

places constraints on the possible values for the letters and hence prunes the search for a solution. Because of the constraints, the predicate `gen_diff_digits` does not iterate over all the permutations. This is because the constraints are tested for every partial assignment of values to variables. Any partial assignment that violates a constraint is abandoned and the system backtracks to pick an alternative value for the most recently instantiated variable (that still has alternatives). Thus a subset of assignments is eliminated each time a partial assignment fails a constraint. This scheme is the backtracking (BT) algorithm discussed in Section 2.1.2. The above program computes the answer $S = 9, E = 5, N = 6, D = 7, M = 1, O = 0, R = 8, Y = 2$.

Suppose now that instead of placing the constraints before generating the digits, we were to switch the subgoals and define `solve` as:

```
solve([S, E, N, D, M, O, R, Y]) :-
    gen_diff_digits([S, E, N, D, M, O, R, Y]),
    constraints([S, E, N, D, M, O, R, Y]).
```

The new definition of `solve` will take a significantly longer time to evaluate than the earlier definition. This is because every permutation of digits is generated and then tested to check if it satisfies the constraints. This is the generate-then-test (GT) algorithm discussed in Section 2.1.2.

Electrical Circuits. We show the CLP(R) formulation of a circuit problem given in [46]. This example illustrates the power of the CLP(R) system and also serves to compare against our work in later chapters. We show the significant parts of the CLP(R) code for modeling and analysis of steady state RLC circuits.

Suppose resistors R1 (100 Ohms) and R2 (50 Ohms) are connected in series with a voltage source V1 (10 Volts). A diode D1 is connected in parallel with resistor R2. This circuit is shown in Figure 2.1. The top level query for obtaining the voltages at various points in such a circuit is:

```
?- W = 0, Vs = 10, R1 = 100, R2 = 50,
    circuit_solve(W, [[voltage_source,v1,c(Vs,0),[n1,ground]],
                      [resistor,r1,R1,[n1,n2]],
                      [resistor,r2,R2,[n2,ground]],
                      [diode,d1,in914,[n2,ground]]],
                  [ground], [n2])).
```

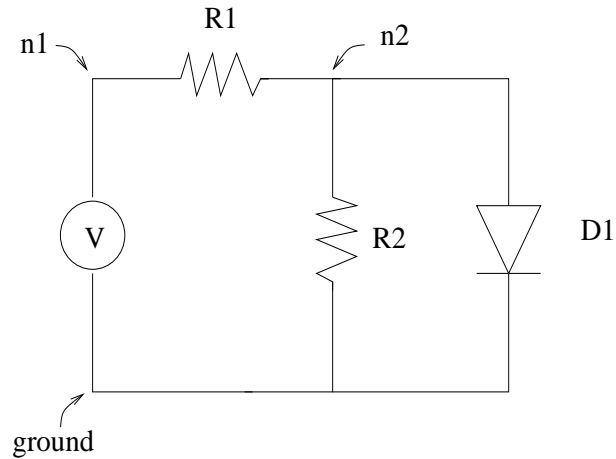


Figure 2.1: Simple Electrical Circuit

In the above query, `v1`, `r1`, `r2`, `d1` are names or labels of the circuit components. Various points or nodes in the circuit are labeled `n1`, `n2`, `ground`. The second argument to predicate `circuit_solve` is a list of components of the circuit containing information (e.g. current, voltage) about the component and the nodes to which it is connected. Current and voltage are complex numbers represent as `c(X, Y)` where `X` is the real part and `Y` the imaginary part. For example, resistor `R1` is connected to nodes `n1` and `n2`. The third argument to `circuit_solve` is a list of nodes that are grounded, and the fourth argument is the list of nodes whose information (current and voltage) should be displayed on the screen.

```
circuit_solve(W, L, G, Selection) :-
    get_node_vars(L, NV),
    solve(W, L, NV, Handles, G),
    format_print(Handles, Selection).
```

The predicate `get_node_vars` extracts the nodes in the circuit (in the above circuit, these are `n1`, `n2`, `ground`) and associates a distinct voltage variable and a current of zero with each of them. Thus `NV` unifies with a list of 3 tuples: each tuple represents a node and its voltage and current.

```
solve(W, [X|Xs], NV, [H|Hs], G) :-
    addcomp(W, X, NV, NV1, H),
    solve(W, Xs, NV1, Hs, G).
solve(W, [], NV, [], G) :-
```

```

zero_currents(NV),
ground_nodes(NV, G).

```

The predicate `solve` iterates through the list of components in the circuit adding the constraints associated with each to the constraint store.

```

addcomp(W, [Comp2, Num, X, [N1, N2]], NV, NV2,
         [Comp2, Num, X, [N1, V1, I1], [N2, V2, I2]]) :-
    c_neg(I1, I2),
    iv_rein(Comp2, I1, V, X, W),
    c_add(V, V2, V1),
    subst([N1, V1, Iold1], [N1, V1, Inew1], NV, NV1),
    subst([N2, V2, Iold2], [N2, V2, Inew2], NV1, NV2),
    c_add(I1, Iold1, Inew1),
    c_add(I2, Iold2, Inew2).

```

The predicate `addcomp` associates a voltage and current with each node of the component. The predicates `c_neg` and `c_add` perform negation and addition of complex numbers respectively. The current at one node is the negative of the current at the other node `c_neg`. The voltage across the component is the difference in the potential drop at the two nodes `c_add`. The list of nodes `NV` is updated by adding the currents `I1`, `I2` to the current computed so far at the nodes (`subst` and `c_add`). The `subst` predicate ensures that the voltage computed at a node for a component be equal to the voltage computed at the same node via a different component.

The predicate `zero_currents` equates the current at each node to zero. In effect this means that the sum of the currents at each node add up to zero. The predicate `ground_nodes` equates the voltage at each node (in its argument) to zero.

```

iv_rein(resistor, I, V, R, W) :-
    c_mult(I, c(R, 0), V).
iv_rein(diode, I, V, D, W) :-
    diode(D, I, V).

diode(in914, c(I, 0), c(V, 0)) :-
    (V < 0-100, DV = V + 100, I = 10*DV);
    (V >= 0-100, V < 0.6, I = 0.001*V);
    (V >= 0.6, DV = V - 0.6, I = 100*DV).

```

The predicate `iv_rein` is defined as a case analysis on the type of component and gives rise to the appropriate constraints associated with each component. There will be

additional clauses for transistor, inductor, capacitor, etc. The syntax $A ; B$ above denotes the disjunction of goals A and B .

The above is an expressive and powerful CLP(R) program that can analyze RLC circuits with different inputs, different known and unknown variables. The complete code for the program is given in [46]. We would like to point out, however, that understanding the code without the detailed explanation that we have given above is a non-trivial task. The implementation of Kirchoff's and Ohm's law in the above code is very roundabout (via `subst`) and non-intuitive. In section 3.2.4 we give a Cob model of RLC circuits which uses a natural and intuitive representation of electrical components.

2.3 Preference Logic Programs

Constraint satisfaction problems often have more than one solution and there may be a preference criterion based on which one solution is considered better than another. The goal of constraint optimization is to use the preference criterion to determine optimal solution(s) to a constraint satisfaction problem. Thus a *constraint optimization* problem consists of a set of variables, a set of constraints over these variables, and one or more preference criteria. In this section we describe a programming paradigm that facilitates a declarative specification of constraint optimization problems whose computational engine computes the optimal solutions.

Preference logic programming is an extension of constraint logic programming to handle constraint optimization problems. A preference logic program can be thought of as a logic program having two types of predicates: ordinary predicates and optimization predicates. Optimization predicates have a preference clause that facilitates comparison between solutions to the predicate. We give the syntax and some examples of PLP programs below. This description is summarized from [38] which contains a thorough discussion of the PLP paradigm.

2.3.1 PLP Syntax

A preference logic program has two parts: a **first-order theory** and an **arbiter** [38]. The first-order theory consists of clauses that can have one of two forms:

1. *Definite clause*: $H \leftarrow B_1, \dots, B_n$, ($n \geq 0$). Each B_i is of the form $p(\bar{t})$ where p is a predicate and \bar{t} is a sequence of terms. A predicate and term are the same as in CLP context. In general, B_i could be a constraint as in CLP [59, 61].
2. *Optimization clause*: $H \rightarrow C_1, \dots, C_l \mid B_1, \dots, B_m$, ($l, m \geq 0$). Each C_i is a constraint as in CLP [59, 61] and C_1, \dots, C_l must be satisfied for this clause to be applicable to a goal. The variables that appear *only* on the RHS of the \rightarrow clause are existentially quantified. The intended meaning of this clause is that the set of solutions to the head is a subset of the set of solutions to the body.

Depending on the clauses used to define a predicate, it will belong to exactly one of the following types:

1. *C-predicates*: These are defined using definite clauses and the bodies of their definition contain only other *C-predicates* (C stands for *core*).
2. *O-predicates*: These are defined using optimization clauses (O stands for *optimization*). An instance of the *O-predicate* at the head of an optimization clause is a candidate for an optimal solution if the corresponding instance of the body of the clause is true.
3. *D-predicates*: These are defined using only *definite* clauses whose body contains one or more *O-predicate* or *D-predicate* goals. (D stands for *derived from O-predicates*.)

The **preference** or **arbiter** clauses of a preference logic program specify the **optimization criterion** for the *O-predicates*. They are of the form:

$$p(\bar{t}) \preceq p(\bar{u}) \leftarrow L_1, \dots, L_n \quad (n \geq 0)$$

where p is an *O-predicate* and each L_i is a constraint as in CLP, or an atom formed from a *C-predicate*. The intended meaning of such a clause is that $p(\bar{t})$ is *less preferred than* $p(\bar{u})$ if L_1, \dots, L_n are true.

A preference logic program is a collection of definitions of *C-predicates*, *O-predicates* and *D-predicates*. Formally, a PLP program is a 3-tuple of the form $\langle \mathcal{T}_C, \mathcal{T}_O, \mathcal{A} \rangle$ where \mathcal{T}_C is the set of the definitions of the *C-predicates*, \mathcal{T}_O is the set of the definitions of the *O-predicates* and *D-predicates*, and \mathcal{A} is the set of preference clauses in the program.

2.3.2 Examples

Preference logic programming finds application in the areas of parsing [15, 66], databases [19, 41], optimization [26, 27, 40], and default reasoning [14, 23, 80, 98]. We give below some simple examples to illustrate the syntax and meaning of preference logic programs. These examples are taken from [38].

Consider the problem of finding the shortest distance between two vertices of a graph. Each edge of the graph is associated with a cost and the cost of a path is the sum of the costs of its constituent edges. The graph is given as a set of facts for the `edge` predicate. The predicate `dist` is defined as:

$$\begin{aligned} \text{dist}(X, Y, C) &\leftarrow \text{edge}(X, Y, C). \\ \text{dist}(X, Y, C) &\leftarrow \text{edge}(X, Z, C1), \\ &\quad \text{dist}(Z, Y, C2), \\ &\quad C = C1 + C2. \end{aligned}$$

The shortest distance between two vertices is given by a path between the vertices having the least cost. We show two different formulations of the shortest distance problem from [38].

The first is a naive formulation of the problem that compares all possible paths between two vertices in order to compute the shortest distance between them as shown by the `naive_sh_dist` predicate defined below.

$$\begin{aligned} \text{naive_sh_dist}(X, Y, C) &\rightarrow \text{dist}(X, Y, C). \\ \text{naive_sh_dist}(X, Y, C1) &\preceq \text{naive_sh_dist}(X, Y, C2) \leftarrow C2 < C1. \end{aligned}$$

The optimization clause states that a shortest distance is a distance. Thus the set of all paths between X and Y are candidates (feasible solutions) for the shortest distance path. The arbiter clause states that a path between X and Y having cost $C1$ is less preferred to a path with cost $C2$ between the same vertices if $C2$ is less than $C1$. In the above definition,

`dist` and `edge` are *C-predicates* while `naive_sh_dist` is an *O-predicate*.

Informally, the answer to a goal $G \equiv ?- \text{naive_sh_dist}(a, b, C)$ is computed by first computing the solutions to the subgoal $\text{dist}(a, b, C)$. This forms the set of feasible solutions to G . Next, all those solutions S_i for which there exists another solution S_j such that S_i is less preferred to S_j are eliminated from the set of feasible solutions. The remaining feasible solutions form the set of optimal solutions to G .

We now give a more efficient formulation of the shortest distance problem [40]. The goal $\text{sh_dist}(X, Y, N, C)$ computes the minimal cost (C) path from vertex X to vertex Y of length N (involving N hops or edges). The predicate `sh_dist` is defined recursively. The cost of a feasible shortest distance path between X and Y involving N hops (where N is more than 1) is the sum of the cost of the shortest distance (of 1 hop) from X to some immediate neighbor vertex Z (other than Y) and the cost of the shortest distance between Z and Y . If there are two feasible solutions of differing costs between the same vertices, then the solution with less cost is preferred.

$$\begin{aligned} &\text{sh_dist}(X, X, N, 0). \\ &\text{sh_dist}(X, Y, 1, C) \rightarrow X \neq Y \mid \text{edge}(X, Y, C). \\ &\text{sh_dist}(X, Y, N+1, C_1+C_2) \rightarrow N > 1, X \neq Y \mid \\ &\quad \text{sh_dist}(X, Z, 1, C_1), \text{sh_dist}(Z, Y, N, C_2). \\ &\text{sh_dist}(X, Y, N, C_1) \preceq \text{sh_dist}(X, Y, N, C_2) \leftarrow C_2 < C_1. \end{aligned}$$

The shortest distance problem has the optimal sub-problem property, viz., that the optimal solution to the problem is composed of optimal solutions to its sub-problems. In the above formulation, this property is stated explicitly and the recursive definition of `sh_dist` obtains an optimal path of length n between X and Y by extending the shortest paths of length $n - 1$ between an intermediate vertex and Y . Note that the variable Z that appears only on the right of the second clause is existentially quantified. Thus the set of feasible solutions to $\text{sh_dist}(a, b, 10, C)$ comprise of the shortest paths of length 9 between neighbors of a and b .

Goals in a PLP program must have certain variables instantiated. For example, in the above program, the goal $\text{sh_dist}(a, b, N, C)$ will not return the correct answer since N is not ground.

2.4 Summary

In this chapter we gave an overview of some important constraint programming paradigms, their basic concepts, their use in modeling constraint satisfaction problems and their applications to various domains. We described the CLP(R) language in detail with several example programs from different application domains. Our work on constrained objects is very closely related to the constraint logic programming paradigm. In later chapters (3 and 7) we will illustrate the expressiveness of constrained object programs and compare them with the equivalent CLP(R) programs given in this chapter. In Chapter 5 we show how our execution model for constrained objects handle non-linear constraints which CLP(R) is unable to solve. Also, the declarative and operational semantics of constrained objects defined in this dissertation (Chapter 4) are closely related to the semantics of constraint logic programs.

In the next chapter we introduce a programming language and modeling environment named **Cob** based on the concept of constrained objects and compare it to some of the relevant paradigms described in this chapter. The paradigm of constrained objects is also useful in expressing constraint optimization problems. We give examples of this in Chapter 5 and describe the operational semantics of Cob programs with preferences in terms of the operational semantics of preference logic programs. We also introduce the notion of relaxation (sub-optimization) goals in PLP with respect to the underlying preference criterion and describe their operational semantics in Chapter 7. The overview of CLP and PLP given in this chapter thus forms a background to understand and compare with the material presented in the rest of the dissertation.

Chapter 3

A Constrained Object Language

This chapter informally presents the language Cob and describes its modeling environment. Section 3.1 gives a detailed syntax of Cob programs including class definitions, attributes, terms, different types of constraints and user-defined predicates. Section 3.2 shows several examples illustrating the syntax of Cob, the use of the different types of constraints with the different constructs and features of the language, and the application of the Cob language to engineering modeling. Section 3.3 describes the Cob modeling environment which includes a visual tool for authoring Cob class diagrams and different domain specific visual interfaces for drawing constrained object models of engineering structures. Section 3.4 compares Cob with existing constrained object languages and other approaches to integrating constraints with object-oriented and logic programming. Section 3.5 summarizes the contributions of the material presented in this chapter.

3.1 Syntax of Cob Programs

Cob Program. A Cob program is a sequence of class definitions, and each constrained object is an instance of some class.

$$program ::= class_definition^+$$

Class. A class definition consists of attributes, constraints, predicates and constructors.

$$class_definition ::= [abstract] class class_id [extends class_id] \{ body \}$$

$$\begin{aligned}
\textit{body} \quad ::= & \quad [\textit{attributes} \textit{attributes}] \\
& [\textit{constraints} \textit{constraints}] \\
& [\textit{predicates} \textit{pred_clauses}] \\
& [\textit{constructors} \textit{constructor_clauses}]
\end{aligned}$$

Each of these constituents is optional, and an empty class definition is permitted as a degenerate case. Single inheritance of classes is permitted and the subclass inherits the constraints of its superclass. There can be more than one constructor for a class. An abstract class is a class without any constructor, and hence cannot be instantiated. A class name must begin with a lowercase letter, e.g. `component`, `series`, `parallel`.

Although multiple inheritance is conceptually allowed in the general notion of constrained objects, it is not supported by the current implementation. Conceptually, a class inheriting from more than one classes will inherit the attributes and constraints of each of its superclasses. The values of the attributes of such a class will be determined (partially) by the conjunction of its own constraints with those of its superclasses (provided this set of constraints is consistent). Since constraints are declarative and additive, such a semantics for multiple inheritance does not have the difficulties that are associated with providing multiple inheritance of procedural methods in traditional object-oriented programming.

3.1.1 Attributes and Terms

Attribute. An attribute is a typed identifier, where the type is either a primitive type or a user-defined type (i.e., class name) or an array of primitive or user-defined types.

$$\begin{aligned}
\textit{attributes} \quad ::= & \quad \textit{decl} \ ; \ [\textit{decl} \ ; \]^+ \\
\textit{decl} \quad ::= & \quad \textit{type} \ \textit{id_list} \\
\textit{type} \quad ::= & \quad \textit{primitive_type_id} \mid \textit{class_id} \mid \textit{type} \ [\] \\
\textit{primitive_type_id} \quad ::= & \quad \textit{real} \mid \textit{int} \mid \textit{bool} \mid \textit{char} \mid \textit{string} \\
\textit{id_list} \quad ::= & \quad \textit{attribute_id} \ [\ , \ \textit{attribute_id} \]^+
\end{aligned}$$

The size of an array may be constant or left unspecified, e.g.,

```

component [ ] [ ] VarSize2DArray;
component [3] FixedSizeArray;

```

An attribute name must begin with an uppercase letter, e.g., *V*, *I*, *R*. There are two internal variables used by the Cob compiler (*Cob* and *cob*), and these names must not appear in a Cob program. For the same reason, the *_* symbol must not appear as part of an attribute name. All the attributes of a class must be declared at the beginning of the body of the class following the keyword *attributes*.

Term. Terms can appear in constraints or as arguments to functions, predicates or constructors.

$$\begin{aligned}
 \textit{term} & ::= \textit{constant} \mid \textit{var} \mid \textit{complex_id} \\
 & \quad \mid \textit{arithmetic_expr} \mid \textit{func_id}(\textit{terms}) \\
 & \quad \mid [\textit{terms}] \mid (\textit{term}) \\
 & \quad \mid \textit{aggreg_op} \textit{ var in enum : term} \\
 \textit{aggreg_op} & ::= \textit{sum} \mid \textit{prod} \mid \textit{min} \mid \textit{max} \\
 \textit{terms} & ::= \textit{term} [, \textit{term}]^+
 \end{aligned}$$

A term may be an arithmetic or boolean expression involving attributes, constants from the primitive types, and built-in functions (e.g. *sin*, *cos*, etc.). We also support lists as a built-in type. A list is a comma separated sequence of terms enclosed within *[* and *]*.

We provide a shorthand for ‘aggregation terms’, where the iteration is over indices or elements of an array or elements of an explicitly specified set. This notation is similar to its mathematical equivalent. For example, the term

$$\begin{aligned}
 \textit{sum X in PC: (X.V)} & \quad \text{stands for} \quad \sum_{i=1}^n \textit{PC}[i].V \\
 \textit{prod Y in IntArray: Y} & \quad \text{stands for} \quad \prod_{i=1}^m \textit{IntArray}[i] \\
 \textit{min X in RealArray: X^2} & \quad \text{stands for} \quad \min\{X^2 \mid X \in \textit{RealArray}\}
 \end{aligned}$$

where $n = \text{length of PC}$ and $m = \text{length of IntArray}$.

Complex_id. A complex identifier refers either to an array element or to an attribute of an object. Attribute selection is specified by the usual dot (*.*) notation.

$$\begin{aligned}
 \textit{complex_id} & ::= \textit{attribute_id}[\textit{attribute_id}]^+ \\
 & \quad \mid \textit{complex_id}[\textit{term}]
 \end{aligned}$$

For example, *X.Y[3]* is a *complex_id*. It is understood that if *X* is of type *t*, then *t* must have an attribute named *Y* whose type is an array of size at least 3.

Scoping. A subclass may declare an attribute already declared by its super class. Suppose class *b* is a subclass of *a*, and suppose both *a* and *b* declare an attribute *V*. Any use of *V* within *b* refers to the attribute defined in *b*. Similarly, the variables occurring as formal parameters of a constructor definition override their namesake in the class and/or its superclass.

3.1.2 Constraints and Predicates

Constraint. A constraint specifies a relation over the attributes of one or more classes. A class can have zero or more constraints. When the class is instantiated, the attributes of the instance are subject to these constraints.

$$\begin{aligned}
 \text{constraints} &::= \text{constraint} ; [\text{constraint} ;]^+ \\
 \text{constraint} &::= \text{simple_constraint} \mid \text{quantified_constraint} \\
 &\quad \mid \text{creational_constraint} \\
 \text{creational_constraint} &::= \text{complex_id} = \text{new class_id}(\text{terms}) \\
 \text{quantified_constraint} &::= \text{forall var in enum} : (\text{constraints}) \\
 &\quad \mid \text{exists var in enum} : (\text{constraints}) \\
 \text{simple_constraint} &::= \text{conditional_constraint} \mid \text{constraint_atom} \\
 \text{conditional_constraint} &::= \text{constraint_atom} : - \text{literals} \\
 \text{constraint_atom} &::= \text{term relop term} \mid \text{constraint_predicate_id}(\text{terms}) \\
 \text{relop} &::= = \mid \neq \mid > \mid < \mid >= \mid <=
 \end{aligned}$$

A constraint can be simple, quantified or creational. A simple constraint can either be a constraint atom or a conditional constraint. These constraints are explained below.

Constraint Atom. A constraint atom is a relational expression of the form *term relop term*, where *term* is composed of functions/operators from the data domain (e.g. integers, reals, etc.) as well as constants and attributes. Examples:

```

V = I * R;
Theta <= 2* 3.141;
X = sin(Theta);

```

The = symbol is a comparison operator and should not be mistaken as an assignment operator. The <= symbol is also a comparison operator and has the usual mathematical meaning.

The other form of a constraint atom is *constraint_predicate_id(terms)* where the *constraint_predicate* is one whose properties are known to the underlying system.

Conditional Constraint. A conditional constraint is a constraint atom that is predicated upon a conjunction of literals each of which is a (possibly negated) ordinary atom or a constraint atom.

$$\text{constraint_atom} \quad :- \quad \text{literals}$$

The conjunction of literals on the right hand side of the $:-$ is referred to as the antecedant and the *constraint_atom* at the head of the conditional constraint (left hand side of the $:-$ symbol) is referred to as the consequent. Although we employ Prolog-like syntax for defining conditional constraints the evaluation of a conditional constraint is very different from that of a Prolog clause.

- If the constraint atom at the head of a conditional constraint is entailed by the current state of the constrained object, then the conditional constraint is considered to be satisfied.
- If the negation of the head of the conditional constraint is entailed by the current state of the constrained object, then the antecedant must evaluate to false in order for the conditional constraint to be satisfied.
- Similarly, if the antecedant is entailed by the state of the constrained object, then for the conditional constraint to be satisfied, the consequent must evaluate to true.
- If the negation of the antecedant is entailed by the state of the constrained object, then the conditional constraint is considered to be trivially satisfied.

A conditional constraint cannot be nested, i.e., it cannot be defined in terms of another conditional constraint. There may be more than one (constraint) atom (possibly negated) in the antecedant but the consequent has exactly one (non-negated) constraint atom or creational constraint. For the antecedant to be entailed by the state of the constrained object,

every literal in it must be entailed by the state. For example consider the following conditional constraint.

```
F = Sy * W * H :- F > 0;
```

If the value of F is non-positive, then the above conditional constraint is trivially satisfied. But if F is positive then its value must be equal to $Sy * W * H$ in order for the conditional constraint to be satisfied.

As another example, consider the relation between the day, month and year attributes of a date shown below.

```
Day =< 29 :- Month = 2, leap(Year);
```

The above conditional constraint specifies that, for the month of February in a leap year, the day must be less than or equal to 29. Another way of reading this constraint is as follows: If the day is more than 29, then either the month is not February or it is not a leap year. Note that the conditional constraint will be satisfied for a day of 30 in February for a non-leap year. This is because this constraint alone does not capture the full behavior of a date object. A complete definition of a date as a constrained object is given in Section 3.2.

Conditional constraints can be used to control object creation dynamically. For example, consider the following conditional constraints over attributes X , Y , and $Shape$ of a certain class:

```
Shape = rectangle(X, Y) :- Input = 1
```

```
Shape = circle(X, Y) :- Input = 2
```

Together, they can be used to set a `Shape` attribute of, for example, a node of a binary tree. In the above example, X and Y stand respectively for the width and height inputs of the `rectangle` constructor; and they stand respectively for the center and radius attributes of the `circle` constructor.

Although in certain cases the behavior of a conditional constraint may resemble an `if $expr$ then $expr$` statement, it is fundamentally different from an imperative if-then statement. The consequent is enforced if *at any point* in the program the antecedent is entailed by the state of the constrained object.

We can also consider an if-then-else form of conditional constraint having two conse-

quents¹. For such a constraint to be satisfied, the following must hold: When the antecedant is entailed by the state of the constrained object, the ‘then consequent’ must evaluate to true; and when the negation of the antecedant is entailed by the state of the constrained object, the ‘else consequent’ must evaluate to true. As with the basic form of conditional constraint this type of if-then-else constraint is different from the imperative if-then-else statement.

Quantified Constraint. A quantified constraint is a shorthand for stating a relation where some participants of the relation range over enumerations (*enum*), i.e., the elements of a set. A quantified constraint is defined in terms of a quantified variable, an enumeration, and the constraint being quantified. The quantification can be either universal or existential. Examples:

```
forall C in Cmp: C.I = I;
exists N in Nodes: N.Value = 0;
```

For a universally quantified constraint to be satisfied, it must be satisfied by every member of the enumeration. On the other hand, an existentially quantified constraint is satisfied if it is satisfied by at least one of the members. The constraint in the body of a quantified constraint can be a simple, quantified, creational or conditional constraint. Any number of nesting of universally and existentially quantified constraints is permitted. The enumeration may be a variable whose type is an array, or an explicitly specified array of integers. We provide a shorthand *M . . N* to represent the sequence of integers from *M* to *N*. The quantification can thus range over the elements of an array or the indices of an array.

Creational Constraint. A creational constraint specifies the creation of an object of a user-defined class and binds it to an attribute. It is of the form *complex_id* = *new class_id(terms)* where *class_id* refers to the class being instantiated. A creational constraint is satisfied by invoking the constructor of *class_id* with *terms* as arguments. Therefore, the class being instantiated must have a constructor with corresponding arity. For example,

```
R1 = new component(V1, I1, 10);
```

creates an instance of the *component* class and binds it to the attribute *R1*. The argument to a constructor may be a constant or an attribute. It may also be the special symbol *_*.

¹This if-then-else form of conditional constraint will be provided in subsequent versions of the Cob language

This may be useful when creating a collection of instances whose attributes are unrelated. A creational constraint may appear as the head of a conditional constraint, and thus object creation can be predicated upon by a conjunction of literals.

Literal. A literal is an atom or the negation of an atom.

$$\begin{aligned}
 \textit{literals} &::= \textit{literal} [, \textit{literal}]^+ \\
 \textit{literal} &::= [\texttt{not}] \textit{atom} \\
 \textit{atom} &::= \textit{predicate_id}(\textit{terms}) \\
 &\quad | \textit{constraint_atom}
 \end{aligned}$$

If p is an n -ary predicate symbol and t_1, \dots, t_n are terms then $p(t_1, \dots, t_n)$ is an atom. An atom may also be a `constraint_atom` which has been defined earlier. An atom may involve a built-in predicate or a user-defined predicate defined below.

User-defined Predicate. A user-defined predicate stands for an n -ary relation over terms and is defined using Prolog-like rules:

$$\begin{aligned}
 \textit{pred_clauses} &::= \textit{pred_clause} . [\textit{pred_clause} .]^+ \\
 \textit{pred_clause} &::= \textit{clause_head} : - \textit{clause_body} \\
 \textit{pred_clause} &::= \textit{clause_head}. \\
 \textit{clause_head} &::= \textit{predicate_id}(\textit{terms}') \\
 \textit{clause_body} &::= \textit{goal} [, \textit{goal}]^+ \\
 \textit{goal} &::= [\texttt{not}] \textit{predicate_id}(\textit{terms}') \\
 \textit{terms}' &::= \textit{term}' [, \textit{term}']^+ \\
 \textit{term}' &::= \textit{constant} | \textit{var}' | \textit{function_id}(\textit{terms}')
 \end{aligned}$$

Note that the only variables that may appear in a *term* are attributes or those that are introduced in a quantification. These variables are generated by the non-terminal *var*. In the above grammar, the variables that appear in a *pred_clause* are the usual logic variables of Prolog. These are referred to as *var'* in the above syntax. Note that when a predicate is invoked in the constraints or constructor clause of a class, its arguments are *terms*. For the

sake of keeping the description relatively simple, we do not present all the syntactic details of user-defined predicates at this point. We keep to the above syntax for now and present more details when required in Chapter 6 and in the case study in Section 7.3. The complete syntax of Cob programs is given in Appendix A.

We distinguish between ordinary *predicate_id* and *constraint_predicate_id*. The former are defined by the user whereas the latter are a set of predefined predicates (as in CLP-like languages) whose properties are known to the underlying constraint satisfaction system.

The declarative semantics of a user-defined predicate are identical to the predicates found in logic programming languages, and their computation is based on resolution.

Constructor. A constructor is a means of creating an instance of a class. A class can have one or more constructors and a class without a constructor must be declared abstract.

$$\begin{aligned} \text{constructor_clauses} &::= \text{constructor_clause}^+ \\ \text{constructor_clause} &::= \text{constructor_id}(\text{formal_pars}) \{ \text{constructor_body} \} \\ \text{constructor_body} &::= \text{constraints} \end{aligned}$$

The *constructor_id* must be identical to the name of the class. The body of a constructor contains a sequence of ; separated constraints. These constraints hold throughout the life of an instance of the class and should not be interpreted as one-time/initialization-only constraints. When a creational constraint instantiates a class, its arguments are passed on to a call on the constructor having appropriate arity. Hence the different constructors of a class should differ by arity.

3.2 Examples

In this section we present examples of constrained object models to illustrate the syntax of Cob and its application to engineering modeling.

3.2.1 Conditional Constraints (Date Object)

We model a date of the Gregorian calendar (the commonly used one) as a constrained object. According to this calendar, the years are divided into two classes: common years and

leap years. A common year is 365 days in length; a leap year is 366 days, with an intercalary day, designated February 29. Leap years are determined according to the following rule: every year that is exactly divisible by 4 is a leap year, except for years that are exactly divisible by 100; these centurial years are leap years only if they are exactly divisible by 400. The definition of the `date` class below captures the above description. This example illustrates the use of the various constructs of the Cob language, including the use of conditional constraints.

```
class date {
  attributes
    int Day, Month, Year;
  constraints
    1 <= Year;
    1 <= Month;      Month <= 12;
    1 <= Day;        Day <= 31;
    Day <= 30 :- member(Month, [4,6,9,11]);
    Day <= 29 :- Month = 2, leap(Year);
    Day <= 28 :- Month = 2, not leap(Year);
  predicates
    member(X, [X|_]).
    member(X, [_|T]) :- member(X,T).
    leap(Y) :- Y mod 4 = 0, Y mod 100 <> 0.
    leap(Y) :- Y mod 400 = 0.
  constructor date(D, M, Y) {
    Day = D;
    Month = M;
    Year = Y;
  }
}
```

The conditional constraint

```
Day <= 29 :- Month = 2, leap(Year)
```

requires `Day <= 29` if the `Month` is February and the `Year` is a leap year. Computationally, an important difference between a conditional constraint and a Prolog rule is the following: if the head of a conditional constraint evaluates to true, then the body need not be evaluated; and, if the head evaluates to false, the body must fail in order for the conditional constraint to be satisfied. In contrast, in Prolog, if the head of a rule unifies with a goal, then the body of the rule must be evaluated; and, if the head does not unify, then the body need not be evaluated.

As mentioned earlier in Section 3.1 under conditional constraints, the above constraint alone is not sufficient to ensure that the month of February has at most 29 days. The conditional constraint

```
Day <= 28 :- Month = 2, not leap(Year)
```

requires `Day <= 28` if the `Month` is February and the `Year` is not a leap year. The above two conditional constraints together ensure that the month of February, in any year, has at most 29 days.

The above definition can be used to validate a given combination of `Day`, `Month`, and `Year` values, and also be used to generate, for example, a range of `Month` values for a given a combination of `Day` and `Year`. For example, if the `Day` is set to 31 and the `Year` to 1999, the set of possible values for `Month` can be deduced to be any integer between 1 and 12 but not 4, 6, 9, 11, or 2. While `1 <= Month <= 12` is directly obtained from the unconditional constraints for `Month`, it is possible to deduce, by a process of *constructive negation* of the goal `member(Month, [4, 6, 9, 11])`, that `Month` is not 4, 6, 9, or 11. And, it can deduce that `Month` is not equal to 2 from the conditional constraint `Day <= 28 :- Month = 2, not leap(Year)`. A possible question can also be posed by providing a range of days and years and asking for the range of months that would together make a valid date.

The above representation of a date as a constrained object illustrates how the behavior of an object can be stated declaratively through constraints. Conditional constraints provide a novel declarative means of expressing state dependent behavior. They can generate new constraints or new instances of objects depending upon the state of the model. The operational semantics of conditional constraints not only subsume the usual conditional

statements (if-then) found in imperative languages, they also provide a truly declarative means of reasoning about the conditional behavior of an object. The ‘date’ example also illustrates the use of predicates within a constrained object class definition. The logic-based definitions of relations or functions via predicates facilitates a declarative specification that can be reasoned with in order to determine the state of the object.

3.2.2 Constraints over Arrays (Heat Plate Object)

We would like to model a heat plate in which the boundary temperatures are specified. The temperature at any interior point is specified mathematically by a 2D Laplace Equation ($\nabla^2\phi = \frac{\partial^2\phi}{\partial x^2} + \frac{\partial^2\phi}{\partial y^2}$). This equation can be formulated in a discrete manner by a constraint which states that the the temperature at a point in the interior of the plate is the average of the temperature of its neighboring four points.

A Cob representation of this problem is shown below in a class called `heatplate`. The constructor initializes the temperature of the border of an 11 x 11 plate. The nested quantified constraint sets up the constraint relating the temperature of an interior node to its neighbors. When creating an instance of the `heatplate` class we can initialize the temperature at the border and the Cob computational engine will compute the heat at all the interior points. A diagram of a heat plate is shown in Figure 3.1.

```
class heatplate {
  attributes
    real [11][11] Plate;
  constraints
    forall I in 2..10:
      forall J in 2..10:
        Plate[I-1,J]+Plate[I+1,J]+Plate[I,J-1]+Plate[I,J+1] =
          4*Plate[I,J];
  constructor heatplate(A,B,C,D) {
    forall M in 1..11: Plate[1,M] = A;
    forall N in 1..11: Plate[11,N] = B;
    forall K in 2..10: Plate[K,1] = C;
```

	1	2		N
1				
2			[I-1, J]	
		[I, J-1]	[I, J]	[I, J+1]
			[I+1, J]	
N				

Figure 3.1: A Heat Plate

```

forall L in 2..10: Plate[L,11] = D;
}
}

```

The above example illustrates the expressiveness of quantified constraints. The nested quantified constraint stated in the `heatplate` class iteratively places a constraint between every interior point of the heatplate and its four neighbors. When an instance of the `heatplate` is created with the boundary temperatures passed as arguments to the constructor, these constraints are solved as a set of simultaneous linear equations to obtain the values at the interior points. If all the border points are not initialized, i.e., the constructor is passed some constants and some variables as arguments, then the computational engine returns as answer the value of the interior points in terms of these variables. In other words, a partial model along with the simplified constraints is returned.

The above is a classic problem modeled in CLP(R). As a comparison, we show below a CLP(R) representation of the heat plate given in [45]. The variables I , J in the code below correspond to the indexing variable of the Cob code above, with I_0 corresponding to $I-1$ above and I_1 corresponding to $I+1$ above. Compared to this CLP(R) represen-

tation, the Cob representation of the problem is easier to understand and very concise due to the use of quantified constraints.

```

laplace([_, _]).
laplace([I0, I, I1|T]):-
    laplace_vec(I0, I, I1),
    laplace([I, I1|T]).
laplace_vec([_, _], [_, _], [_, _]).
laplace_vec([-I0J0, I0J, I0J1|T1],
    [ IJ0, IJ, IJ1|T2],
    [-I1J0, I1J, I1J1|T3]):-
    I0J + I1J + IJ0 + IJ1 = 4 * IJ,
    laplace_vec([I0J, I0J1|T1], [IJ, IJ1|T2], [I1J, I1J1|T3]).
main:- X =
    [
        [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
        [100, -, -, -, -, -, -, -, -, -, 100],
        [100, -, -, -, -, -, -, -, -, -, 100],
        [100, -, -, -, -, -, -, -, -, -, 100],
        [100, -, -, -, -, -, -, -, -, -, 100],
        [100, -, -, -, -, -, -, -, -, -, 100],
        [100, -, -, -, -, -, -, -, -, -, 100],
        [100, -, -, -, -, -, -, -, -, -, 100],
        [100, -, -, -, -, -, -, -, -, -, 100],
        [100, -, -, -, -, -, -, -, -, -, 100],
        [100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100]
    ],
    laplace(X).

```

It is clear that the Cob model of the problem is far more readable than the above CLP(R) model. One reason is that the use of constrained objects allows a clear and separate statement of an object, its components, and the constraints governing their state. The other

reason is that quantified constraints provide a concise means for stating a large collection of iterative constraints. Thus, while the CLP(R) program can provide a solution, it is difficult to see the constraints that influence it (having variable names in the CLP(R) program correspond to the variable names in the Cob code does help). Also, how the constraints relate to the structure of the heatplate is not clear in the CLP(R) model.

Apart from readability, there are several other advantages to modeling the problem in Cob. In the above CLP(R) model of the problem, changing the size of the matrix or the initial values of the border is a cumbersome process that must be done manually and hence is likely to be error prone. In contrast changes to the size of the matrix or to its initialization in the Cob model are easy and quick to incorporate. The same Cob model can be used to create different instances of a heatplate with the border points set to different temperatures. Later in section 5.2.2 we show a different way to model this problem in Cob, such that the size of the matrix is an argument to the constructor, making it even easier to modify the size at run-time.

3.2.3 Constraints over Aggregated Objects (Simple Truss)

To illustrate the use of constrained objects in engineering design, we define Cob classes needed to model a simple truss structure, as shown in Figure 3.2. A truss consists of bars placed together at joints. The constraints in the beam class express the standard relations between its modulus of elasticity (E), yield strength (S_y), dimensions (L , W , H), moment of inertia (I), stress (σ) and the force (F) acting on the beam. Depending upon the direction of this force (inward or outward), it may act as either the buckling or tension force and will accordingly be related to the other attributes by different constraints. These relations are expressed as conditional constraints in the beam class. These conditions are taken from the text by Mayne and Margolis [84].

```
class beam {
  attributes
    real E, Sy, L, W, H, F, I, Sigma;
  constraints
```

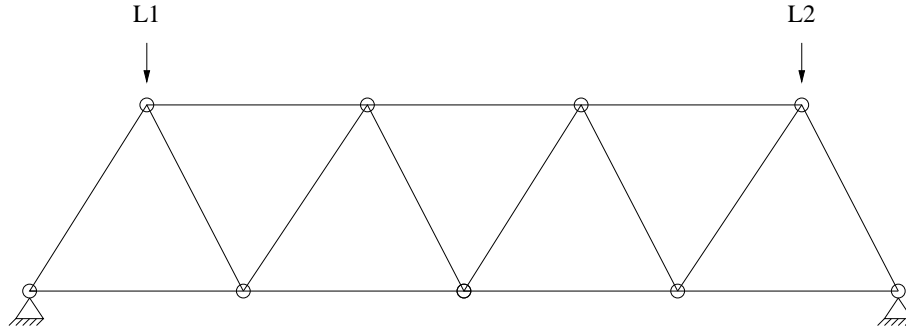


Figure 3.2: A Simple Truss

```

Pi = 3.141 ;
L > 0; W > 0; H > 0; I > 0;
I = W * H * H * H / 12;
(I = - F * L * L / (Pi * Pi * E)) :- F < 0;
(F = Sy * W * H) :- F > 0;
constructor beam(E1,Sy1,L1,W1,H1,Sigma1,I1,F1) {
    E=E1; Sy=Sy1; L=L1; H=H1; W=W1;
    Sigma=Sigma1; I=I1; F=F1;
}

```

The expressions on the left hand side of the two conditional constraints, $(\pi^2 * E * I) / L^3$ and $S_y * W * H$, represent respectively, the maximum tension and compression forces that can be applied to a bar without causing deformation. By equating them to an applied non-zero force, these expressions can be used to compute the minimum height or width for a bar required to support the applied force without the bar getting deformed. If the force is zero, these equations are not applicable.

A bar is a beam placed at an angle (A), and a load is a force applied at a certain angle. The two classes below model bars and loads. The two ends of a beam form a part of two different joints and hence are associated with two different angles. Hence, the same beam object can be used to form two different instances of a bar class. The `joint` class aggregates an array of bars and an array of loads (that are incident at the joint), and its constraints state that the sum of the forces in the horizontal and vertical directions respec-

tively must be 0. In this example, we restrict the law of balance of forces to two dimensions; in practice, we may need to use three dimensions (x, y and z).

```

class bar {
  attributes
    beam B; real A;
    % beam B placed at angle A
  constraints
    0 <= A; A <= 360;
  constructor bar(B1, A1) {
    B = B1; A = A1;
  }
}

class load {
  attributes
    real F; real A;
  constraints
    0 <= A; A <= 360;
  constructor load(F1, A1) {
    F = F1; A = A1;
  }
}

class joint {
  attributes
    bar [] Bars; load [] Loads;
  constraint
    sum X in Bars: (X.B.F * sin(X.A)) +
      sum L in Loads: (L.F * sin(L.A)) = 0;
    sum Y in Bars: (Y.B.F * cos(Y.A)) +
      sum M in Loads: (M.F * cos(M.A)) = 0;
  constructor joint(B1, L1) {
    Bars = B1; Loads = L1;
  }
}

```

The Cob classes defined above can be used to build a model of a truss on which loads may be incident at different points. Instances of the load and bar class are aggregated together to form various instances of joints, and the underlying computational engine tries

to solve the constraints of each of the instances of joints, bars, beams, etc. to obtain the forces playing in each member of the truss. If the incident loads are known, the model returns the forces in each member in terms of the unknown loads. The above model can be used to validate a design of a truss; test how much load it can bear; decide the material (e.g. steel) for the beams; and decide the thickness of the beams for a given load. By adding trigonometric constraints to the model, it is also possible to compute the length of each beam given the angle it forms at each joint. The above model can be extended to compute bending force in a bar when force is applied perpendicular to the bar. A bending force causes tension and compression in the lower and upper surfaces of the bar respectively and is related to the attributes of a bar by the equation $F = (8 * I * \text{Sigma}) / (H * L)$.

Constrained objects allow a modular representation of the truss built using subcomponents. There is a direct correlation between classes/objects and the physical components of the real structure, and hence the Cob model is easier to understand, use and manipulate than a pure constraint representation, as in CLP(R). For example, in the Cob representation, it is easy to use the same beam instance in different bar objects by simply using the same instance name of the beam. In contrast, in a CLP(R) representation, one has to keep track of the individual variables representing forces, angles, lengths etc. for every beam and reuse them where appropriate. Such a representation is not only non-intuitive, it is also prone to mistakes for large structures. The above constrained object model makes it easy to modify the number of bars, their angle as well as the loads. Also note that this model involves non-linear constraints. Such constraints cannot be solved in CLP(R). Our computational engine described in Chapter 5 provides a means by which more powerful symbolic math packages can be employed to solve such non-linear constraints.

3.2.4 Constraints with Inheritance and Aggregation (DC Circuits)

We model the well-known example of a series/parallel electrical circuit as a constrained object. The electrical circuit in Figure 3.3 consists of a battery connected to a series/parallel combination of resistors. We model the components and connections of such circuits as objects, and their properties and relations as constraints on the attributes of these objects. Given below is the code that defines these classes.

The component class below models any electrical entity that is characterized by its resistance and obeys Ohm's law. The attributes of this class represent the voltage, current and resistance and the constraint represents Ohm's law.

```
class component {
  attributes
    real V, I, R;
  constraints
    V = I * R;
  constructors component(V1, I1, R1) {
    V = V1;
    I = I1;
    R = R1;
  }
}
```

When a set of components are connected in series, the behavior of the resultant series object is similar to a component in that it is characterized by its effective voltage, current and resistance which are related by Ohm's law. For this reason, the `series` class is made a subclass of the `component` class and aggregates an array of components and is thus a recursive class definition. The `series` class inherits the voltage, current and resistance attributes and the Ohm's law constraint from the `component` class and further defines some constraints: its effective resistance and voltage is the sum of the resistances and voltages of the components it aggregates respectively; its effective current is equal to the currents through each of its constituent components. These relations are modeled as constraints in the `series` class. The `parallel` class is similarly defined.

```
class series extends component {
  attributes
    component [] Cmp;
  constraints
    forall C in Cmp: C.I = I;
    sum C in Cmp: C.V = V;
    sum C in Cmp: C.R = R;
  constructors series(A) {
    Cmp = A;
  }
}
class parallel extends component {
  attributes
```



```

        component [] PC;
constraints
    forall X in PC: X.V = V;
    sum X in PC: X.I = I;
    sum X in PC: 1/X.R = 1/R;
constructors parallel(B) {
    PC = B;
}
}

```

The battery class represents a source of constant DC voltage (V). The connect class models the connection of a battery across a component and equates their effective voltages.

```

class battery {
    attributes
        real V;
    constructors battery(V1) {
        V = V1;
    }
}
class connect {
    attributes
        battery B;
        component CC;
    constraints
        B.V = CC.V ;
    constructors connect(B1, C1) {
        B = B1;
        CC = C1 ;
    }
}

```

The above Cob classes can be used to model any series/parallel combination of resistors. Given initial values for some attributes, this model can be used to calculate values of the remaining attributes. For example, the class `samplecircuit` below models the circuit given in Figure 3.3. Given the battery voltage and resistances, this model can compute the current through every component of the circuit. In general, any of the attributes of a Cob model may be left unspecified and the underlying computational engine will return as answer the constraints on these attributes that must be satisfied for the Cob model to be valid.

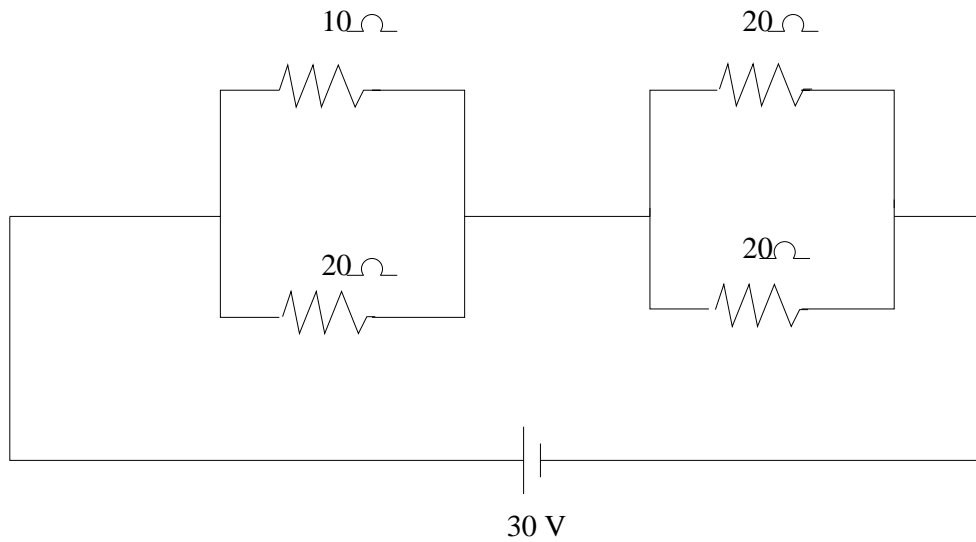


Figure 3.3: Simple Electrical Circuit

```

class samplecircuit {
  attributes
    battery B;
    connect C;
    component R1, R2, R3, R4, P1, P2, S;
    component[] R12, R34, P12;
  constructors samplecircuit() {
    R1 = new component(V1, I1, 10);
    R2 = new component(V2, I2, 20);
    R3 = new component(V3, I3, 20);
    R4 = new component(V4, I4, 20);
    P1 = new parallel([R1,R2]);
    P2 = new parallel([R3,R4]);
    S = new series([P1,P2]);
    B = new battery(30);
    C = new connect(B, S);
  }
}

```

The above Cob classes illustrate the use of quantified constraints where the quantification ranges over the elements of an array. The attribute `Cmp` of the `series` class represents an array of components and the quantified constraint `forall C in Cmp: C.I = I` iterates over every element of `Cmp`. As shown in the code for `samplecircuit`, the elements of `Cmp` can bind with variables of type `component` or any of its subclasses (viz., `series`, `parallel`). This polymorphism allows a variable declared to be of type `T` to

refer to an instance of any subclass of `T`. Such a variable can be used to access only those attributes of the instance that are inherited from class `T`. The `series` class inherits from `component` and also aggregates a set of components illustrating that a recursive class definition can be meaningful in Cob.

3.3 Cob Modeling Environment

In addition to the text based development of Cob code, we provide two types of modeling tools for developing Cob models. A domain-independent tool is provided for authoring Cob class diagrams. This tool can be used to generate as well as run Cob programs that model complex systems from any domain. The visual aspects of this tool were implemented by Rachna Jotwani, a recent graduate student of the Department of Computer Science and Engineering, University at Buffalo.

The process of building models of engineering structures is greatly aided if there is a visual tool that allows engineers to assemble parts of a structure. We provide a collection of domain dependent visual interfaces for drawing certain engineering structures. The compiler underlying each of these tools translates the drawings into Cob code which can then be executed (to obtain answers) within this tool. Currently there are tools under development for drawing analog as well as DC electrical circuits, trusses (civil engineering), and hydrological surfaces. In Section 3.3.2 we describe the modeling tool for drawing trusses. The visual aspects of this tool were implemented by Kapil Gajria, Abhilash Dev and Narayan Menon, recent graduate students of the Department of Computer Science and Engineering, University at Buffalo. We now describe these two types of modeling tools in more detail.

3.3.1 Domain Independent Interface

Consider the Cob classes defined for electrical circuits in Section 3.2.4. The Cob class diagram that represents these classes and their relationships is shown in Figure 3.4. Every node of the class diagram represents a class definition. The primitive attributes and constraints of the class are described inside the node. The aggregation relation $A \diamond \rightarrow B$ indicates that class `A` is composed of one or more constrained objects of the user-defined class `B`. In

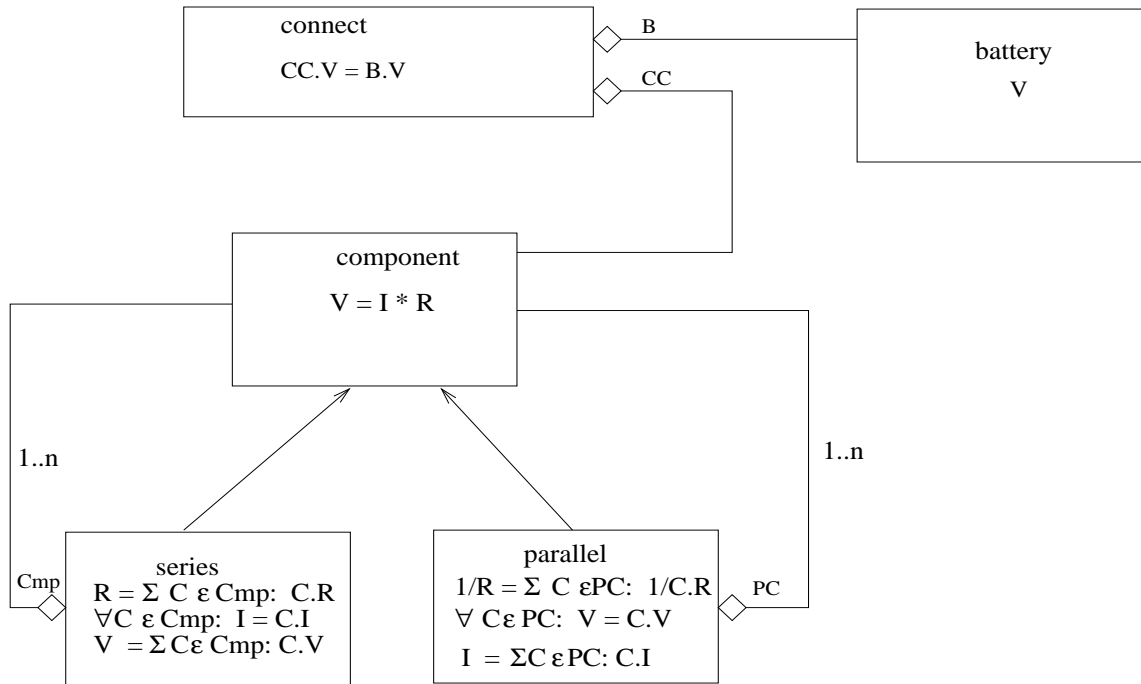


Figure 3.4: Class Diagram for Circuit Example

other words, class A has one or more attributes of type B. Since a constrained object class can be composed of one or more different classes, there can be an aggregation relation from A to B as well as from A to other classes. A label of $1..1$ or $1..n$ on the aggregation relation $A \diamond \rightarrow B$ indicates that that the relation is one-one (class A aggregates one object of type B) or one-many (class A aggregates n objects of type B) respectively. The subclass relation $C \rightarrow D$ states that C is a subclass of D. This notation is similar to that of the Unified Modeling Language (UML) [95].

Constraint-UML. We provide a domain-independent interface for drawing constrained object class diagrams. This interface consists of a palette of buttons for creating and editing the nodes and edges of a Cob class diagram (see Figure 3.5). The button for creating a node (class definition) places a node on the canvas containing a default skeletal class definition. By right-clicking on this node one can open the specifications of the node. This pops open a window (shown in Figure 3.6) through which the modeler can define or edit the class name and its primitive attributes, constraints, predicates and the constructors of the class.

There are buttons for defining relationships of inheritance and aggregation between

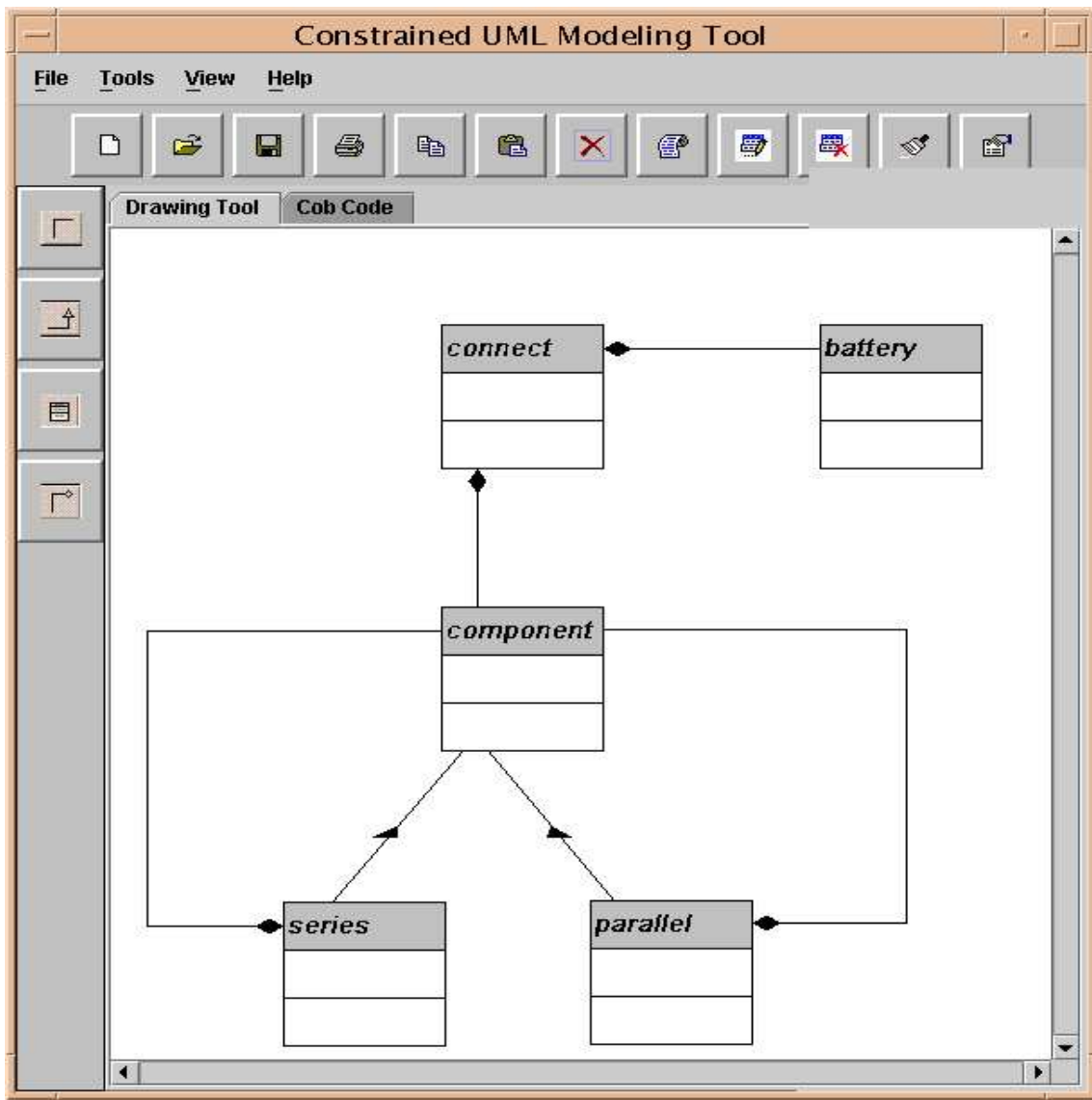


Figure 3.5: Snapshot of CUML tool with Class Diagram for Circuit Example

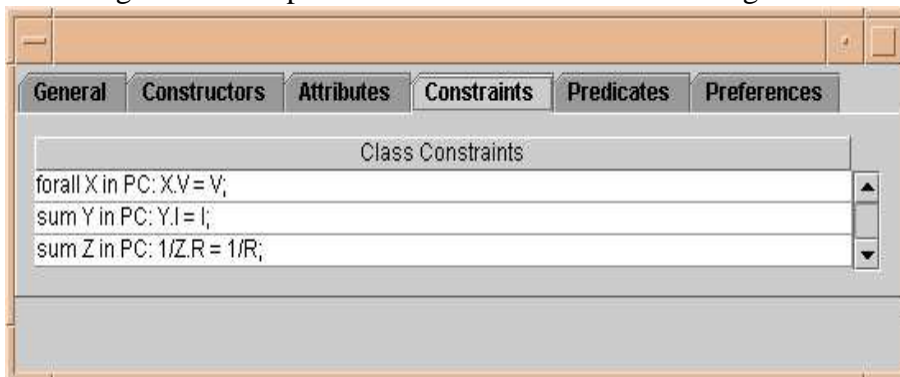


Figure 3.6: Popup window for entering/editing constraints and other specifications of a Class

classes by drawing the appropriate edges between them. The tool includes buttons for cutting, copying and pasting parts of the class diagram. The tool places the nodes in a default arrangement which can be modified by the user by dragging and dropping the nodes onto any part of the main canvas. The diagrammatic notation used is similar to the Unified Modeling Language, with extensions to describe constraints, and we refer to it as the Constraint-UML or **CUML**. Figure 3.5 shows a snapshot of this tool in which the class diagram of Figure 3.4 has been drawn. The modeler can choose to see the class diagram with full details of the attributes and constraints or, to reduce clutter, hide the details. In the snapshot shown in Figure 3.5 the details of the classes have been hidden.

In addition to the classes shown in Figure 3.5, the CUML tool can be used to build the class corresponding to a particular circuit, i.e., an assembly of components. For example, consider the electrical circuit shown in Figure 3.3. The class `samplecircuit` of Section 3.2.4, corresponding to this circuit, can be defined using the CUML tool. Figure 3.7 shows the window through which the constructor of this class is specified.

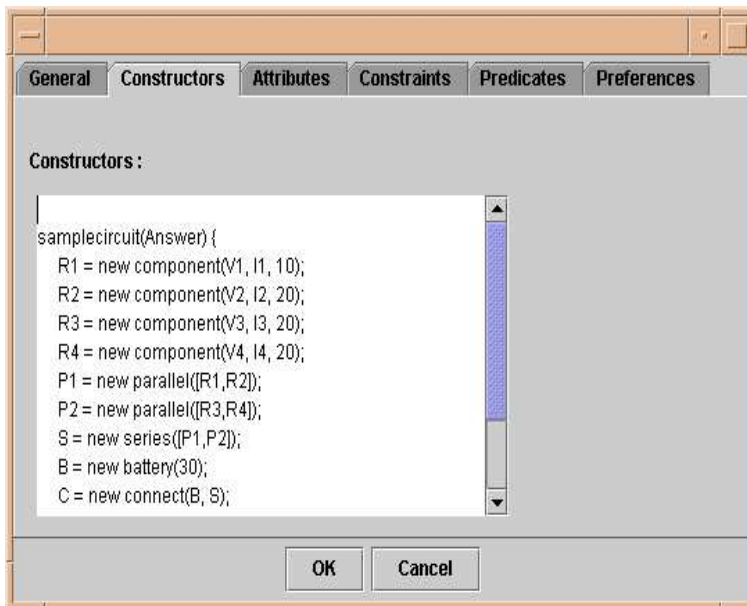


Figure 3.7: Popup Window through which the constructor of a class can be defined/edited.

A constrained object class diagram stores sufficient information for automatic generation of Cob code. The CUML tool described above can generate the Cob code corresponding to a class diagram. By clicking on the Tool button and selecting the `create Cob`

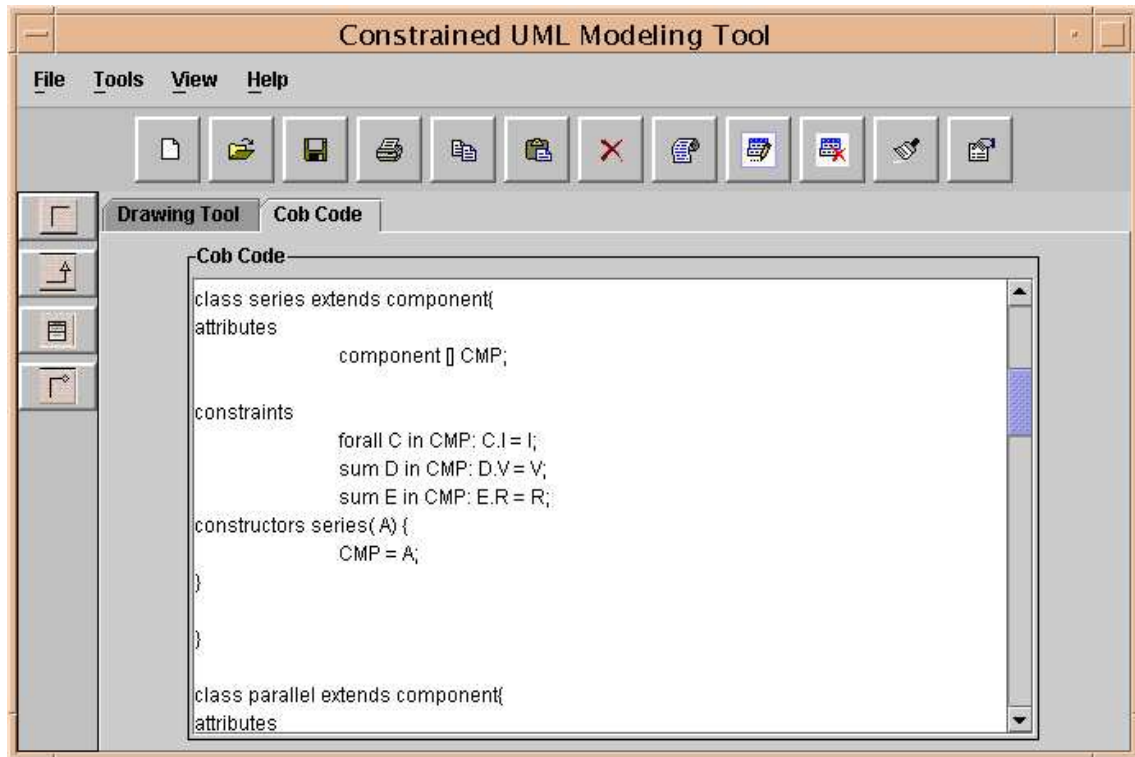


Figure 3.8: Snapshot of CUML tool showing the Cob code generated from the Class Diagram of Figure 3.5.

code option, the Cob code corresponding to the class diagram can be generated. This code can be viewed by clicking on the Cob code tab (see Figure 3.8). Thus, an entire Cob program can be written and edited via its class diagram using this tool. The user has the choice of viewing different aspects of the class diagram: in full detail showing the attributes and constraints of each class; only the classes and their relationships (this view is shown in Figure 3.5); or the textual Cob code (Figure 3.8). The textual Cob code generated from a class diagram can be viewed through this tool, but it cannot be edited directly. In order to change the generated Cob program/code, the original class diagram should be edited and then recompiled (using the `create Cob code` option to generate the modified code).

Once the complete class diagram is developed, the tool can be used to compile the diagram to generate the corresponding Cob code and subsequently, the `execute` button can be used to run the Cob code to obtain answers. The compilation and execution of the Cob code are described in detail in Chapters 4 and 5. The answers (values for variables) obtained by running the code are displayed back on the class diagram. The programmer can then

modify the class diagram, generate the corresponding modified Cob code, and recompile and re-execute the generated program any number of times to observe its behavior.

3.3.2 Domain-Specific Interfaces

We now describe the visual drawing interface for developing Cob models of trusses.

Cob Diagrams of Trusses We provide a domain-specific tool for modeling trusses. The visual aspects of this tool were implemented by Kapil Gajria, a recent graduate student of the Department of Computer Science and Engineering, University at Buffalo. This modeling tool provides a palette of buttons for creating instances of bars and loads. Member of a truss can be placed together at a joint by merging the ends of two or more bars. Figure 3.9 shows a truss drawn using the truss drawing tool. The interface has a predefined library of classes corresponding to each component (bar, load, joint) of the structure. Placing the icon of a component on the canvas creates an instance of the corresponding class. To create an instance of a beam, the user clicks on the beam icon. This pops open a window (Figure 3.10) through which the user can enter values for the attributes of a beam (e.g. its Young's modulus, yield strength, length, width, etc.) and any instance-level constraints on the attributes (e.g. force must be less than 100lbs). Any or all attributes may be left uninstantiated or undefined (i.e., specified by a variable). Every instance of a component is labeled with a default name.

Once the drawing is completed, the user can compile the drawing to generate the textual Cob code corresponding to the diagram. This is done by clicking on the `Compile` button from the `Tools` menu. Below is a part of the code generated for the truss in Figure 3.9.

```
class sampletruss {
  attributes
    beam AB, BC, CD, BD, AC; load IAV, IAH, ICV, IDV;
    .....
    joint JA, JB, JC, JD;
  constraints
    W1 = H1; W2 = H2; W3 = H3; W4 = H4; W5 = H5;
```

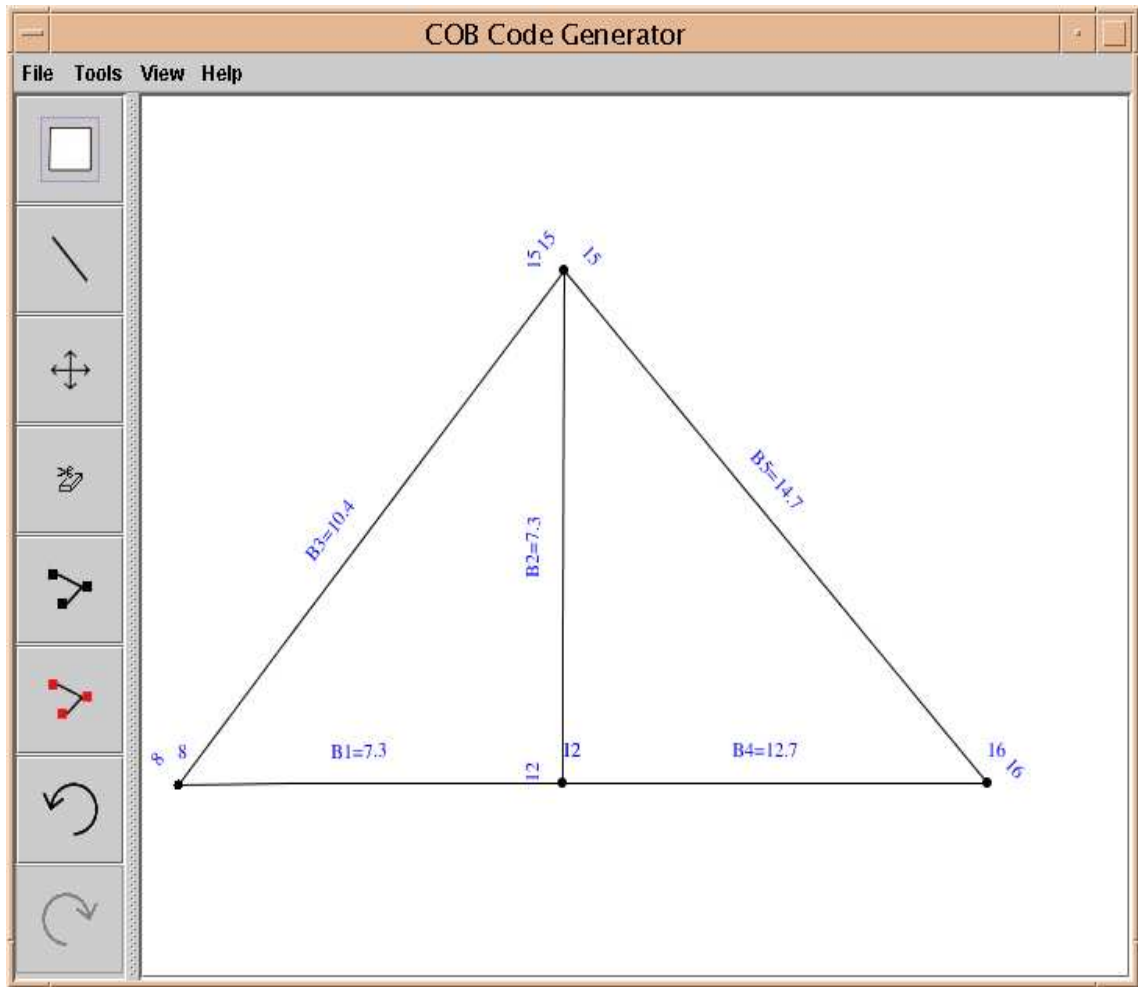
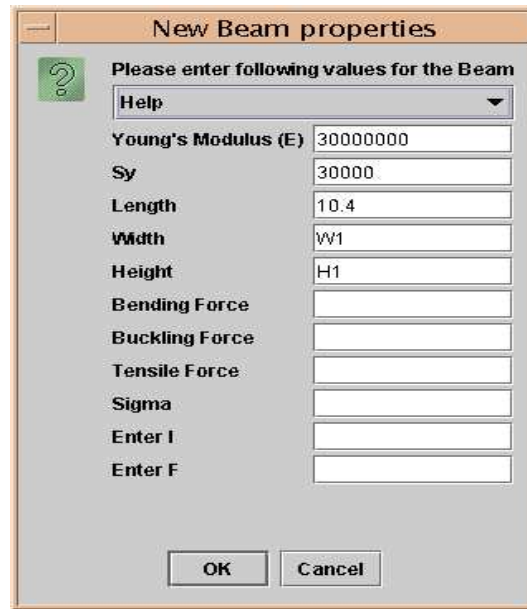



Figure 3.9: Snapshot of the Truss Drawing Tool showing a Sample Truss

```

.....
constructor sampletruss() {
    AB=new beam(Es,Sy,10.4,W1,H1,Fab.bn,Fab.bk,Fab.t,Sigab,Iab,Fab);
    BC=new beam(Es,Sy,7.3,W2,H2,Fbc.bn,Fbc.bk,Fbc.t,Sigbc,Ibc,Fbc);
    CD=new beam(Es,Sy,12.7,W3,H3,Fcd.bn,Fcd.bk,Fcd.t,Sigcd,Icd,Fcd);
    BD=new beam(Es,Sy,14.7,W4,H4,Fbd.bn,Fbd.bk,Fbd.t,Sigbd,Ibd,Fbd);
    AC=new beam(Es,Sy,7.3,W5,H5,Fac.bn,Fac.bk,Fac.t,Sigac,Iac,Fac);
    IAB = new bar(AB,Pi/4); IAC = new bar(AC,0);
    IAV = new load(Fav,Pi/2); IAH = new load(Fah,0);
    Ba = [IAB, IAC]; La = [IAV, IAH]; JA = new joint(Ba,La);

```



The image shows a 'New Beam properties' dialog box. It has a title bar with the text 'New Beam properties'. Inside, there is a green question mark icon and the text 'Please enter following values for the Beam'. Below this is a 'Help' button. The main area contains several labels and corresponding input fields: 'Young's Modulus (E)' with the value '30000000', 'Sy' with '30000', 'Length' with '10.4', 'Width' with 'W1', 'Height' with 'H1', 'Bending Force', 'Buckling Force', 'Tensile Force', 'Sigma', 'Enter I', and 'Enter F'. At the bottom are 'OK' and 'Cancel' buttons.

Property	Value
Young's Modulus (E)	30000000
Sy	30000
Length	10.4
Width	W1
Height	H1
Bending Force	
Buckling Force	
Tensile Force	
Sigma	
Enter I	
Enter F	

Figure 3.10: Popup Window for Entering Values for Attributes of a Beam

```

.....
}
}

```

By clicking on the Run button from the **Tools** menu, the generated Cob code is compiled and executed. The answers resulting from this execution are displayed on the diagram. The user can get different views of the structure: the diagram; the Cob code; the translated CLP(R) code; and the window showing the script of the execution. The modeler can click on any component of the diagram and get information about the instance, its input constraints and output constraints.

A Cob diagram can be saved and viewed at a later time. To modify an existing diagram, the modeler can use the mouse buttons to drag and drop parts of the diagram on the canvas or erase button to delete parts of the diagram. To modify the values of attributes of a component or change its constraints, the modeler can right click on the component to get its information displayed on a separate window and then edit the information through this window. The modeler can then click the Run button to view the results of the modified diagram.

3.4 Related Work

A number of languages have been developed by integrating objects with constraints and object-oriented programming with logic programming. While some of the integration is motivated purely from a programming language standpoint - to develop a language that has the advantages of both paradigms - some efforts are motivated by the application area, namely, interactive development of user interfaces, drawing and manipulation of geometrical figures, data integrity, etc. Motivations range from making object-oriented programming more declarative to making logic programming more modular to using the multi-programming paradigm for modeling problems from the engineering/physical/user-interface domains. This section gives an overview of several such languages, their motivation and their applications. In surveying the related work we categorize these languages according to their underlying programming paradigms.

3.4.1 Constrained Objects Languages

One way to make object-oriented programming more declarative is to provide constructs for explicit specification of relations that otherwise hold implicitly in imperative code. The purpose of making the relations explicit is to make the code more readable, less brittle to change and more reusable by making the patterns of interaction between objects easier to detect. Languages motivated by these reasons try to simulate imperative or procedural computation of object-oriented programming via automatic satisfaction of constraints and/or pre/post conditions. These languages give a declarative specification of the state of an object and hence are closely related to our work on constrained objects.

ThingLab. An early forerunner in the area of constrained objects is the work of Alan Borning on ThingLab [10]. This language is intended for interactive graphical simulations in physics and geometry, e.g, geometric objects like lines and triangles, electrical circuits, mechanical linkages, and bridges under load. ThingLab is an object-oriented language with constraints for specifying the relation between parts of an object. A constraint consists of a rule and a set of methods for satisfying the constraint. The rule is used to test how well the constraint is satisfied. Each method has a message plan that describes how to invoke

the method and specifies its side effects. The system uses this information, to decide on a constraint satisfaction plan when a state change occurs. The order of the methods indicates the user's preference for how the constraint should be solved and can be thought of as simulating constraint hierarchies. Classes can be defined by example, by creating prototypical instances that serve as the default instance of the class. Multiple inheritance of constraints along with method overriding is permitted. The user can view and edit simulations/objects through a graphical user interface similar to that of Smalltalk with a window provided for drawing and manipulating objects. The ThingLab system is built on top of Smalltalk and uses a constraint solver based on local propagation for constraint satisfaction. For cyclic numeric constraints, it uses relaxation, an iterative approximation algorithm. Although constraints can be stated explicitly within a class definition in ThingLab, the programmer has to provide the methods that will ensure that the constraints are satisfied. This is in contrast to the constraints provided in the Cob language, which are a declarative statement of relations between attributes and are not accompanied by any user defined algorithm for solving them.

Siri. The constrained object language Siri [55], developed by Bruce Horn integrates constraints into an object-oriented framework. Object encapsulation, inheritance, constraints and message passing are modeled using constraint patterns. A constraint pattern is of the form *Label: Prefix { Body } 'Type*; This creates an object called *Label* of type *'Type* which inherits from the objects listed in *Prefix* and *Body* describes the object's implementation. The *Body* consists of constraint expressions (constraints), imperative expressions or nested constraint patterns. Variations of this basic form of a constraint pattern can represent classes, instances, methods or control structures. Constraints define relationships between the attributes of an object. A relationship may be equality, arithmetic or user-defined. Methods are specified by declaring invariants, i.e., the constraints that must hold during the execution of a method, and by specifying which attributes must remain unchanged and which may change and how they should change during the method execution. Multiple inheritance is provided and is based on the BETA [69] prefixing mechanism for resolving name conflicts. Constraint patterns are an extension of the pattern in BETA. Encapsulation is preserved by allowing state change (externally) only through state change methods.

Constraints are also allowed only on ‘owned’ attributes (an object owns an attribute if it is its sole modifier). The Siri language is implemented in C and extends Bertrand [72], a language aimed at graphics applications. Siri uses Bertrand’s augmented term-rewriting for equation-solving and can handle limited types of constraints: simple linear equations and certain non-linear algebraic constraints over real numbers. Although Siri gives a uniform representation of object-oriented constructs through constrain patterns, it cannot handle the wide variety of features available in Cob, viz., conditional constraints and existentially quantified constraints, predicates, symbolic constraints and optimization.

Kaleidoscope. The Kaleidoscope ’91 [30] language integrates constraints and object-oriented programming for developing interactive graphical user interfaces. The main issue that it addresses is solving constraints over the types of variables as well as constraints over the values of variables. User-defined constraints are simplified until a primitive constraint solver can be used to solve the constraints over the primitive domain. The compiler tries to determine statically which constraints will be active at run-time, and uses this knowledge to produce a short sequence of instructions instead of a run time call to a constraint solver. There are three primitive domain constraint solvers, and provision is made so that they can communicate to solve inter-domain constraints. In Kaleidoscope ’93, constraints are solved according to their type: class/type constraints, identity constraints, or value/structure constraints. This ‘constraint imperative language’ uses constraints to simulate imperative constructs such as updating, assignment, and object identity [77, 75]. These are important issues in a constraint imperative language, but not for a declarative object-oriented language such as Cob. For the class of modeling applications that we target, it is not essential for us to consider such imperative concepts. In our modeling scenarios, *model execution* and the *model revision* are carried out in mutual exclusion of one another. Changes are made at the level of the modeling environment. Thus we have a clear separation of the declarative and procedural parts of a constrained object. Also, Kaleidoscope does not provide many features available in Cob such as conditional constraints and existentially quantified constraints, predicates, and optimization.

Modelica. Modelica [34] is a constrained object language for modeling and simulation of engineering structures. Modelica classes have a variety of numeric constraints, including quantified and the if-then-else form of conditional constraints, for modeling the behavior of the physical component a class represents. The underlying system uses Matlab [109] for constraint solving. Classes, inheritance, and aggregation are used to represent the structure of the physical system. Separate constructs are provided for performing imperative changes to the state. Modelica also provides visual tools for drawing models of different types of engineering structures through a graphical user interface. Although the constructs, visual interfaces, and applications of this language are very similar to our work, the constrained object paradigm that we present makes greater use of the power of constraint (logic) programming by providing the following features: optimization, answer constraints representing under specified models, constraint queries, logic variables, predicates, and symbolic constraint solving. Our novel scheme for implementation of constrained objects (described in Chapters 4 and 5) us to define formal semantics for constrained objects. No other constrained object language provides such rigorous formal semantics. Also, our implementation techniques (described in Chapter 5) facilitate the development of unique interactive execution and visual debugging interfaces based on the underlying object structure of a Cob model.

Other constrained object languages include ASCEND IV which is an equation-based environment featuring a strongly-typed, object-oriented model-description language. It has interactive support tools for modeling, debugging, and solving systems with nonlinear algebraic or differential equations. In addition to constraints , it also allows imperative methods to be written within the model definitions. However, ASCEND does not deal with non-mathematical constraints (an example of such constraints in Cob is given in the document layout example in Chapter 7). Another constrained object language (also named COB) [90] is intended for analyses integration and automated design. It handles numeric constraints but does not appear to provide logic variables, conditional constraints, aggregate constraints or preferences.

Compared to Cob, the above languages provide only a limited capability for expressing constraints, and also provide no support for handling multiple solutions to constraints. Al-

though ThingLab has a preference construct that orders constraint solving methods, it does not address the more general problem of optimization. The Cob language and modeling environment facilitates a systematic approach to modeling engineering systems by providing a rich set of features including arithmetic, symbolic, non-linear constraints, quantified and conditional constraints, preferences as well as visual interfaces for developing Cob models.

3.4.2 Constraints, Logic and Objects

The use of constraints for problem modeling can be traced to the work on Sketchpad by Sutherland [108]. Sketchpad is a general-purpose system for drawing and editing pictures on a computer based on the notion of constraints. The system used local propagation for constraint solving and had multiple cooperating solvers. Leler's Bertrand language and system integrated term-rewriting with linear constraint solving and was aimed at graphics applications [72]. A major breakthrough was achieved when Jaffar and Lassez integrated logic programming with constraint solving in their CLP family of languages [59]. Other important developments in the area of constraint programming include van Hentenryck's work on incorporating finite-domain constraint satisfaction techniques into Prolog [49]; and Saraswat's work on the family of concurrent constraint languages [99]. Research at the UNH Constraints Computation Center under Freuder is related to our efforts. From a language standpoint, there are two important differences: (i) we integrate the concepts of object and constraint, and (ii) we adopt the more general CLP paradigm as opposed to a pure constraint language. Our approach to constraints and preferences builds upon the work on PLP given in [38, 40] which subsumes the paradigms of CLP as well as HCLP (Hierarchic CLP) [40]. Firstly, CLP does not support conditional constraints or preferences. Also, our provision of conditional constraints allows object creation to take place dynamically and be controlled by constraint satisfaction.

Efforts directed at making logic programming more modular try to give a notion of state to logic programs to increase their (re)usability. By providing a multi-paradigm programming language, they hope to increase understandability of a complex software system, its synthesis and its analysis. They aim at extending logic programming for flexible structuring, sharing and reuse of knowledge in large logic programming applications. We give

below a survey of work in multi-paradigm programming involving constraint, logic and object-oriented programming that is relevant to our work on constrained objects.

Objects as Intensions [17]: provides an intensional semantics for objects from a logical perspective. First-order logic is extended with the concepts of intensional variables, and static and dynamic predicates. Intensional variables represent the state changing variables found in object-oriented programming and their value can be retrieved or updated using the static or dynamic predicates respectively. When state changes, only the value of the variable being updated changes, all other variables continue to have the same value as in the previous state. This is referred to as the frame assumption. Every object is associated with an intension which is a mapping from states to values. An object can have different values in different states and hence the meaning or behavior of an object instance is given by the entire history of its state (i.e., a sequence of state-value pairs). The main issue addressed by this work is the semantics of state change in the logic programming context. This differs from our approach to semantics of constrained objects described in Chapter 4. We provide a set-theoretic semantics for constrained objects which is based upon a translation to CLP. Essentially, the meaning of constrained object class is the set of values of its attributes that satisfies the constraints of the class. We do not focus on state change but on the set of valid states with respect to the constraints of a class.

Logical Objects [20]: is a scheme for modeling objects using first-order logic. There are two categories of literals, one representing objects and the other representing procedures. The former when present as the head of a Horn clause define the structure (attributes and possibly their values) of a class. When used with procedural literals in the head of (an extension of) Horn clauses, they define the methods of a class. Each object literal has an extra argument which refers to a unique instance of a class and such references are static, i.e., they can be referred to by other clauses or procedures. The state of an object is stored in an internal database within the system and state change is provided at the programming level by associating a different structure with the same reference internally. At the programming level, logical objects is an extension of Prolog-like syntax to simulate the concepts of objects, encapsulation, state change, and modularity, and tries to incorporate procedural behavior

into a logic programming framework. It does not deal with the notion of constraints. This is unlike Cob which extends the syntax of a traditional object-oriented language to provide constraints and logic predicates at the modeling level with focus not on procedural methods or state change, but on declarative specification of the state or behavior of objects through constraints.

SICStus Objects [104]: is a module of SICStus Prolog which provides object as a named collection of predicate definitions. In this sense an object is similar to a Prolog module. The object system can be seen as an extension of SICStus Prolog's module system. In addition an object may have attributes that are modifiable. Predicate definitions belonging to an object are called methods. So an object is conceptually a named collection of methods and attributes. Some of the methods defined for an object need not be stored explicitly within the object, but are rather shared with other objects by the inheritance mechanism. Although SICStus has separate constraint libraries for CLP(R), CLP(Q), and CLP(bool), unlike Cob, constraints cannot be stated directly as a part of a class definitions but only as a part of the methods and hence do not automatically come into play when an instance is created.

OLI [71]: is an approach to integrating logic with object-oriented programming and stands for the Object Logic Integration design methodology which allows programming in either or mixed paradigm. From the object point of view, the logic part of OLI is an object with logic programs as states and methods performing logical deduction. The mixed paradigm allows the usual class definitions and methods can be written as queries. However, no notion of constraints is supported by this language.

LyriC [12]: can store relations between objects in a database as constraints. Constraints are treated as first class objects: a constraint itself is an object, and can have attributes and methods and operations (to attach additional information and manipulate constraints). A constraint object can be an attribute of another object and this attribute can in turn be part of another constraint. Constraints can be used in a query to filter stored constraints and to create new constraint objects. Unlike the wide variety of constraints provided in the Cob programming language, LyriC, which is essentially a database querying language, supports

only linear equality or inequality constraints.

CHIC [89]: maintains coherence of a composite object model using the concept of constraints. Constraints exist outside of the objects. A constraint stating a relation between two or more classes is defined separately as a subclass of the predefined class *Constraint*. The relation is then stated inside of this class definition. Thus a constraint itself is an object and the constraint manager performs maintains consistency or coherence of these constraints by using a perturbation model. In this model an application is initially coherent, and when a variable is modified, it is the task of the system is to compute a new coherent application by generating a plan (order) for solving the constraints on different classes. Constraints are kept outside of objects to provide abstraction. However, we feel that writing constraints within class definitions as provided in Cob is a more natural way of describing the behavior of an object. This increases the readability of the code and with suitable visibility modifiers can maintain the desired level of abstraction.

Logic++ [116]: is an effort at making the methods in object-oriented programming more declarative. A Logic++ class contains methods which are Prolog Horn clauses while the rest of the class is similar to a C++ class. The compiler translates these methods to C++ functions. Unlike Cob, this multi-paradigm programming language does not support constraints.

Prolog++ [87]: extends logic programming by allowing storage and modification of the state of a system by providing constructs for classes, methods and assignment. A Prolog++ program is compiled into standard Prolog and does not have any constraint solving capabilities.

Two other related logic-based languages are LIFE [4] and Oz [106]. LIFE combines logic with inheritance, however, it does not deal with full-fledged constraints as we do. Oz is a language combining constraints and concurrency as well as objects. Both these languages do not support the notion of preference nor do they consider engineering applications.

3.4.3 Constraint-based Specifications

For the sake of completeness, we also mention software specification languages that make use of constraints.

OCL. The need for a formal representation of constraints in object-oriented programming is illustrated by the development of the Object Constraint Language [110]. For practitioners of object-oriented design and analysis, constraints provide an unambiguous and concise means for expressing the relations between objects. OCL is a specification language that helps in making explicit these relations that would otherwise be implicit in the code and not apparent to the programmer who reads or modifies it.

Eiffel. Eiffel is another language which employs constraints for specifying pre- and post-conditions that must hold on the operations of an object [85]. These languages use constraints as specifications; no constraint solving is done at run-time in order to deduce the values of variables.

Contracts. Contracts [47] provide a formal language for specifying behavioral compositions in object oriented systems. A contract defines a set of communicating participant classes and their contractual obligations as constraints. A class conforming to a contract must implement the methods exactly as specified in the contract. Contracts promote an interaction-oriented design rather than a class-based design.

There are several modeling tools for engineering structures which have a graphical drawing front end. These are not programming languages, but a method of creating a model of a structure through a GUI. The representation of the structure and checking its validity are done separately, and usually there is no provision for representing the results back onto the drawing. Nonetheless, these are highly specialized and useful drawing tools intended for restricted fields within the engineering domain. In general, the more specialized an application/language, the more restricted its use. This is usually because the more specific the problem domain, the more tractable the problem.

3.5 Summary

We have described in detail the syntax of the Cob language and given several examples of Cob programs that illustrate its syntax and application to engineering modeling. The examples demonstrate the use of several different important features of the Cob language: constraints, quantified constraints, conditional constraints, inheritance, aggregation, polymorphism, predicates, etc. Each example illustrates that the constrained object paradigm is not only better than a pure object-oriented language but also facilitates building modular, more intuitive and easier to understand models of engineering structures than those built using pure constraint languages such as CLP(R).

We have described two types of visual interfaces for the development of Cob models. The visual interface for drawing Cob class diagrams facilitates rapid development of Cob code in which the programmer specifies the high level object-oriented design of a Cob program in terms of constrained objects classes and their relationships with each other. This interface can be used to generate Cob code as well as execute it to obtain results. The domain specific visual interfaces for drawing engineering structures are a convenient, intuitive, and high level tool of developing Cob models of engineering systems. They are specialized for a particular domain with built-in library of classes for the engineering structures in that domain. In this way a modeler can focus on building the model of a structure and analyze it without having to understand the underlying constrained object model.

The concept of inheritance, gives rise to several other object-oriented features such as visibility modifiers for attributes, multiple inheritance, interfaces, constraint overriding, etc. Most of these features are conceptually possible in the constrained object paradigm. For example, if multiple inheritance is permitted, then a class will inherit the constraints of all its parent classes. The inheritance of constraints as well as attributes can be controlled by visibility modifiers (e.g. private, public, protected etc.). Constraints can be labeled and subclasses may be permitted to override the constraints of the parent class. Although it is possible to incorporate these features into the constrained object paradigm, the current prototype Cob language does not support all of them. Their omission from the current prototype Cob language is not an inherent limitation in the language. They can be included

in Cob in much the same way that they are used in imperative languages. The development of Cob and its constructs has been guided by modeling problems in the engineering domain.

Chapter 4

Semantics of Constrained Objects

Formal semantics of a programming language provide a framework for studying (properties of) classes of programs in that language. The mathematical model of a language provided by its formal semantics can be used to reason about programs, including their analysis, verification, computability, etc. In the context of modeling complex systems, for example, formal semantics of constrained objects can be used to determine if a Cob model of a system does indeed model its behavior. Formal semantics of a language typically have two parts: the *declarative semantics* define the meaning of a program; and the *operational semantics* provide a scheme for execution and can be used as a basis for developing a compiler or interpreter for the language.

Different techniques have been used to define declarative semantics, e.g., logical, algebraic, denotational, axiomatic, etc. [60, 107, 114]. In traditional object-oriented languages, a class defines an abstract datatype whose behavior is characterized *procedurally*, i.e., by its operations. Cob classes also define abstract datatypes but their “behavior” is characterized *declaratively*, i.e., by the constraints on the data. We give a set-theoretic semantics for Cob datatypes using an approach that can satisfactorily account for recursively defined classes. We translate a Cob class c into a CLP-like predicate definition p_c , and define the semantics of the class c in terms of the least model of the predicate p_c . A Cob program C is thus translated to a CLP-like program P by translating each class in C to a corresponding predicate in P . The top-level call on a constructor class is equivalent to a top-level goal or query. The semantics of C are given in terms of the least model of the program P . However,

a conditional constraint is an important construct that cannot be directly mapped to CLP. This is because CLP semantics do not provide any means to determine the validity of a constraint or atom, they can only check for its consistency with the state of a computation. We describe a technique for handling conditional constraints in this chapter.

The material in this chapter is organized as follows. In Section 4.1, we first give an overview of the declarative semantics of constraint logic programs; then we describe a scheme for the translation of Cob programs to CLP programs; and finally we use this translation to define the declarative semantics of Cob programs. In Section 4.2 we give an overview of the operational semantics of CLP programs; then we define the operational semantics of Cob programs followed by their soundness result. We formulate the completeness result for the operational semantics of Cob programs with a brief sketch of our intended approach for its proof.

4.1 Declarative Semantics

The declarative semantics of Cob programs are based upon a translation to CLP and build upon some of the concepts from CLP semantics. Therefore, in order to understand the declarative semantics of Cob programs, an understanding of the declarative semantics of CLP programs is required. Hence, before defining the declarative semantics of Cob programs, we first describe the formal declarative semantics of CLP programs as given in [60]. In Section 2.2 we gave an overview of the constraint logic programming language CLP(R) in which the constraints are over the real number domain \mathbb{R} . In general, the CLP scheme defines a family of languages parameterized by the domain of the constraints and the solver for that domain. A constraint domain refers to a set with operations and boolean relations, e.g., real numbers with multiplication and addition, and relational operations such as inequality and equality. Certain properties of the constraint domain are known to the constraint solver, e.g., associativity of addition, distributive property of multiplication over addition, etc. The operators and relations together define the primitive constraints of the domain. In the CLP paradigm, given a constraint domain, one can define constraints/predicates in terms of the underlying constraint domain and primitive constraints.

Answers to queries are computed using rule-based evaluation and constraint solving. The formal definitions of the above notions and the formal declarative semantics of a CLP program are described in [60] and we give an overview of this material in the next section.

4.1.1 Constraint Logic Programs

The definitions and description of the declarative semantics of CLP languages presented in this subsection is summarized from [60] which gives a more detailed and thorough discussion. The $\text{CLP}(C)$ scheme is parameterized by the constraint domain $C \equiv \langle \Sigma, \mathcal{D}, \mathcal{L}, \mathcal{T}, \text{solv} \rangle$ where:

- Σ is the **signature** which consists of a set of function and predicate symbols and their respective arities. The set of function and predicate names appearing in a CLP program is a subset of its signature.
- \mathcal{D} is a Σ -**structure** which consists of a set D and functions and relations over D that correspond to the function and predicate symbols in Σ with corresponding arity. A Σ -structure is an interpretation of Σ , i.e., it gives a meaning to the symbols of the signature. \mathcal{D} can be thought of as the domain of computation.
- \mathcal{T} is a Σ -**theory** which is a (possibly infinite) set of closed first order Σ -formulas. A first-order Σ -**formula** is a logical formula constructed from variables and the function and predicate symbols of Σ using logical connectives $\wedge, \vee, \neg, \rightarrow$ and quantifiers \exists and \forall . \mathcal{T} represents the set of known properties of the constraint domain.
- \mathcal{L} is a subset of first-order Σ -formulas that represents the **primitive constraints** of the language. The primitive constraints, referred to as explicit constraints in the syntax of $\text{CLP}(\text{R})$ in Section 2.2.1, are basically the set of constraints that are permitted in the language.
- solv is a solver for \mathcal{L} . The solver solv defines a mapping from the set of formulae in \mathcal{L} to either *true*, *false* or *unknown*.

Thus the constraint domain defines the constraints permissible in the language, their domain and properties as well as the solver. The following assumptions are made about

the constraint domain. It is assumed that Σ has the binary predicate symbol “=” which is interpreted as the identity relation in \mathcal{D} and \mathcal{T} contains the standard equality axioms. Also, \mathcal{L} is assumed to contain all atoms constructed from $=$. The solver is assumed to ignore variable renamings, i.e., $\text{solv}(c) = \text{solv}(\rho(c))$ where ρ maps distinct variable names occurring in c to some other distinct variable names. Finally, it is assumed that the domain of computation \mathcal{D} , solver solv and constraint theory \mathcal{T} are in agreement, i.e., \mathcal{D} is a model of \mathcal{T} and for any constraint $c \in \mathcal{L}$, if $\text{solv}(c) = \text{false}$ then $\mathcal{T} \models \neg \exists c$ and if $\text{solv}(c) = \text{true}$ then $\mathcal{T} \models \exists c$, where $\exists c$ denotes the existential quantification of all the free variables occurring in c . In the rest of this section, symbol \mathcal{C} will refer to the constraint domain $\langle \Sigma, \mathcal{D}, \mathcal{L}, \mathcal{T}, \text{solv} \rangle$.

Consider as examples, the following two constraint domains. The constraint domain *Real* has the relation symbols $\leq, \geq, <, >, =$, the function symbols $+, -, *, /$, and constants which are sequences of digits including atmost one decimal point per sequence. The intended interpretation or the domain of computation of *Real* is the set of real numbers \mathbb{R} . The primitive constraints ($\leq, \geq, <, >, =$), the function symbols and constants have the usual arithmetic meaning. The theory is a theory for *Real* given in [102]. An implementation of a solver for *Real* can be based on Gauss-Jordan elimination [63]. The constraint domain *Term* has the relation “=” forming the primitive constraints and alphanumeric character strings as the function symbols and constants. $\text{CLP}(\text{Term})$ forms the traditional logic programming language such as Prolog. A solver for this domain implements a unification algorithm.

Before giving the declarative semantics we give some useful definitions from [60].

Definition 4.1: A **model** of a Σ -theory \mathcal{T} is an interpretation of Σ in which all the formulae in \mathcal{T} evaluate to true. A **\mathcal{D} -model** of a Σ -theory \mathcal{T} is a model of \mathcal{T} that extends (is a superset of) a Σ -structure \mathcal{D} .

A theory \mathcal{T} of a constraint domain $\langle \Sigma, \mathcal{D}, \mathcal{L}, \mathcal{T}, \text{solv} \rangle$ is said to be **satisfaction complete** if for every constraint $c \in \mathcal{L}$, either $\mathcal{T} \models \exists c$ or $\mathcal{T} \models \neg \exists c$. This means that every constraint c in this domain and theory can be deduced to be either true or false. A solver solv is said to be **theory complete** if $\text{solv}(c) = \text{false}$ iff $\mathcal{T} \models \neg \exists c$ and $\text{solv}(c) = \text{true}$ iff $\mathcal{T} \models \exists c$.

We have defined the syntax of CLP(R) programs in Section 2.2.1. The syntax of a general CLP(C) program is similar except that the syntax of constraints may be different for each constraint domain. We recapitulate the syntax of CLP programs briefly here. A CLP program is a set of rules of the form $H : - B$ where H is called the head of the rule and B is the body of the rule. The head consists of an atom and the body is a collection of literals. A literal is an atom or a primitive constraint. An atom is of the form $p(t_1, \dots, t_n)$ where p is an n -ary user-defined predicate symbol and t_1, \dots, t_n are terms of the constraint domain. A term is either a variable or a constant or of the form $f(t_1, \dots, t_n)$ where f is an n -ary function symbol and t_1, \dots, t_n are terms. A goal is a conjunction of literals.

Definition 4.2: A C -interpretation of a CLP(C) program P is an interpretation that agrees with \mathcal{D} , i.e., it gives the same meaning to the function and predicate symbols of the signature of C as \mathcal{D} does.

Since the meaning of primitive constraints is fixed by C , it suffices to regard a C -interpretation as a subset of

$$C\text{-base of } P \equiv \{p(d_1, \dots, d_n) \mid p \text{ is an } n\text{-ary user defined predicate in } P \text{ and each } d_i \in D\}$$

The logical reading of a CLP program rule of the form

$$p(\bar{t}) : - q_1(\bar{t}_1), \dots, q_n(\bar{t}_n)$$

is given by the following Σ -formula

$$\tilde{\forall} p(\bar{t}) \leftarrow q_1(\bar{t}_1) \wedge \dots \wedge q_n(\bar{t}_n)$$

where the symbol $\tilde{\forall}$ indicates the universal quantification of all the free variables occurring (in the formula) on its right hand side. In other words, the truth of the body of a rule implies the truth of the head of the rule. Thus the logical semantics of a CLP(C) program can be understood as the theory \mathcal{T} appended with the logical formulae corresponding to each of the program rules.

Definition 4.3: A C -model M of CLP(C) program P is a C -interpretation of P such that for every rule of P of the form

$$p(\bar{t}) : - q_1(\bar{t}_1), \dots, q_n(\bar{t}_n)$$

if $q_1(\bar{d}_1), \dots, q_n(\bar{d}_n) \in M$, then $p(\bar{d}) \in M$ where $d_i \in \mathcal{D}$ and there exists a valuation ρ such that $\rho(\bar{t}) = \bar{d}, \rho(\bar{t}_1) = \bar{d}_1, \dots, \rho(\bar{t}_n) = \bar{d}_n$.

Thus, a model M of a program P is a subset of the C -base of P such that whenever a ground instance of the body of a rule of P belongs to M , the corresponding instance of the head of the rule also belongs to M . This in effect means that the logical semantics of every rule is a formula which is always true.

The intersection of two models of a program is also a model of the program. The proof of this proposition is quite straightforward [60]. The intersection of all the C -models of a CLP(C) program P is also a C -model of the program [60] and is referred to as the **least model** $lm(C, P)$. Every CLP(C) program has a least model [60]. Thus an instance of any user-defined predicate whose truth is not implied by the program is not present in the least model of the program. The meaning of a CLP(C) program is defined in terms of its least model.

Definition 4.4: *The **declarative semantics** of a CLP(C) program P is defined as its least model M_p .*

The least model is thus the intended interpretation of a CLP(C) program as well as for the underlying primitive constraints and function symbols. Given a CLP(C) program P and a goal G , we are interested in finding an instance of the G that is implied by the program.

Definition 4.5: *Given a CLP(C) program, a valuation θ is a mapping from the variables of the program to values in the constraint domain C , i.e., values in D .*

Given a goal, we are interested in those valuations that make the goal true in the intended interpretation of the program.

Definition 4.6: *The **correct answer** to a goal G is a valuation σ such that $(G \sigma) \subseteq M_p$.*

We have described the declarative semantics of CLP(C) programs and goals. We will use these semantics as a basis to define the declarative semantics of constrained objects.

4.1.2 Translation of Constrained Objects to CLP

A Cob program C is translated to a CLP-like program P by translating each class in C to a corresponding predicate in P . The top-level call on a class constructor is equivalent to a top-level goal or query. We describe this translation for the key components of a Cob program. We first consider Cob programs without conditional constraints.

Class. We use V_i , C_i and P_i to denote collections of variables, constraints, and predicates respectively. For every class c in a Cob program C , we define a predicate p_c with two arguments: the first is a list of the arguments of the constructor of c , and the second is the list of attributes of class c . For predicates corresponding to abstract classes, the first argument is a null list. For classes with more than one constructor, one predicate is defined per constructor.

<pre>class c { attributes V₁ constraints C₁ predicates P₁ constructor c(V₂) { C₂ } }</pre>	\Rightarrow_T	$P_1 \cup \{defn(p_c)\}$ where $defn(p_c)$ is: $p_c(V_2, V_1) :- C_1, C_2.$
---	-----------------	---

If class c' is a subclass of c , then the second argument of predicate $p_{c'}$ is the list of attributes of c' prepended with V_c , which is the list of attributes of the parent class c as well as the attributes of the superclass of c (if any) and so on recursively. The body of predicate $p_{c'}$ invokes predicate p_c on V_c in order that the constraints of the parent class are met by the corresponding attributes of the subclass.

<pre>class c' extends c { attributes V'₁ constraints C'₁ predicates P₂ constructor c'(V'₂) { C'₂ } }</pre>	\Rightarrow_T	$P_2 \cup \{defn(p_{c'})\}$ where $defn(p_{c'})$ is: $p_{c'}(V'_2, V_c \circ V'_1) :-$ $p_c(---, V_c), C'_1, C'_2.$
---	-----------------	--

The predicates in a Cob class remain unchanged under the translation. To maintain encapsulation and prevent name clashes, predicate names within a class are suitably renamed during translation.

Variable Renaming. In an inheritance hierarchy, a subclass may redeclare a variable of its superclass. Our approach to handling variable redeclaration involves a systematic renaming of variables in subclasses to provide variable over-riding. Suppose class C_1 is a superclass of C_2 and that class C_i is a superclass of C_{i+1} for all $i \in 2..n-1$. Suppose that C_1 does not extend any class and C_n is not extended by any class. C_1, \dots, C_n is called a chain of hierarchy and its variables are renamed using the algorithm shown in Figure 4.1.

```

 $C_1, \dots, C_n$ : forms a chain of hierarchy.
 $declares(C_j)$ : set of variables declared within the definition of class  $C_j$ .
 $uses(C_j)$ : set of variables used but not declared within the definition of class  $C_j$ .
 $subst(X, Y, C)$ : substitute all occurrences of variable  $X$  with variable  $Y$ 
                    in definition of class  $C$ .
 $renamed(X, Y, C)$ : true if  $subst(X, Y, C)$  has already been performed, false otherwise.

rename_variables()
  for  $i \in 1..n$  {
    for  $X \in declares(C_i)$  {
      if  $X \in \cup_{k=1}^{i-1} declares(C_k)$  then {
         $subst(X, NewX, C_i)$ ;
         $renamed(X, NewX, C_i) := true$ 
      }
    }
    for  $X \in uses(C_i)$  {
       $m = \max\{k \mid 1 \leq k \leq i-1, X \in declares(C_k)\}$ 
      if  $renamed(X, Y, C_m)$  then
         $subst(X, Y, C_i)$ 
    }
  }

```

Figure 4.1: Algorithm for renaming variables

In the chain of hierarchy C_1, \dots, C_n , if C_i declares a variable X which is also declared

in one of the classes C_1, \dots, C_{i-1} , then X and all of its occurrences in C_i are renamed to a new variable name. If C_i uses a variable Y without declaring it, then we inspect the nearest class C_m (with the highest value for m less than i) that declares Y . If Y in C_m was renamed to $NewY$, then all occurrences of Y in C_i are replaced with $NewY$. If Y in C_m was not renamed, then Y in C_i is also not renamed. The above algorithm is executed for every chain of hierarchy of classes in a Cob program. Thus when a subclass redeclares a variable, it over-rides the variable declaration of its superclass.

The translation of a Cob class to a CLP predicate involves translation of its constructors, constraints and terms. Below we give the high level description of this translation; the finer details are given in Chapter 5 along with an implementation of the compiler.

Constraint Atom. A list of constraints is translated by translating every element of the list. A constraint atom is translated to an explicit CLP constraint. Since the underlying constraint domain remains the same, the relational operator is left unchanged and only the constituent terms of the constraint atom are translated.

$$\begin{aligned} \mathcal{T} \text{ relop } \mathcal{U} &\implies_T \mathcal{T}_T \text{ relop } \mathcal{U}_T \\ \text{where } \mathcal{T}_T &\text{ is obtained by translating term } \mathcal{T} \\ \mathcal{U}_T &\text{ is obtained by translating term } \mathcal{U} \\ \text{and } \text{relop} &\text{ is a relational operator.} \end{aligned}$$

Translation of terms is required since they might use array indexing or selection operations which are not directly available in CLP. An appropriate translation of such operations ensures that the translated CLP constraint compares corresponding equivalent terms. The details of the translation of terms is given in Chapter 5.

Constructor. A constructor invocation in Cob is translated to a CLP goal as shown below. A call to the constructor of class c is translated to an invocation of the predicate p_c which is obtained by translation of the class c . The list of constructor arguments form the first argument of the call to p_c and the variable being equated to the newly formed object of class c becomes the second argument.

$$\begin{aligned} X = \text{new } c(\bar{t}) &\implies_T p_c(\bar{t}_T, X_T) \\ \text{where } p_c &\text{ is the translation of class } c \end{aligned}$$

```

class component {
    attributes    real V, I, R;
    constraints V = I * R;
    constructors component(V1,I1,R1){
        V = V1; I = I1; R = R1;
    }
}

class parallel extends component {
    attributes component [] PC;
    constraints
    forall X in PC : X.V = V;
    (sum Y in PC : Y.I) = I;
    (sum Z in PC : 1/Z.R) = 1/R;
    constructors parallel(B) {
        PC = B;
    }
}

```

Figure 4.2: Cob Class Definitions

X_T is the translation of X

and \bar{t}_T is the translation of \bar{t}

Thus we represent a Cob class definition by a predicate definition, and an object or instance of class C is represented as a list of variables corresponding to the attributes of C . For this translation of constructor calls to be meaningful, it is required that in classes with more than one constructor, each of the constructors should be of a different arity. An important point to note is that any or all of the arguments to the constructors may be non-ground (unbound) variables.

To illustrate the above translations with a concrete example, consider the Cob classes shown in Figure 4.2. Class `component` models an electrical component with attributes

```

Pcomponent([V1, I1, R1], [V, I, R]) :-
    V = I * R,
    V = V1,
    I = I1,
    R = R1.

Pparallel([B], [V, I, R, PC]) :-
    PC = [_],
    PC = B,
    Pcomponent(_, [V, I, R]),
    forall1(PC, X, V),
    N1 = I,
    sum1(PC, N1, Y, I),
    N2 = 1/R,
    sum2(PC, N2, Z, R).

```

Figure 4.3: Translated CLP Program

voltage(V), current(I), and resistance(R), and a constraint that expresses Ohm's law ($V=I \cdot R$). Class `parallel` represents a collection of components(PC) connected in parallel.

Using the translation scheme described above, the Cob classes in Figure 4.2 are translated to the CLP predicates shown in Figure 4.3. The list of arguments to the constructor of class `component` become the first argument of the corresponding CLP predicate `pcomponent`; the list of attributes of the class become the second argument of the predicate. The constraints of the class become the constraints in the body of the corresponding predicate. A similar translation of arguments is done for the `parallel` class. The translation of quantified constraints and quantified terms is done by defining recursive predicates that iterate over the enumeration and whose definition captures the body of the quantified constraint. In the `parallel` class the quantified constraints and terms are translated to calls on the `forall1`, `sum1` and `sum2` predicates. The definitions of these predicates will be shown later in Chapter 5 after we give a detailed description of the translation of each type of constraint and term.

4.1.3 Constrained Object Programs

Like CLP programs, constrained object programs are parameterized by their constraint domain. The constraint domain defines the kinds of constraints permitted in the language, their properties and the set of permissible values for their variables. Given a constraint domain $\mathcal{C} \equiv \langle \Sigma, \mathcal{D}, \mathcal{L}, \mathcal{T}, \text{solv} \rangle$, we define a class of constrained object programs with the syntax described in Section 3.1. where \mathcal{L} can be thought of as defining the grammar rules for the non-terminal *constraints* in Section 3.1., Σ contains the predicate and function names appearing in a class definition, \mathcal{T} defines the properties of the constraints, \mathcal{D} is the domain of the variables appearing in the constraints, and *solv* is the solver for the constraints.

The semantics of Cob programs without conditional constraints can be understood directly in terms of the semantics of their CLP-translation (using the scheme given in Section 4.1.2). In the rest of this chapter and dissertation, the underlying constraint domain \mathcal{C} for Cob will be assumed to be the tuple $\langle \Sigma, \mathcal{D}, \mathcal{L}, \mathcal{T}, \text{solv} \rangle$ and, for the sake of brevity, in the discussion below Cob(\mathcal{C}) programs will be referred to simply as Cob programs.

Definition 4.7: *Let C be a Cob program. The CLP program P obtained by translating each class of C by the scheme described in Section 4.1.2 is called the **CLP-translation** of C . For each class c in C , the predicate p_c obtained by translating c is called the CLP-translation of c . For a goal $G \equiv X =_{\text{new}} c(\bar{t})$, the query $p_c(\bar{t}, X)$ obtained by using the translation in Section 4.1.2 is called the CLP-translation of G .*

Given a Cob program with a constraint domain \mathcal{C} , the underlying constraint domain of its corresponding translated CLP program will be a slight extension of \mathcal{C} . This is because the predicate symbols generated by the translation will have to be accommodated in the underlying signature, structure and interpretation. That said, for simplicity, we will continue to refer to this augmented constraint domain as \mathcal{C} .

Intuitively, we would like to define the semantics of a class as the set of values for its attributes that satisfies the constraints of the class. In general, when a Cob class is translated to a CLP predicate, its attributes form a part of the arguments of the predicate, and the constraints of the class become the constraints in the body of the predicate. The instances of a class correspond to ground instances of the corresponding predicate, and the

set of possible instances of the class naturally become the set of ground instances of the corresponding predicate that are present in the intended model of the CLP program. The declarative semantics of a Cob program can therefore be given in terms of the least model of its CLP-translation.

Definition 4.8: *Let C be a Cob program and P be its CLP-translation. The **declarative semantics** of C is the least model M_p , of P . More specifically, the set-theoretic semantics of each class $c \in C$ is defined as $M_c = \{\bar{t}_2 \mid p_c \text{ is the CLP-translation of } c \text{ and } p_c(\bar{t}_1, \bar{t}_2) \in M_p, \text{ for some } \bar{t}_1, \text{ where } \bar{t}_1, \bar{t}_2 \in \mathcal{D}\}$.*

For example, consider the program P defined earlier in Figure 4.2 with classes `component` and `parallel`. The declarative semantics of the `component` class is the set of all instances of the tuple $[V, I, R]$ such that the constraint $V = I * R$ is true. Thus

$$M_{\text{component}} = \{[V, I, R] \mid \exists V1, I1, R1 \in \mathcal{R}, p_{\text{component}}([V1, I1, R1], [V, I, R]) \in M_p\}.$$

The creation of a Cob model of a complex system is initiated by a query or goal of the form $X = \text{new } c(\bar{t})$ where \bar{t} are the arguments to the constructor and X corresponds to the attributes of class c . A solution to such a query should be an instance of class c that satisfies the above equality. In general, since the constructor arguments are fixed by \bar{t} , we define the solutions to such a query as the set of all ground instances of X such that the goal evaluates to true, i.e., X satisfies the constraints of class c . A ground instance of the attributes of a class is defined as follows.

Definition 4.9: *A valuation θ is a mapping from Cob program variables to values in the constraint domain \mathcal{D} .*

The declarative semantics of a Cob query is defined in terms of a valuation and the CLP-translation of both the Cob program and query. The notion of a correct answer to a Cob query is similar to the concept of a correct answer to a CLP query: the CLP-translation of the Cob program can be compared to a CLP program and the CLP-translation of the Cob query can be compared to a CLP query.

Definition 4.10: *Let C be a Cob program with CLP-translation P and let M_p be the least model of P . Given a goal $G \equiv X = \text{new } c(\bar{t})$, let $p_c(\bar{t}, \bar{t}')$ be its CLP-translation. A **correct answer** to G is a valuation θ for the attributes \bar{t}' of c , such that $M_p \models_{\theta} p_c(\bar{t}, \bar{t}')$, .i.e., under the valuation θ , $p_c(\bar{t}, \bar{t}')$ is in the least model M_p .*

For example, consider again the definition of class component. The correct answer to a goal `new_component(V1, I1, 10)` is a valuation θ for the variables `V1` and `I1` such that $\theta(V1, I1) = \{(X, Y) \mid X = 10 * Y\}$.

4.1.4 Cob Programs with Conditional Constraints

The translation scheme and declarative semantics described in Sections 4.1.2 and 4.1.3 do not account for Cob programs having conditional constraints. This is because constraint logic programs and their semantics cannot handle conditional constraints. To give the semantics of Cob programs in general, we need to augment CLP semantics to account for conditional constraints. Conditional constraints are of the form $A :- B$, where A is the head and B is the body of the conditional constraint. The head can be a constraint atom which could be defined in terms of a user-defined predicate if the head is a creational constraint. The body is a collection of literals (positive or negative).

To account for conditional constraints, we first extend the CLP paradigm with a special construct that represents conditional constraints. To distinguish conditional constraints from CLP program rules, we represent the conditional constraint $A :- B$ as $A \leftarrow_{cc} B$. We augment the syntax of CLP programs to include conditional constraints in the body of a rule. We refer to this extension of constraint logic programs as the **CCLP**(C) paradigm where C is a constraint domain as before. For the rest of the discussion, we will refer to **CCLP**(C) as simply **CCLP**. The logical semantics of a conditional constraint of the form

$$p(\bar{t}) \leftarrow_{cc} q_1(\bar{t}_1), \dots, q_n(\bar{t}_n)$$

is given by the logical formula

$$\tilde{\forall} p(\bar{t}) \leftarrow q_1(\bar{t}_1) \wedge \dots \wedge q_n(\bar{t}_n).$$

where the symbol $\tilde{\forall}$ represents universal closure. Thus a conditional constraint is true if the truth of its antecedent implies the truth of its consequent.

Having given a logical meaning to a conditional constraint, the declarative semantics of a **CCLP** program can be defined by incorporating this meaning into the semantics of CLP programs. Before defining the meaning of a **CCLP** program, we give a formal definition of the model of a conditional constraint.

Definition 4.11: Assuming a constraint domain C as before for a **CCLP** program, a set M_c

models a conditional constraint of the form

$$p(\bar{t}) \leftarrow_{cc} q_1(\bar{t}_1), \dots, q_n(\bar{t}_n)$$

if for every valuation ρ such that $\rho(\bar{t}) = \bar{d}$ and for $i = 1..n$ $\rho(\bar{t}_i) = \bar{d}_i$,

if $q_1(\bar{d}_1), \dots, q_n(\bar{d}_n) \in M_c$,

then $p(\bar{d}) \in M_c$ where $d_i \in \mathcal{D}$.

This notion of a model of a conditional constraint allows us to define a model of a CCLP(C) program as follows.

Definition 4.12: *Given a CCLP program P , a **model** M of P is a C -interpretation of P such that for every ground instance of a rule of P of the form*

$$p(\bar{t}) :- q_1(\bar{t}_1), \dots, q_n(\bar{t}_n), c_1(\bar{u}_1), \dots, c_m(\bar{u}_m)$$

where q_i s are user-defined predicates or constraints and c_i s are conditional constraints, if

there exists a valuation ρ such that $\rho(\bar{t}) = \bar{d}, \rho(\bar{t}_1) = \bar{d}_1, \dots, \rho(\bar{t}_n) = \bar{d}_n, \rho(\bar{u}_1) = \bar{d}'_1, \dots, \rho(\bar{u}_m) = \bar{d}'_m$ and if

$$q_1(\bar{d}_1), \dots, q_n(\bar{d}_n) \in M \text{ and } M \models c_1(\bar{u}_1), \dots, c_m(\bar{u}_m)$$

then $p(\bar{d}) \in M$.

In other words, a model of a CCLP program P is an interpretation of P such that for every rule of P , if a ground instance of the antecedant belongs to the model, then the corresponding ground instance of the consequent also belongs to the model. Clearly, a program can have more than one model and we are interested in the intended model of the program, i.e., any ground instance of a predicate that is not implied by the program rules should not be in the intended model. As before with CLP models, the intersection of two models of a CCLP program P is also a model of P . The intersection of all the models of a program is the least model of the program and this is the intended model.

Definition 4.13: *The **declarative semantics** of a CCLP program P is defined as the least model M_P of P .*

Having defined the declarative semantics of CCLP programs, we can now extend the translation scheme of Section 4.1.2 to include the following case.

Translation of Conditional Constraints. A conditional constraint is translated to a special predefined predicate cc with appropriate translation of its constituent constraints,

i.e., its antecedant and consequent.

$$A :- B \implies_T \text{cc}(A_T, B_T)$$

where A_T is obtained by translating constraint A

B_T is obtained by translating constraint B

and cc is the predefined predicate with semantics:

$$\text{cc}(X, Y) :- X. \quad (\text{i})$$

$$\text{cc}(X, Y) :- \neg Y. \quad (\text{ii})$$

The goal $\text{cc}(A, B)$ represents the conditional constraint $A \leftarrow_{\text{cc}} B$ defined previously. The above case together with the translation scheme of Section 4.1.2 can satisfactorily translate Cob programs with conditional constraints. The program resulting from such a translation is clearly a CCLP program.

Definition 4.14: *Let C be a Cob program. The CCLP program P obtained by translating each class of C using the scheme in Section 4.1.2 and every conditional constraint in C using the above rule is called the **CCLP-translation** of C . For each class c in C , the predicate p_c obtained by translating c is called the CCLP-translation of c .*

We can now define the declarative semantics of a general Cob program with conditional constraints in terms of its CCLP-translation.

Definition 4.15: *Given a Cob(C) program C defining a class c , if P is the CCLP-translation of C , p_c is the CCLP-translation of c , and M_p is the least model of P , then the **declarative semantics** of c , M_c is defined as: $M_c = \{\bar{t}_2 \mid p_c(\bar{t}_1, \bar{t}_2) \in M_p, \text{ for some } \bar{t}_1, \text{ where } \bar{t}_1, \bar{t}_2 \in \mathcal{D}\}$*

For example, the consider the Cob model of a date object given in Section 3.2. In addition to some simple constraints, this definition has the following conditional constraint.

$$\text{Day} \leq 29 :- \text{Month} = 2, \text{ leap}(\text{Year});$$

A set S of tuples $[\text{Day}, \text{Month}, \text{Year}]$ will model the above conditional constraint if for every $[X, Y, Z] \in S$, if Z is a leap year and Y is 2 then the value of X is less than 29. A model of the Cob class `date` should model all its conditional constraints as well as simple constraints. The declarative semantics of the `date` class is the least such model. Thus the intended interpretation of the `date` class is the set of $[\text{Day}, \text{Month}, \text{Year}]$ tuples that form valid dates according to the Gregorian calendar.

The declarative semantics of a query with respect to a Cob program with conditional constraints is defined in terms of a valuation and the CCLP-translation of both the Cob program and query. Since a query cannot have conditional constraints, the CCLP-translation of a query is the same as its CLP-translation.

Definition 4.16: *Let C be a Cob program with CCLP-translation P and let M_p be the least model of P . Given a goal $G \equiv X = \text{new } c(\bar{t})$, let $p_c(\bar{t}, \bar{t}')$ be its CCLP-translation. A **correct answer** to G is a valuation θ for the attributes \bar{t}' of c , such that $M_p \models_{\theta} p_c(\bar{t}, \bar{t}')$, .i.e., under the valuation θ , $p_c(\bar{t}, \bar{t}')$ is in the least model M_p .*

For example, consider again the definition of class `date`. The correct answer to a goal `Date = new date(D1, M1, 2003)` is a valuation θ for the variables `Day`, `Month` and `Year` of class `date` such that $\theta(\text{Day}, \text{Month}, \text{Year}) = \{(X, Y, 2003) \mid X \leq 31, X \leq 28 \leftarrow Y = 2, X \leq 30 \leftarrow Y \in \{4, 6, 9, 11\}\}$.

4.2 Operational Semantics

The declarative semantics of constrained objects described in Sections 4.1.2, 4.1.3 and 4.1.4 define the meaning of a Cob program and the meaning of an answer to a goal with respect to the program. Given the meaning of Cob program and a goal, we need a scheme that will compute an answer to the goal. We therefore define the operational semantics of Cob programs that provide a scheme for computing an answer (if it exists) with respect to a goal. The operational semantics of Cob programs are based upon the translation of Cob to CLP (described in Section 4.1.2) and build upon the rewrite rules of the operational semantics of CLP languages. Therefore, before defining the operational semantics of Cob programs, we give an overview of the operational semantics of CLP programs. This serves to compare the two and illustrate why CLP operational semantics alone cannot account for the CCLP programs obtained by translating Cob programs.

4.2.1 Constraint Logic Programs

In Section 2.2 we gave an informal overview of such a technique for goal evaluation in the constraint logic programming language CLP(R). We now describe the scheme for evaluat-

ing goals in any CLP(C) language as given in [60]. The concepts, definitions, and theorems described in this subsection are obtained from [60]. Let the constraint domain $C \equiv \langle \Sigma, \mathcal{D}, \mathcal{L}, \mathcal{T}, \text{solve} \rangle$, be as defined before. The operational semantics of a goal G with respect to a CLP program P are defined in terms of a *derivation* which is a finite or infinite sequence S_0, \dots, S_i, \dots of states. A *state* is a tuple $\langle G \parallel C \rangle$ where the goal G is the current set of literals and C is the current set of constraints. In every step of a derivation, a state $S_{i+1} \equiv \langle G_{i+1} \parallel C_{i+1} \rangle$ is *derived* from state $S_i \equiv \langle G_i \parallel C_i \rangle$ by selecting a literal in G_i and applying one of the following transition rules. The literal is selected based on some predefined convention (usually left-to-right).

Let a goal G_i be of the form $L_1, \dots, L_{i-1}, L_i, L_{i+1}, \dots, L_m$ and let L_i be the selected literal.

- if L_i is the user-defined constraint $p(t_1, \dots, t_n)$ and if

$$p(s_1, \dots, s_n) \text{ :- } B.$$

is a rule in the program P , then S_{i+1} is the state

$$\langle L_1, \dots, L_{i-1}, t_1 = s_1, \dots, t_n = s_n, B, L_{i+1}, \dots, L_m \parallel C \rangle$$

Note that the variables of the program rule are renamed (to new variable names) before this transition rule is applied.

- if L_i is the user-defined constraint $p(t_1, \dots, t_n)$ and if there does not exist a rule of the form

$$p(s_1, \dots, s_n) \text{ :- } B.$$

in the program P , then S_{i+1} is the state

$$\langle \phi \parallel \text{false} \rangle$$

and the derivation is said to fail.

- if L_i is a primitive constraint and $\text{solve}(L_i \wedge C) \neq \text{false}$ then S_{i+1} is the state

$$\langle L_1, \dots, L_{i-1}, L_{i+1}, \dots, L_m \parallel C \wedge L_i \rangle$$

- if L_i is a primitive constraint and $\text{solve}(L_i \wedge C) = \text{false}$ then S_{i+1} is the state

$$\langle \phi \parallel \text{false} \rangle$$

and the derivation is said to fail.

The state S_0 for a derivation of goal G is $\langle G \parallel \phi \rangle$. A derivation is said to succeed if it is finite and its last state is $\langle \phi \parallel C \rangle$ such that $\text{soln}(C)$ does not evaluate to false. In this case, the answers to the goal G with respect to a program P are the constraints $\exists_{\text{var}(G)} C$, where \exists represents existential closure. C is the computed answer and is called the **answer constraint** and any solution to C is a solution to the original goal G .

Note that in the first transition rule above, when goal reduction leads to constraints of the form $t_1 = s_1, \dots, t_n = s_n$, it is understood that the underlying constraint domain has the equality constraint which is the identity relation over D . If either t_i or s_i is a functor term, then the equality reduces to pairwise equality between their constituent arguments provided the functors are identical (otherwise the equality fails).

Having defined the declarative and operational semantics of a language, it is important to show that they agree, i.e., the answer computed by the operational semantics is correct as per the declarative semantics.

Theorem: 4.1 (Soundness) [60]: *Let P be a CLP(C) program. If goal G has answer constraint c , then $\text{lm}(P, C) \models c \rightarrow G$.*

The soundness of CLP derivations states that the answer computed by any derivation is indeed in the intended model of the program. The soundness of the derivations is proved by showing that at every step of the derivation for every type of transition rule, if $\langle G' \parallel C' \rangle$ is derived from $\langle G \parallel C \rangle$, then $P, \mathcal{T} \models (\mathcal{G}' \wedge C') \rightarrow (\mathcal{G} \wedge C)$. Thus if the start state of a derivation is $\langle G \parallel \phi \rangle$ and the final state is $\langle \phi \parallel c \rangle$, i.e., c is the computed answer, then by the transitivity of the logical connective \rightarrow , it can be seen that $P, \mathcal{T} \models (\phi \wedge c) \rightarrow (G \wedge \phi)$, which is the same as $P, \mathcal{T} \models c \rightarrow G$.

The soundness result ensures that an answer computed by goal evaluation (operational semantics) will always be correct. It is also important to see if the operational semantics can compute an answer whenever one exists.

Theorem: 4.2 (Completeness) [60]: *Let P be a CLP(C) program, G a goal and θ a valuation. If $\text{lm}(P, C) \models_{\theta} G$, then G has an answer c such that $\mathcal{D}_C \models_{\theta} c$.*

The completeness theorem states that if there is a answer to a goal in the intended model, then there is a derivation that will compute this answer. The proof of the completeness theorem uses the fixpoint semantics of constraint logic programs. The fixpoint

semantics are based on two operators. The *immediate consequence operator* T_P^C which given a CLP program P maps a set of facts F (subset of C -base of P) to the set of facts that can be deduced from F using the program rules. The power set of C -base of P , $\mathcal{P}(C\text{-base}_P)$ forms a complete lattice ordered by the subset relation. The function $T_P^C: \mathcal{P}(C\text{-base}_P) \rightarrow \mathcal{P}(C\text{-base}_P)$ is continuous and its least fixpoint is exactly the least model $lm(P, C)$ of P . A different fixpoint semantics is based on the *immediate consequence* $S_P(F)$ of a set of facts F of the form $A :- c$, using program rules P and *breadth-first* (BF) derivations. The operator $S_P(F)$ is also continuous. The least fixpoint of the two operators are directly related to each other by $[lfp(S_P)]_C = lfp(T_P^C)$, where $[F]_C = \{\sigma(A) \mid \mathcal{D} \models_\sigma c\}$. The proof of the completeness theorem uses the result: a successful BF derivation of a goal G with respect to a program P corresponds to a successful BF derivation of G with respect to the program $S_P \uparrow n$, the n^{th} ordinal power of S_P . This result is proved by induction on the length of the derivation.

4.2.2 Constrained Object Programs

We define the operational semantics of a $\text{Cob}(C)$ program in terms of its CCLP-translation. As before, C denotes the constraint domain $\langle \Sigma, \mathcal{D}, \mathcal{L}, \mathcal{T}, \text{solv} \rangle$. The solver solv is assumed to be theory complete, although in practice this may not be true. Given a Cob program C and a goal $G \equiv \text{new } c(\bar{t})$, suppose P is the CCLP-translation of C and $G_0 \equiv p_c(\bar{t})$ is the CCLP-translation of G . The operational semantics of G with respect to C are defined in terms of a *derivation* which is a finite or infinite sequence S_1, \dots, S_i, \dots of states. This is similar to the derivation sequence of CLP semantics but differs in the representation of a state and the transition rules used for deriving one state from another. A *state* is represented by the tuple $\langle A, Sc, Cc, N \rangle$, where A is a collection of atoms and constraints and Sc, Cc, N are respectively collections of simple constraints (i.e., non-conditional constraints), conditional constraints and conjunction of negated atoms. A simple constraint is any constraint other than a conditional constraint. Sc, Cc , and N can be regarded as the constraint stores. Given a goal $G \equiv \text{new } c(\bar{t})$, the *start state* is $S_0 \equiv \langle p_c(\bar{t}, \bar{t}'), \phi, \phi, \phi \rangle$. At each step in the derivation, a literal is selected from the state $S_i \equiv \langle A_i, Sc_i, Cc_i, N_i \rangle$ and one of the following transition rules is applied to obtain the next state S_{i+1} .

1. $\langle A \cup a, Sc, Cc, N \rangle \rightarrow_r \langle A \cup B, Sc \cup \{\overline{t_1} = \overline{t_2}\}, Cc, N \rangle$
 if a is the selected atom and $h \leftarrow B$ is a rule of P and a and h have the same outermost predicate and $a = p(\overline{t_1})$ and $h = p(\overline{t_2})$.
2. $\langle A \cup a, Sc, Cc, N \rangle \rightarrow_r \text{fail}$
 if a is the selected atom and there does not exist a rule $h \leftarrow B$ of program P such that a and h have the same outermost predicate, namely, $a = p(\overline{t_1})$ and $h = p(\overline{t_2})$
3. $\langle A \cup c, Sc, Cc, N \rangle \rightarrow_c$
 $\langle A, Sc \cup \{c\}, Cc, N \rangle$ if c is a simple constraint.
 $\langle A, Sc, Cc \cup \{c\}, N \rangle$ if c is a conditional constraint.
4. For conditional constraints of the type $p \leftarrow_{cc} q$, where both p and q are constraints (not literals),

$\langle A, Sc, Cc \cup \{p \leftarrow_{cc} q\}, N \rangle$	\rightarrow_{cc}
$\langle A, Sc, Cc, N \rangle$	if $P, \mathcal{T} \models \overline{\forall}(p \leftarrow (Sc \wedge N))$
$\langle A, Sc, Cc, N \cup \{\neg q\} \rangle$	if $P, \mathcal{T} \models \overline{\forall}(\neg p \leftarrow (Sc \wedge N))$
$\langle A, Sc \cup \{p\}, Cc, N \rangle$	if $P, \mathcal{T} \models \overline{\forall}(q \leftarrow (Sc \wedge N))$
5. $\langle A, Sc, Cc, N \rangle \rightarrow_f \text{fail}$ if $\neg \text{consistent}(Sc, N)$

The case-analysis in rule 4 above presents the formal operational semantics of a certain class of conditional constraints. In the first case, the condition $P, \mathcal{T} \models \overline{\forall}(p \leftarrow (Sc \wedge N))$, states that p is implied by the current set of constraints in Sc together with the negated atoms in N . If this condition is true, then the conditional constraint is satisfied. This checking is carried out by the underlying constraint solver. In the other cases, depending on the condition the appropriate constraint (or its negation) is added to the constraint store. This is an important transition rule that cannot be handled by CLP semantics alone. A derivation ends when no more transition rules can be applied on the last state. A state S_j is derived from state S_i , denoted $S_i \Rightarrow^* S_j$, if S_j is obtained from S_i by applying one or more of the above rules. If S_j is obtained from S_i by the application of exactly one of the above rules, then we say that $S_i \Rightarrow S_j$.

Definition 4.17: *If a derivation for a goal G w.r.t. a Cob program C ends in the state $\langle \phi, Sc, Cc, N \rangle$ such that $\text{consistent}(Sc, N)$, then the derivation is said to be **successful** and Sc, Cc and N are the **answer constraints**. A valuation θ is a **computed answer** if the answer constraints evaluate to true under θ , i.e., $P, T \models_{\theta} (Sc \cup Cc \cup N)$*

In terms of the engineering structure that is being modeled, a valuation means giving values for attributes of various parts of the structure (e.g. values for current, voltage and resistance of an electrical component). For example, consider the Cob classes defined in program P in Figure 4.2. A solution to the goal or Cob query `new component(10, I, 5)` with respect to P , will be obtained by a derivation of the corresponding CCLP goal $p_{\text{component}}([10, I1, 5], \bar{t})$ with respect to the CCLP program P' of Figure 4.3. A solution to this goal will be a valuation that assigns a value of $[10, 2, 5]$ to the variables $[V, I, R]$ of class `component`.

4.2.3 Soundness and Completeness

It is important to ensure that, given a goal, the answer computed by the above operational semantics is correct according to the declarative semantics proposed in Section 4.1.3. Similarly, one would like to know if the operational semantics can compute a solution whenever one exists. In this section we state and prove the soundness of Cob derivations. The completeness result is stated along with a proposed approach for its proof.

The soundness result (Lemma 4.1 and Theorem 4.1) below is proved along the lines of a similar lemma for CLP programs with the main difference being the representation of state and the case for handling conditional constraints.

Lemma 4.1: *Let C be a Cob program with CCLP-translation P . If a state $\langle G, Sc, Cc, N \rangle \Rightarrow \langle G', Sc', Cc', N' \rangle$, then $P, T \models (G' \wedge Sc' \wedge Cc' \wedge N') \rightarrow (G \wedge Sc \wedge Cc \wedge N)$.*

Proof : Let G be of the form $L_1, \dots, L_i, \dots, L_n$ and let L_i be the selected literal that is used to transform the state. The state $\langle G', Sc', Cc', N' \rangle$ is obtained by applying one of the transition rules of Section 4.2.4. We prove the lemma for every case of the transition.

1. L_i is an atom of the form $p(\bar{t}_1)$ and $p(\bar{t}_2) \leftarrow B$ is the selected rule of the program P . Then G' is $L_1, \dots, L_{i-1}, L_{i+1}, \dots, L_n, B$ and Sc' is $Sc \cup \{\bar{t}_1 = \bar{t}_2\}$ and $Cc = Cc'$ and

$$N = N'.$$

Since $p(\bar{t}_2) \leftarrow B$ is a rule of P , $P \models B \rightarrow p(\bar{t}_2)$. Also in the constraint theory \mathcal{T} , $\mathcal{T} \models \bar{t}_1 = \bar{t}_2 \rightarrow (p(\bar{t}_1) \leftrightarrow p(\bar{t}_2))$ (i)

therefore

$$P, \mathcal{T} \models ((\bar{t}_1 = \bar{t}_2 \wedge B) \rightarrow p(\bar{t}_1))$$

therefore

$$P, \mathcal{T} \models (G' \wedge Sc' \wedge Cc' \wedge N') \rightarrow (L_1, \dots, L_{i-1}, L_{i+1}, \dots, L_n, B, \wedge Sc \wedge \bar{t}_1 = \bar{t}_2 \wedge Cc \wedge N)$$

therefore

$$P, \mathcal{T} \models (G' \wedge Sc' \wedge Cc' \wedge N') \rightarrow (G \wedge Sc \wedge Cc \wedge N) \text{ using (i)}$$

2. L_i is an atom of the form $p(\bar{t}_1)$ and there does not exist a rule $h :- B$ of P such that L_i and h have the same outermost predicate of the same arity. Therefore $< G', Sc', Cc', N' >$ is actually the state *fail*. Hence,

$$P, \mathcal{T} \models (G' \wedge Sc' \wedge Cc' \wedge N') \rightarrow (G \wedge Sc \wedge Cc \wedge N)$$

3. L_i is a simple constraint. Then G' is $L_1, \dots, L_{i-1}, L_{i+1}, \dots, L_n$, and Sc' is $Sc \cup \{L_i\}$ and $Cc = Cc'$ and $N = N'$. Thus, $(G' \wedge Sc' \wedge Cc' \wedge N')$ is just a reordering of $(G \wedge Sc \wedge Cc \wedge N)$. Hence

$$P, \mathcal{T} \models (G' \wedge Sc' \wedge Cc' \wedge N') \rightarrow (G \wedge Sc \wedge Cc \wedge N)$$

4. L_i is a conditional constraint. Then G' is $L_1, \dots, L_{i-1}, L_{i+1}, \dots, L_n$, and Cc' is $Cc \cup \{L_i\}$ and $Sc = Sc'$ and $N = N'$. Thus, $(G' \wedge Sc' \wedge Cc' \wedge N')$ is just a reordering of $(G \wedge Sc \wedge Cc \wedge N)$. Hence

$$P, \mathcal{T} \models (G' \wedge Sc' \wedge Cc' \wedge N') \rightarrow (G \wedge Sc \wedge Cc \wedge N)$$

5. Let G be as before and let Cc be c_1, \dots, c_m and let c_i be the selected conditional constraint for applying transition rule 4, which is \rightarrow_{cc} . Let c_i be of the form $p \leftarrow_{cc} q_1, \dots, q_r$.

Cc' is $c_1, \dots, c_{i-1}, c_{i+1}, \dots, c_m$ and $G' = G$. (ii)

- (a) If $P, \mathcal{T} \models \bar{\nabla}(p \leftarrow (Sc \wedge N))$, then $N' = N$ and $Sc' = Sc$. (iii)

$$\text{Then } P, \mathcal{T} \models \bar{\nabla}((Sc \wedge N) \rightarrow p)$$

$$\leftrightarrow P, \mathcal{T} \models \bar{\nabla}((Sc \wedge N) \rightarrow (p \leftarrow q_1, \dots, q_r))$$

$$\begin{aligned}
& \Leftrightarrow P, \mathcal{T} \models \bar{\nabla}((Sc \wedge N) \rightarrow c_i) \\
& \Leftrightarrow P, \mathcal{T} \models \bar{\nabla}((G' \wedge Sc \wedge Cc' \wedge N) \rightarrow (G' \wedge Sc \wedge Cc' \wedge N \wedge c_i)) \\
& \Leftrightarrow P, \mathcal{T} \models \bar{\nabla}((G' \wedge Sc' \wedge Cc' \wedge N') \rightarrow (G \wedge Sc \wedge N \wedge Cc' \wedge c_i)) \text{ using (iii)} \\
& \Leftrightarrow P, \mathcal{T} \models \bar{\nabla}((G' \wedge Sc' \wedge Cc' \wedge N') \rightarrow (G \wedge Sc \wedge N \wedge Cc)) \\
\text{(b) If } P, \mathcal{T} \models \bar{\nabla}(\neg p \leftarrow (Sc \wedge N)), \text{ then } N' \text{ is } N \wedge \neg(q_1, \dots, q_r) \text{ and } Sc' = Sc. \text{ (iv)} \\
& \text{Then } P, \mathcal{T} \models \bar{\nabla}((Sc \wedge N) \rightarrow \neg p) \\
& \Leftrightarrow P, \mathcal{T} \models \bar{\nabla}((Sc \wedge N \wedge \neg(q_1, \dots, q_r)) \rightarrow (\neg p \wedge \neg(q_1, \dots, q_r))) \\
& \Leftrightarrow P, \mathcal{T} \models \bar{\nabla}((Sc \wedge N \wedge \neg(q_1, \dots, q_r)) \rightarrow (p \leftarrow (q_1, \dots, q_r))) \\
& \Leftrightarrow P, \mathcal{T} \models \bar{\nabla}((Sc \wedge N \wedge \neg(q_1, \dots, q_r)) \rightarrow c_i) \\
& \Leftrightarrow P, \mathcal{T} \models \bar{\nabla}((Sc \wedge N \wedge \neg(q_1, \dots, q_r)) \rightarrow (Sc \wedge N \wedge c_i)) \\
& \Leftrightarrow P, \mathcal{T} \models \bar{\nabla}((Sc' \wedge N') \rightarrow (Sc \wedge N \wedge c_i)) \text{ using (iv)} \\
& \Leftrightarrow P, \mathcal{T} \models \bar{\nabla}((G' \wedge Sc' \wedge Cc' \wedge N') \rightarrow (G' \wedge Sc \wedge N \wedge Cc' \wedge c_i)) \\
& \Leftrightarrow P, \mathcal{T} \models \bar{\nabla}((G' \wedge Sc' \wedge Cc' \wedge N') \rightarrow (G \wedge Sc \wedge N \wedge Cc)) \text{ using (ii)} \\
\text{(c) If } P, \mathcal{T} \models \bar{\nabla}((q_1, \dots, q_r) \leftarrow (Sc \wedge N)), \text{ then } N = N' \text{ and } Sc' = Sc \wedge p. \text{ (iv)} \\
& \text{Then } P, \mathcal{T} \models \bar{\nabla}((Sc \wedge N) \rightarrow (q_1, \dots, q_r)) \\
& \Leftrightarrow P, \mathcal{T} \models \bar{\nabla}((Sc \wedge p \wedge N) \rightarrow (p \wedge (q_1, \dots, q_r))) \\
& \Leftrightarrow P, \mathcal{T} \models \bar{\nabla}((Sc \wedge p \wedge N) \rightarrow (p \leftarrow (q_1, \dots, q_r))) \\
& \Leftrightarrow P, \mathcal{T} \models \bar{\nabla}((Sc \wedge p \wedge N) \rightarrow c_i) \\
& \Leftrightarrow P, \mathcal{T} \models \bar{\nabla}((Sc \wedge p \wedge N) \rightarrow (Sc \wedge N \wedge c_i)) \\
& \Leftrightarrow P, \mathcal{T} \models \bar{\nabla}((Sc' \wedge N') \rightarrow (Sc \wedge N \wedge c_i)) \text{ using (iv)} \\
& \Leftrightarrow P, \mathcal{T} \models \bar{\nabla}((G' \wedge Sc' \wedge Cc' \wedge N') \rightarrow (G' \wedge Sc \wedge N \wedge Cc' \wedge c_i)) \\
& \Leftrightarrow P, \mathcal{T} \models \bar{\nabla}((G' \wedge Sc' \wedge Cc' \wedge N') \rightarrow (G \wedge Sc \wedge Cc \wedge N)) \text{ using (ii)}
\end{aligned}$$

Theorem 4.3 (Soundness): *Given a Cob(C) program C , its CCLP-translation P with least model M_p , and a goal G , if a derivation for G w.r.t. P ends in $\langle \phi, Sc, Cc, N \rangle$ such that $\text{consistent}(Sc, Cc, N)$, then $M_p, \mathcal{T} \models (Sc \cup Cc \cup N) \rightarrow G$.*

Proof : A derivation for G with respect to P has the start state $S_0 \equiv \langle G, \phi, \phi, \phi \rangle$. Suppose that in n steps, the derivation reaches the final state $S_n \equiv \langle \phi, Sc, Cc, N \rangle$ such that $\text{consistent}(Sc, Cc, N)$. Suppose the intermediate states of the derivation are $S_i \equiv \langle G_i, Sc_i, Cc_i, N_i \rangle$ such that state $S_{i+1} \equiv \langle G_{i+1}, Sc_{i+1}, Cc_{i+1}, N_{i+1} \rangle$ is derived from S_i by one derivation step.

By Lemma 4.1,

$$M_p, \mathcal{T} \models (G_{i+1} \wedge Sc_{i+1} \wedge Cc_{i+1} \wedge N_{i+1}) \rightarrow (G_i \wedge Sc_i \wedge Cc_i \wedge N_i) \forall i \in 0..n-1$$

By repeated use of Lemma 4.1 and the transitivity of \rightarrow ,

$$M_p, \mathcal{T} \models (G_0 \wedge Sc_0 \wedge Cc_0 \wedge N_0) \rightarrow (G_n \wedge Sc_n \wedge Cc_n \wedge N_n)$$

Since Sc_0, Cc_0, N_0 and G_n are empty and $G_n \equiv G, Sc_n \equiv Sc, Cc_n \equiv Cc$ and $N_n \equiv N$, the following holds.

$$M_p, \mathcal{T} \models (Sc \cup Cc \cup N) \rightarrow G.$$

Thus the above lemma and theorem prove that a computed answer (as per the operational semantics) to a Cob program with respect to a query or goal is indeed a correct answer (as defined by the declarative semantics of Cob).

Theorem 4.4 (Completeness): *Given a Cob(C) program C , its CCLP-translation P and a goal G , if a valuation θ is such that $M_p \models_{\theta} G$, then there exists a finite derivation with start state $\langle G, \phi, \phi, \phi \rangle$ and ending in $\langle \phi, Sc, Cc, N \rangle$ such that $\text{consistent}(Sc, Cc, N)$ and $P, \mathcal{T} \models_{\theta} (Sc \cup Cc \cup N)$.*

We do not provide a full proof of completeness in this dissertation, but provide an outline of the main steps needed for this proof. Basically, the proof strategy follows that of the CLP language, i.e., it requires an intermediate step involving fixpoint semantics. We may define an immediate consequence operator for a CCLP program whose least fixpoint coincides with the least model of the program. By defining another operator that computes immediate consequences of certain rules based on breadth-first derivations, and relating it to the immediate consequence operator, we may be able to show that an element in the least model of the program can be reached within a finite number of steps of breadth-first derivations. The presence of user-defined predicates in the antecedant of a conditional constraint may give rise to nontermination. Hence in the proof of completeness bounded-depth recursion may have to be assumed. The operational semantics given above can be compared with the semantics of CLP languages given in [61, 60].

4.2.4 Operational Semantics of General Conditional Constraints

Thus far, we have considered conditional constraints in which the head and body consist of constraints. In general, the head of a conditional constraint consists of a constraint or a positive atom and the body can have constraints or positive or negative atoms. To determine

the consistency of such conditional constraints, the operational model must determine if the validity (truth) of a constraint or atom is implied by the state of a computation. This cannot be determined by the CLP semantics (transition rules) described in Section 4.2.1. In addition to the transition rules given for Cob programs in Section 4.2.2, we define below the transition rules required for handling a state consisting of general conditional constraints. The selected conditional constraint is shown enclosed in parenthesis ($\{\}$) and the following notation is used.

A : collection of atoms and constraints.

Sc : set of simple constraints.

Cc : set of conditional constraints.

N : set of negated literals.

c, c_i : a constraint atom.

a, a_i : an atom.

l_i : literal (positive or negative atom).

b, b_i : a constraint or an atom.

$$1. \langle A, Sc, Cc \cup \{c : - c_1, \dots, c_m\}, N \rangle$$

This case was discussed in Section 4.2.4.

$$2. \langle A, Sc, Cc \cup \{c : - b_1, \dots, b_m\}, N \rangle \rightarrow_{cc} \langle A, Sc, Cc, N \rangle \text{ if } P, \mathcal{T} \models \bar{\forall}(c \leftarrow (Sc \cup N))$$

$$3. \langle A, Sc, Cc \cup \{c : - b_1, \dots, b_m\}, N \rangle \rightarrow_{cc} \langle A, Sc, Cc, N \cup \{\neg(b_1, \dots, b_m)\} \rangle \text{ if } P, \mathcal{T} \models \bar{\forall}(\neg c \leftarrow Sc \cup N)$$

$$4. \langle A, Sc, Cc \cup \{b : - c_1, \dots, c_m\}, N \rangle \rightarrow_{cc} \langle A \cup \{b\}, Sc, Cc, N \rangle \text{ if foreach } i \in \{1..m\}, P, \mathcal{T} \models \bar{\forall}(c_i \leftarrow (Sc \cup N))$$

$$5. \langle A, Sc, Cc \cup \{b : - c_1, \dots, c_m\}, N \rangle \rightarrow_{cc} \langle A, Sc, Cc, N \rangle \text{ if for some } i \in \{1..m\}, P, \mathcal{T} \models \bar{\forall}(\neg c_i \leftarrow (Sc \cup N))$$

$$6. \langle A, Sc, Cc \cup \{a : - b_1, \dots, b_m\}, N \rangle \rightarrow_{cc} \langle A, Sc, Cc, N \rangle \text{ if } ground(a) \text{ and } P, \mathcal{T} \models \bar{\forall}(a \leftarrow (Sc \cup N))$$

$$7. \langle A, Sc, Cc \cup \{a : - b_1, \dots, b_m\}, N \rangle \rightarrow_{cc} \langle A, Sc, Cc, N \cup \{\neg(b_1, \dots, b_m)\} \rangle \text{ if } ground(a) \text{ and } P, \mathcal{T} \models \bar{\forall}(\neg a \leftarrow (Sc \cup N))$$

8. $\langle A, Sc, Cc \cup \{b : -l_1, \dots, l_m\}, N \rangle \rightarrow_{cc} \langle A \cup \{b\}, Sc, Cc, N \rangle$ if for each $i \in \{1..m\}$, $ground(l_i)$ and $P, T \models \forall(l_i \leftarrow (Sc \cup N))$
9. $\langle A, Sc, Cc \cup \{b : -l_1, \dots, l_m\}, N \rangle \rightarrow_{cc} \langle A, Sc, Cc, N \rangle$ if for some $i \in \{1..m\}$, $ground(l_i)$ and $P, T \models \forall(\neg l_i \leftarrow (Sc \cup N))$

Clearly, the above transition rules are not subsumed by the CLP semantics given in Section 4.2.1 since entailment of constraints or atoms is not handled by CLP. The operational semantics of Cob, on the other hand, subsume and extend CLP semantics to handle general conditional constraints. The above transition rules together with the operational semantics for Cob programs given in Section 4.2.2 serve as a basis for an implementation of a compiler for Cob programs. Currently, our computational model (compiler) implements rules 1,2,4 and 7-9. Essentially, evaluation of conditional constraints can be done if they are simple conditional constraints, or after sufficient information about them is obtained, i.e., they become sufficiently ground thus making it possible to check for the validity of a constraint and ground literals. A more detailed description of how the compiler handles conditional constraints is given in Chapter 5.

4.3 Summary

In this chapter we defined the formal declarative and operational semantics of constrained objects. First an overview of the declarative semantics of CLP programs [60] was given. The logical meaning of a CLP rule and a program were described. The set-theoretic semantics of a $CLP(C)$ program is the smallest interpretation of the program symbols that models the rules of the program, i.e., its least model. We then defined the declarative semantics of Cob programs.

A Cob program can be translated to a CLP predicate in a natural way: the arguments of the predicate represent aggregation and the constraints in the body of the predicate correspond to the constraints of the Cob class. We defined a novel scheme for systematic translation of Cob programs to CLP programs. Using this translation, we defined the declarative semantics of a Cob program (without conditional constraints) as the least model of its corresponding CLP-translation.

Cob programs with conditional constraints cannot be translated directly to CLP programs. Hence we defined a CCLP program as an augmentation of a CLP program with conditional constraints. We gave a logical semantics for conditional constraints and defined the notion of a model of a conditional constraint as well as a CCLP program. Cob programs with conditional constraints are translated to CCLP programs and their declarative semantics is defined as the least model of their CCLP-translation. Informally, the meaning of a Cob class is given by the set of values for its attributes such that the constraints of the class evaluate to true.

We presented an overview of the operational semantics of CLP programs [60] and defined the operational semantics of Cob programs in terms of derivations of their CCLP-translations. These derivations are similar to CLP derivations except for the case of conditional constraints. We gave the complete operational semantics of Cob programs with restricted conditional constraints. The soundness result for such programs was stated with proof and the completeness result was stated with a proposed approach for a proof along the lines of the completeness result for CLP derivations. We then briefly presented an overview of the possible cases for goal evaluation with respect to Cob programs with general conditional constraints. The declarative semantics of constrained objects, the translation scheme for Cob to CLP and the operational semantics of Cob programs presented in this chapter define the meaning of constrained objects and a scheme for computing this meaning. We have successfully used these schemes for developing a compiler for Cob programs. The implementation of this compiler is discussed in detail in Chapter 5.

Chapter 5

Implementation of Constrained Objects

The overall modeling scenario for constrained objects can be summarized as follows:

$$modeling ::= build; solve; [[modify; re_solve]^+; query^*]^+$$

A modeler can *build* a complex object either by defining classes of constrained objects and composing together instances of these classes or by drawing the complex object through a domain specific visual interface. Given a complex object, i.e., a composition of constrained objects, the Cob computational engine tries to *solve* their constraints and arrive at values for the (uninitialized) attributes of these constrained objects. Operations can then be performed through the visual interfaces to *modify* these values, and the Cob engine will attempt to re-solve the constraints and arrive at new values for attributes that will satisfy the constraints. The complex object can then be *queried* to find possible assignments of values to attributes that will satisfy some given constraints in addition to the ones already present in the constrained objects. The process of modifying and querying can be repeated as many times as the modeler wishes.

As described in Chapter 3, Cob models can be developed (*build* phase) either by: (i) a domain-independent interface for building class diagrams; (ii) domain-dependent interfaces for drawing diagrams of engineering structures; or (iii) by writing textual Cob code. For the first two cases, the interfaces are furnished with domain-independent (CUMML) and domain-specific compilers that translate class diagrams and drawings respectively to textual Cob code. The Cob code generated by any of the above three methods is then translated to CLP(R) code by the prototype Cob compiler. This development and compilation of Cob

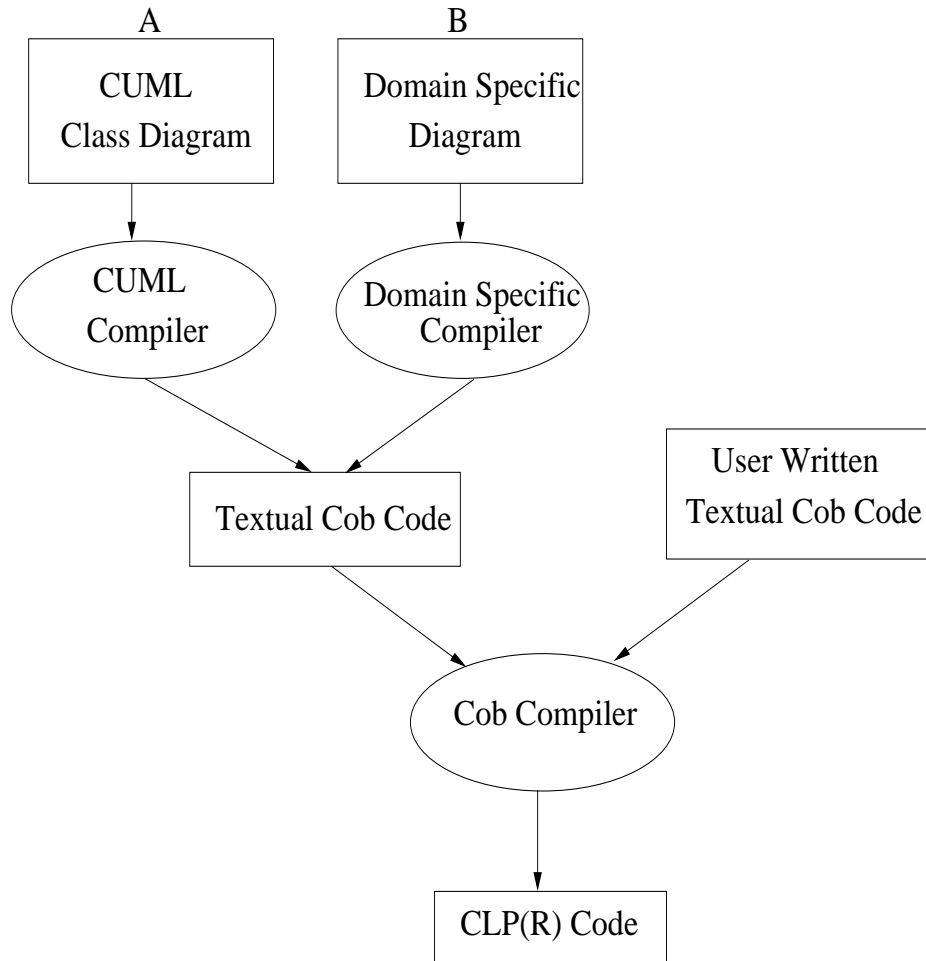


Figure 5.1: Cob Computational Environment: *build* phase

programs is illustrated in Figure 5.1.

The implementation of the Cob compiler is based upon the translation of Cob to CLP discussed in Chapter 4. Since a Cob class defines a datatype satisfying certain constraints or relations, the mapping of a class to a predicate is natural, as the predicate can be thought of as a procedure for enforcing the constraints of the class. This translation also facilitates a straightforward translation of the CLP predicates that may be defined within a Cob class and used in the constraints of the class.

The next phase (*solve*) involves the evaluation of this CLP(R) code with respect to the query obtained by translating the top-level call to the constructor of the complex object. The default query corresponds to a call to main (when such a class exists); otherwise it is specified by the user (see Figure 5.2). The translated CLP(R) code can be run directly on the

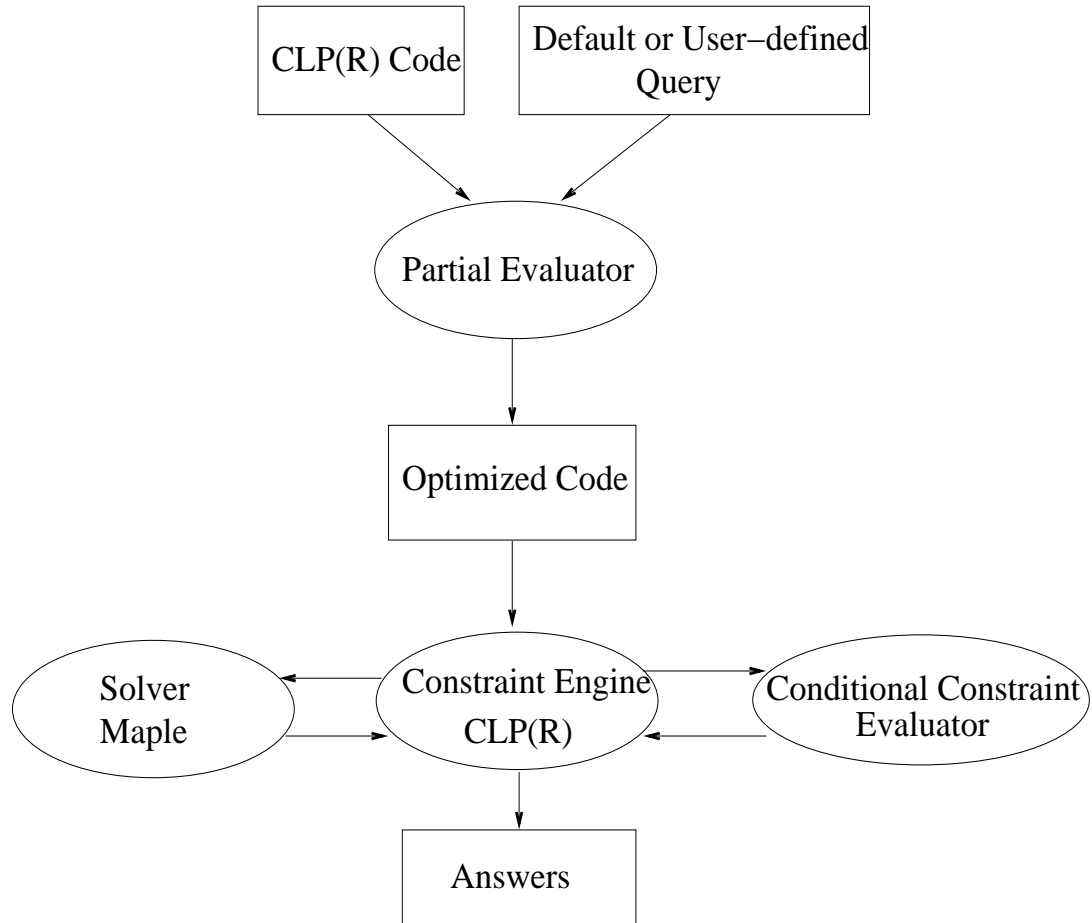


Figure 5.2: Cob Execution Environment: *solve* phase

CLP(R) interpreter. The first version of our computational model [68] did precisely this. Such an evaluation however, suffers from certain limitations: constraints can be solved only if they are sufficiently linear; there is no mechanism in CLP to handle conditional constraints; and, for large programs the evaluation can be slow due to repeated attempts to solve a large constraint store while generating more constraints and variables.

We can obtain a more efficient and general implementation by employing a “pre-execution” step wherein we partially execute the translated CLP(R) program in order to unwrap constraints from their object containers and also unravel loops arising from quantified constraints. The optimized code resulting from this partial evaluation is essentially a collection of constraints including linear, non-linear and conditional constraints. Partial evaluation enables us to collect non-linear constraints and use a more powerful constraint solver, such

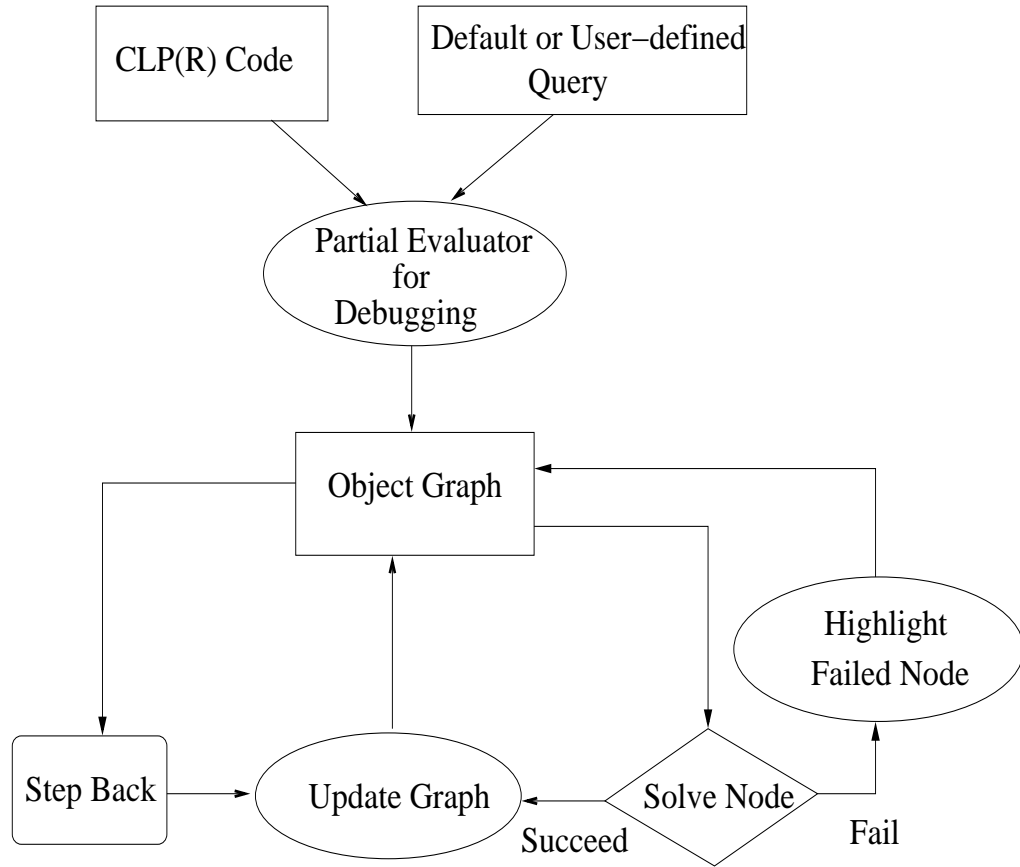


Figure 5.3: Cob Interactive Execution Environment

as Maple [112] to solve them. The partial evaluator also allows us to handle conditional constraints separately and satisfactorily (see Figure 5.2).

Our technique for partial evaluation for optimized code generation also enables novel techniques for interactive execution including visual debugging of constrained object programs. A variant of the partial evaluator is used to generate the constrained object graph (defined in Section 5.3) for a given Cob program and query. This graph contains information about Cob object instances present during the execution of a Cob program and their relation (aggregation) with each other. In the interactive execution mode, the solving can be performed step wise and viewed via the graph, on a per node (object instance) basis. It is possible to step forward and backward through the *solve* phase and in case there is an error, the object in which it occurs is highlighted. This process is shown in Figure 5.3.

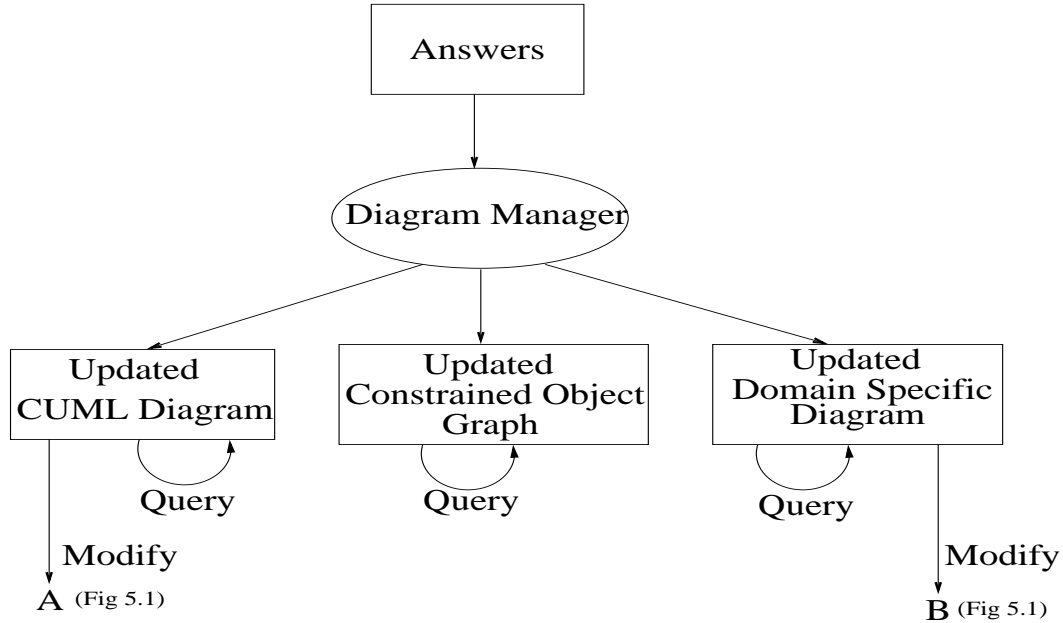


Figure 5.4: Cob Execution Environment: *query phase*

The final answers thus obtained are displayed either at the command line, on the class diagram, on the drawing, or on the object graph as the case may be. The user can now query the diagram or graph for detailed information about a component or node by clicking on the relevant part of the diagram/graph. The user can now modify the model either through the textual code (not shown in above diagram), the domain-specific drawing, or the CUMML diagram, but not through the object graph. Subsequently the modeler can go through one or more iterations of the above (*solve* and *query* computation). This process is show in Figure 5.4.

The material presented in this chapter is organized as follows. We first describe the computational model underlying the Cob programming language and modeling environment. Section 5.1 gives details of the translation of Cob programs to CLP programs and describes some properties of the translated programs. In Section 5.2 we describe a scheme for partial evaluation of the CLP-translations of Cob programs and illustrate it with some examples. We also describe how this scheme enables us to improve the efficiency of large-scale models and handle conditional as well as non-linear constraints. Section 5.3 describes a visual environment for interactive execution of Cob programs. Section 5.4 compares our partial evaluation technique with related work and Section 5.5 summarizes the material

presented in this chapter.

5.1 Cob Compiler

We have developed a prototype Cob compiler based on the translation of Cob to CLP shown in Section 4.2. The Cob compiler takes as input a Cob program C and generates a CLP program P . The execution of a Cob program is initiated by a call to the constructor of a Cob class, e.g., $X = \text{new } c(\text{Args})$. This call is translated by the compiler to the CLP query $q \equiv p_c(\text{Args}, X)$. The result of evaluating the call on the constructor of c is given in terms of the result of evaluating the query q with respect to the CLP program P .

5.1.1 Translation of Cob to CLP

In Section 4.2 we described the translation of a Cob class, a constraint atom and a constructor. In this section, we give more details of this translation pertaining to quantified constraints and terms, arrays, indexing, lists, etc.

Quantified Constraint. The i^{th} occurrence of a `forall` quantified constraint is translated to a predicate `foralli` with suitable parameters that capture the variables in the constraint in the body of the `forall`.

$$\begin{aligned} \text{forall } X \text{ in } E : C &\implies_T \text{forall}_i(E, \mathcal{V}) \\ \text{where } &\text{forall}_i([], \mathcal{V}). \\ &\text{forall}_i([X | \text{Tail}], \mathcal{V}) :- C_T, \text{forall}_i(\text{Tail}, \mathcal{V}). \\ &\mathcal{V} \text{ is a list of the variables in } C \\ \text{and } &C_T \text{ is obtained by translating constraint } C \end{aligned}$$

The predicate `foralli` is defined recursively and iterates, via variable X , through its first argument. In the translation of the quantified constraint above, the enumeration E is passed as the first argument to this predicate. This ensures that the variable X iterates through every element of the enumeration. Note that since the original constraint C contains X , so does its translation C_T . Hence, every time the second clause of `foralli` is evaluated, the X in the head of the clause matches the X present in C_T and the constraint is iteratively evaluated for every element of the enumeration E .

Passing \mathcal{V} as an argument to `foralli` ensures that the constraint C_T is evaluated in the correct context, i.e., the variables present in C_T unify or bind with the same value as they did in the context in which the quantified constraint appeared in the original program. This is important since otherwise the variables of C_T , other than X , will become free variables in the second clause.

In general, a programmer may use same variable names for the quantified variables in two separate quantified constraints. Semantically, this does not lead to any unintentional sharing of variables between the two constraints since the scope of the quantified variables is well defined and separate. To ensure such semantics (i.e., no unintentional sharing of variables) in the translation of the quantified constraints, quantified variables like X above are renamed (to unique internal variable names) before carrying out the above translation. Similarly, if the enumeration is an interval of integers, it is translated as shown below. Note that the predicate `makelist` is a predefined predicate whereas the predicate `foralli` is defined during the translation.

$$\text{forall } X \text{ in } I..J: C \quad \Longrightarrow_T \quad \text{makelist}(I, J, ItoJ),$$

$$\text{forall}_i(ItoJ, \mathcal{V})$$

where $\text{makelist}(N, M, NtoM)$ is defined such that
 $NtoM$ is the list of integers from N to M .

Existentially quantified constraints are translated along the same lines, with an appropriate definition for the `existsi` predicate. As explained in the case of quantified constraints above, the list of variables \mathcal{V} is passed in order to ensure that the translated constraint C is evaluated in the correct context in the body of the `existsi` predicate.

$$\text{exists } X \text{ in } E: C \quad \Longrightarrow_T \quad \text{exists}_i(E, \mathcal{V})$$

where $\text{exists}_i([X | \text{Tail}], \mathcal{V}) :- C_T.$
 $\text{exists}_i([X | \text{Tail}], \mathcal{V}) :- \text{exists}_i(\text{Tail}, \mathcal{V}).$
 \mathcal{V} is a list of the variables in C
and C_T is obtained by translating constraint C

Conditional Constraints. As shown in Section 4.2.2, conditional constraints are translated as follows.

$$A : - B \implies_T cc(A_T, B_T)$$

where A_T is obtained by translating constraint A

B_T is obtained by translating constraint B

and cc is the predefined predicate with semantics:

$$cc(X, Y) : - X. \quad (i)$$

$$cc(X, Y) : - \text{not } Y. \quad (ii)$$

The implementation of cc is:

$cc(A, _) :- \text{ground}(A), \text{call}(A), !.$

$cc(A, B) :- \text{ground}(A), !, (B).$

$cc(A, B) :- \text{ground}(B), \text{call}(B), !, \text{call}(A).$

$cc(_, B) :- \text{ground}(B), !.$

It is clear that this simplistic implementation of the conditional constraints will result in the correct answer only for sufficiently ground instances of the constraint. This was the implementation of conditional constraints in the first version of the Cob compiler [67, 68] whose output ran on CLP(R). The current version of the compiler translates Cob code to Sicstus Prolog [104]. This reduces the groundness requirement to some extent because Sicstus provides a builtin predicate to check entailment of a constraint. This scheme is described in the section on partial evaluation (Section 5.3).

Term. Terms appear in constraints and as arguments to functions (sin, cos, etc.) and constructors. Translation of constraints involves the translation of terms. We show below how the different kinds of terms are translated.

The i^{th} occurrence of a summation term is replaced by a new variable Sum_i whose value is computed by an invocation of the predicate sum_i with suitable parameters that capture the variables in the term in the body of the sum term.

$$\text{sum } X \text{ in } E : T \implies_T Sum_i$$

where $sum_i(E, \mathcal{V}, Sum_i)$

where $sum_i([], \mathcal{V}, 0).$

$$\begin{aligned} \text{sum}_i([X \mid \text{Tail}], \mathcal{V}, \mathcal{T}_T + \text{SumRest}) : - \\ \text{sum}_i(\text{Tail}, \mathcal{V}, \text{SumRest}). \end{aligned}$$

\mathcal{V} is a list of the variables in term \mathcal{T}

and \mathcal{T}_T is obtained by translating term \mathcal{T}

Product terms are translated similarly with a $*$ instead of the $+$. Terms computing the minimum of a collection of terms (\min) are translated similarly by iteratively computing the minimum of two terms instead of the sum as shown above.

$$\begin{aligned} \min X \text{ in } E : \mathcal{T} &\Longrightarrow_T \text{Cobvar} \\ \text{where } \min(E, \mathcal{V}, \text{Cobvar}) & \\ \text{where } \min([X], \mathcal{V}, \mathcal{T}_T). & \\ \min([X \mid \text{Tail}], \mathcal{V}, \min(\mathcal{T}_T, \text{MinRest})) : - & \\ \min(\text{Tail}, \mathcal{V}, \text{MinRest}). & \\ \mathcal{V} \text{ are the variables in } \mathcal{C} & \\ \text{and } \mathcal{T}_T \text{ is obtained by translating } \mathcal{T} & \end{aligned}$$

Terms computing the maximum are translated similarly with $\max/2$ replacing $\min/2$.

Selection. Selection terms of the type $A.B$ are translated by retrieving the list of attributes of the class t_A (type of A) and extracting attribute B from it as shown below.

$$\begin{aligned} A.B &\Longrightarrow_T A.B \\ \text{where } A &= A.V_{t_A} \\ \text{and } t_A &\text{ is the type of } A \\ V_{t_A} &\text{ is the list of attributes of the class } t_A. \\ A.V_{t_A} &\equiv \text{prefix every element of } V_{t_A} \text{ with } A_-. \end{aligned}$$

Suppose an attribute A is of type t_A , and $A.B_i$ is obtained from $A.B_i$ using the above translation. Suppose class t_A is defined as:

`class t_A { attributes $X_1 B_1; \dots; X_n B_n$; constraints \dots }`

To illustrate that $A.B_i$ indeed refers to the attribute B_i of A , we give the following justification.

From the translation of constructor calls, we know that an instance of a class is represented as a list of variables corresponding to the attributes of the class. It is given that A is of type t_A , i.e., A is an instance of class t_A . Hence we know that

$$A = [X_{B_1}, \dots, X_{B_n}]$$

where *forall* $i \in 1..n$, X_{B_i} refers to the attribute B_i of A

From the translation, it is also known that V_{t_A} is the list of attribute names of class t_A . Thus,

$$V_{t_A} = [B_1, \dots, B_n]$$

By prefixing every attribute name with $A_$ we get

$$A_V_{t_A} = [A_B_1, \dots, A_B_n]$$

From the translation, it is known that

$$A = A_V_{t_A}$$

Therefore,

$$[X_{B_1}, \dots, X_{B_n}] = [A_B_1, \dots, A_B_n]$$

This means that corresponding elements of the list are equal. Hence, for $i = 1..n$,

$$A_B_i = X_{B_i}$$

Thus A_B_i refers to the attribute B_i of A .

Arithmetic expressions are translated by translating their component terms. For example $A + B$ is translated to $T_A + T_B$ where T_A and T_B are the translations of the terms A and B respectively. Similarly, an enclosed term (A) is translated to (T_A) . A list of terms is translated by translating each element of the list. A function call is translated by translating each of its arguments, i.e., $func_{id}(terms)$ is translated to $func_{id}(T_{terms})$, where T_{terms} is the translation of the list of terms $terms$. A variable V is translated as is, i.e., it does not change under translation unless the context in which it appears requires that it be renamed to another variable name (for example, the quantified variable in a quantified constraint is renamed in order to avoid unintentional variable sharing).

Arrays and Indexing. Arrays are represented as lists and multidimensional arrays are represented as list of lists of lists and so on depending on the dimension ¹. Array declarations are translated to calls on the predefined predicate `makearray`. An array of undefined size is represented as the list `[_]`.

$$typeName [I][J] X \implies_T \text{makearray}([I, J], X)$$

$$\text{where } \text{makearray}(ListDim, Y): Y \text{ a list of size } N$$

¹The inefficiency arising from list traversal during array creation and indexing is overcome by partial evaluation (see Section 5.2).

where N is the length of the list *ListDim*.

If the n^{th} element of *ListofDim* is M , then

the n^{th} element of Y is a list of size M .

As array declarations of unspecified size are bound to $[_]$, and elements can be inserted into such an array one at a time, the tail of the array may be a variable ($[_]$). Hence the actual definitions of recursive predicates like `foralli`, `sumi`, etc. contain a conditional statement that checks if the tail is a variable.

We provide a predefined predicate `index` for accessing elements of a list. This predicate takes as input a list L and a positive integer I (the index) and returns the I^{th} element of L . For large values (up to n) of the index, this predicate is defined by pattern matching against the input list (array). For all values of the index less than n , the goal `index(L, n)` matches exactly one non-recursive clause in the definition of `index`. In other words, by using pattern matching we are able to avoid a recursive definition as well as iterative list traversal. Evaluation of the `index` predicate thus takes the same amount of time for all indices less than n . This ensures that for all practical purposes, list or array accessing is done in constant time. Translation of array indexing in Cob makes use of the `index` predicate as shown below.

Indexed terms are translated to a new variable whose value is obtained by a call to the predefined predicate `index`. Translation of multilevel indexing involves multiple calls to `index`, the result of one call being indexed by the next.

$$\begin{aligned}
 X [I] &\implies_T X_I \\
 \text{where } &\text{index}(X, I_T, X_I) \\
 \text{and } &I_T \text{ is obtained by translating } I \\
 X [I] [J] &\implies_T X_I_J \\
 \text{where } &\text{index}(X, I_T, X_I) \\
 \text{and } &\text{index}(X_I, J_T, X_I_J) \\
 \text{and } &J_T \text{ is obtained by translating } J
 \end{aligned}$$

In our scheme of partial evaluation, calls to the `index` predicate are evaluated at the “pre-execution” stage. Although evaluation of indexing goals takes constant time as explained

```

class component {
    attributes    real V, I, R;
    constraints V = I * R;
    constructors component(V1,I1,R1){
        V = V1; I = I1; R = R1;
    }
}

class parallel extends component {
    attributes component [] PC;
    constraints
    forall X in PC : X.V = V;
    sum Y in PC : Y.I = I;
    sum Z in PC : (1/Z.R) = 1/R;
    constructors parallel(B) {
        PC = B;
    }
}

```

Figure 5.5: Cob class definitions

before, their processing away during partial evaluation leads to a more efficient execution of the model at runtime.

To illustrate the above translations with a concrete example, consider the Cob classes shown in Figure 5.5. Class `component` models an electrical component with attributes voltage(V), current(I) and resistance (R) that obey Ohm's law ($V=I \cdot R$). Class `parallel` represents a collection of components(PC) connected in parallel. Figure 5.6 shows the translation of these Cob classes to CLP predicates.

```

Pcomponent([V1, I1, R1], [V, I, R]) :-
    V = I * R,
    V = V1,
    I = I1,
    R = R1.

Pparallel([B], [V, I, R, PC]) :-
    PC = [],
    PC = B,
    Pcomponent(_, [V, I, R]),
    forall1(PC, X, V),
    N1 = I,
    sum1(PC, N1, Y, I),
    N2 = 1/R,
    sum2(PC, N2, Z, R).
forall1([], X, V).
forall1([X|Tail], X, V) :-
    X_V=V,
    X=[X_V, X_I, X_R],
    forall1(Tail, X, V).
sum2([], 0, Z, R).
sum2([Z|Tail], (1/Z_R)+ Sumrest, Z, R) :-
    sum2(Tail, Sumrest, Z, R),
    Z = [Z_V, Z_I, Z_R].
sum1([], 0, Y, I).
sum1([Y|Tail], (Y_I)+Sumrest, Y, I) :-
    sum1(Tail, Sumrest, Y, I),
    Y=[Y_V, Y_I, Y_R].

```

Figure 5.6: Translated CLP(R) code

5.1.2 Classification of Predicates

The CLP program obtained by translating a Cob program is a sequence of predicate definitions. Since some of these definitions make use of certain builtin or predefined predicates, the translated program is appended with the definition of these predicates before evaluation. The predicates in the resulting composite program can be classified into the following categories.

1. CLP builtins The predicates in this category are either CLP builtin predicates or defined directly in terms of CLP builtin predicates.

- **I/O Predicates:** Input-output can be performed in a Cob program through the use of the `dump/1` or the CLP built-in predicates `print/1`, `tell/1`, `told/1`, `read/1`, `write/1`. These are generated by the non-terminal *constraint_predicate_id* in the grammar in section 3.1. The predicate `dump/1` is predefined in Cob using the CLP built-in predicate `print/1`. The Cob programmer can directly use these predicates in a Cob program.
- **Predicate for constraint handling (`expand/0`):** In the first version of the compiler mentioned in [68], the translated code was run on CLP(R). The output of the current version of the Cob compiler is run on the `clpr` module provided by SICStus Prolog [104]. In the `clpr` module of Sicstus Prolog, constraints must be explicitly enclosed in `{}` to indicate their interpreted meaning. The translation described in the previous subsection generates programs where symbols and constraints are interpreted by default without the need for a surrounding `{}`. To transform such translated programs into programs where constraints are enclosed in `{}` for input to Sicstus `clpr` module, we use the `expand/0` predicate provided for this purpose by Sicstus Prolog. As a result of using `expand/0`, all constraints get explicitly enclosed in `{}`. A call to the `expand/0` predicate is inserted at the beginning of the translated CLP program. A slightly modified translation can achieve a similar output by explicitly enclosing constraints within `{}` during the translation itself.

2. Cob compiler builtins The predicates in this category are either predefined or generated automatically by the Cob compiler.

- **Predicate for conditional constraints (cc/2):** The conditional constraints appearing in a Cob program are translated into a call to the predefined predicate `cc/2`. A simple implementation of this predicate was given in section 5.2.1 which gave correct answers only for sufficiently ground instances of conditional constraints. The current version of the Cob compiler uses a modified definition which reduces the groundness requirement to some extent and the Cob computational model delays the evaluation of conditional constraints until sufficient information is known about them. This strategy is discussed in Section 5.3.2.
- **Predicates for quantification and aggregation:** These predicates are generated as a result of translating the quantified constraints, existential constraints and quantified terms present in a Cob program (as shown in Section 5.2.1). These predicates are named `foralli/2`, `existsi/2`, `mini/3`, `maxi/3` or `sumi/3` depending on the occurrence of the constraint or term they represent. The body of these predicate definitions captures the corresponding constraint (or term) of the Cob program and hence evaluation of these predicates gives rise to constraints. Note that these predicates are not predefined, as their definition depends on the Cob program and hence is given by the compiler.
- **Predicates for array operations:** Array declarations, array accessing and certain enumerations in a Cob program are translated to calls on the predicates `makelist/3`, `makearray/2` and `index/3` (as shown in Section 5.2.1). These are predefined predicates whose definitions are appended to the translated program before its evaluation. Note that the Cob programmer does not invoke these predicates, they appear only in the translated CLP code. They are defined recursively and involve list processing and their evaluation will not generate any constraints. The scheme for partial evaluation described in Section 5.3 will process away these predicates.

3. User-defined predicates This category consists of the predicates appearing in a Cob class and the translated predicates corresponding to a class.

- **User-defined CLP predicates:** A Cob program can contain predicate definitions under the `predicates` part of a class definition. Although these predicates can be invoked by passing them Cob program variables, their definitions cannot involve general Cob terms (such as class attributes and complex identifiers or quantified terms) but only the usual logic variables. These predicates are translated as is by the compiler (except for some internal renaming to avoid name clashes between classes). The syntax of these predicates is the full fledged CLP syntax and hence these are like any other CLP predicate.
- **Predicates corresponding to class definitions or constructors:** Each class definition in a Cob program is translated to one or more predicates. A Cob program with n class definitions where the i^{th} class has c_i constructors is translated into a CLP program with at least one predicate for each class and a predicate (clause or definition) corresponding to each constructor. Thus the number of such predicates is $\sum_{i=1}^n \min(1, c_i)$.

Clearly, the CLP builtin predicates and the Cob builtin predicates for array accessing are predefined predicates and hence do not give rise to any constraints. Except for the user-defined CLP predicates, the CLP program obtained from the translation of a Cob program thus has a known form, i.e., the format of the definitions etc. This knowledge enables us to predict some properties of the translated programs.

5.1.3 Properties of Translated Programs

The CLP programs obtained from translation of Cob program have a known structure. We make some practical assumptions about the CLP program that are justified in the context of using constrained objects for modeling engineering structures.

1. We assume that there can be only a finite depth of predicate invocations, for predicates derived (or translated) from class definitions.
2. We assume that the user-defined predicates in a Cob program terminate on all invocations.

With the above assumptions, the CLP program obtained by translating a Cob program has the following properties.

Property 5.1 (Termination): The CLP programs resulting from the translation of Cob models of engineering structures have the property that their predicates have only finite depth recursion.

Justification: We need to show that the predicates in each category listed in Section 5.2.2 terminates. Due to assumptions (i) and (ii) above, it is clear that the category of user-defined predicates terminates on all invocations. The CLP builtin predicates `expand`, `print`, `tell`, `told`, etc. are known to terminate. Of the third category of predicates, viz., the Cob compiler builtins, the `cc`, `makearray`, `makelist` and `index` are predefined and even though the definition is recursive, they are known to terminate on all invocations. The predicates resulting from quantified constraints and terms (`forall`, `exists`, `sum`, etc.) are recursively defined, with the recursion occurring on the argument representing enumeration (an array or list of integers). Since the arguments cannot be infinite lists, all invocations of these predicates also terminates. Hence, every predicate in the translated CLP program has only a finite depth recursion.

Property 5.2 (CLP subset): The predicates, constructs and queries appearing in the CLP program and its evaluation are a subset of the CLP(R) language.

Justification: This property follows straightforwardly from the fact that the translated CLP program contains predicates from the categories of Section 5.2.2. These predicates are either predefined or generated by the compiler and hence their structure is known a priori or from the syntax of the Cob program. A query to the CLP program is always obtained by translating a call to a Cob class constructor as shown in Section 5.2.2 and hence its format is also well known. Every predicate invocation in the translated program arises from translating a Cob constructor invocation (unless it is a call on user-defined CLP predicate). Hence we can say that the constructs and predicates appearing in the translated CLP program are a subset of the CLP language and their form is well-known.

Besides the above properties, the following are some straightforward outcomes of the translation.

Property 5.3 (Identifying Constraints): The constraints in a CLP-translation that correspond to the constraints of the original Cob program can be identified from those that are generated as a result of the translation.

Justification: The constraints of the original Cob program are straightforward to identify in the CLP-translation since they are explicitly enclosed in `{}`. Although the use of the `expand/0` predicate generates some equalities (not enclosed in parentheses) in the CLP translation in addition to the original Cob constraints, it is possible to identify these since they involve CLP internal variables (not present in the Cob program). Conditional constraints can be easily identified since they are handled by a special built-in predicate.

Non-logical constructs. Translated CLP programs do not have cuts except in definition of the compiler built-in predicate `index`. Negated goals, which can appear only in conditional constraints, are evaluated only when ground.

The above properties tell us that the CLP-translations are well structured subsets of the CLP language in which constraints are easy to identify and knowing which predicates will not and may have constraints allows us to design a strategy for partial evaluation of such programs discussed in Section 5.3.2.

5.2 Partial Evaluation

Partial evaluation is a pre-execution of the source program at compile time in order to obtain a more efficient execution in terms of space/time. Traditional techniques for code optimization include loop unraveling and data flow analysis [3] and have been used in languages such as FORTRAN and C. In logic programming, partial evaluation refers to a source-to-source transformation of a program with respect to a query such that for all instances of the query, the output program runs more efficiently than the original. In the context of Cob programs, we provide a scheme for partial evaluation of their CLP-translations with respect to a query.

As mentioned earlier, given the CLP-translation of a Cob program, a query can be evaluated directly on a CLP interpreter ([63] or [104]). Due to the limitations of the CLP solver, such a computation will return answers only when the constraints are sufficiently

linear. Also, the computational model of CLP(R) checks for consistency of the constraint store every time the constraint store is modified. This means that for every iteration of a loop (in the translated code) corresponding to a quantified constraint or term the constraint store is tested for consistency. For large-scale models this repeated checking of constraints and the presence of a large number of variables (including those generated as a result of the translation) may lead to a slowdown of the evaluation. Thus, execution of the translated CLP predicates directly on CLP(R) may not yield adequate performance for large-scale systems. Hence we have developed a partial execution technique for optimized code generation. The optimized code is run on a system which consists of a CLP-engine, a powerful constraint solver like Maple, and our conditional constraint evaluator.

5.2.1 Strategy

The execution of a Cob program is initiated by invoking the constructor of some Cob class, e.g., `X = new c(Args)`. The evaluation of such a call involves the creation of instances of constrained object classes and solving of their internal as well as interface constraints. Answers are obtained by evaluating the CLP-translation of the query with respect to the CLP-translation of the program. In general, this evaluation combines goal reduction with constraint solving (recall the operational semantics of CLP given in Sections 2.2.1 and 4.2.1). For large scale models, with large numbers of variables and constraints, this scheme may be inefficient since it involves repeated checking for consistency of constraints.

Our approach to a more efficient evaluation of the CLP-translation with respect to a query performs goal reduction to obtain all the constraints underlying a Cob model without solving any of the constraints. Different instances of the query can then be evaluated by solving the corresponding instances of these constraints. Separating goal reduction from constraint solving leads to a more efficient evaluation than when the two are interleaved.

In order to collect constraints without processing (solving) them, we require knowledge about where and in what form the constraints occur in the CLP-translation of a Cob program. The classification of predicates in a CLP-translation given in Section 5.2.1 provides us with this information and Property 5.3 gives us a means of identifying these constraints. With this knowledge, we can design a meta-interpreter that inspects a goal to

decide whether it should be evaluated or added to the collection of constraints. Since the types of predicates/goals that can appear in a CLP translation is finite and well known (Property 5.2), such a meta-interpreter is written as a case analysis on the type of goal.

The partial evaluator takes as input a CLP program obtained by translating a Cob program and a query obtained by translating a constructor call, and returns a list of constraints. A solution to this list of constraints is a solution to the query. Formally, the partial evaluator is described as follows.

Input: A CLP(\mathcal{D}) program P satisfying the properties in Section 5.2.3 and a goal G .

Output: A set of constraints C such that $\mathcal{P}, \mathcal{D} \models C \rightarrow G$.

In the following discussion, it is understood that the CLP-translation of a Cob program is given as input to the partial evaluator. We do not show the input program explicitly as an argument to the predicate `parEval / 2` which defines the partial evaluator.

$$\begin{array}{l} \text{parEval}(G, C) \text{ :-} \\ \text{parEval}(G, [], C). \end{array}$$

The predicate

`parEval(+Goal, +InputConstraints, -OutputConstraints)`

takes the input `Goal` and an initially empty list of constraints `InputConstraints` and returns the list of constraints of the Cob model in `OutputConstraints`. Its definition is given in Figure 5.6 and we describe below some of the interesting clauses from this definition.

- The result of partial evaluation of a pair of goals (A, B) is the union of the constraints resulting from partial evaluation of A followed by the partial evaluation of B .
- The CLP built-in predicates do not contain any constraints. The partial evaluator evaluates such goals to completion without further analysis. Similarly unification goals are also evaluated without further analysis.
- Of the Cob builtin predicates, we know from Section 5.2.2 that predicates for array declaration and accessing do not involve constraints. Hence these goals are evaluated to completion. Thus the partial evaluator processes away calls to predicates `index`, `makearray` and `makelist`.

```

parEval(true, C, C).
parEval((A,B), Cin, Cout) :-
    parEval(A, Cin, Cout1),
    parEval(B, Cout1, Cout).
parEval(G, C, C) :-
    clp_built_in(G), call(G).
parEval(G, C, C) :-
    is_unification(G), call(G).
parEval(G, C, C) :-
    cob_built_in_(G), call(G).
parEval(cc(A,B), Cin, Cout) :-
    evalCC(A,B,Result),
    (Result -> (Cin = Cout);
    (Result=callA -> Cout = [A | Cin];
    Cout = [cc(A,B) | Cin])).
parEval(A→B; C, Cin, Cout):-
    call(A)→parEval(B, Cin, Cout1);
    parEval(C, Cout1, Cout).
parEval(G, C, [G | C ] ):-
    predicate_property(G, imported_from(clpr)).
parEval({X = N}, C, C) :-
    var(X), number(N), call({X = N}).
parEval(G, Cin, Cout) :-
    clause(G,Body), parEval(Body,Cin,Cout).

```

Figure 5.7: Scheme for Partial Evaluation

- A conditional constraint is of the form $cc(A, B)$ and is evaluated using the predefined predicate `evalCC` shown below.

```

evalCC(A, B, true) :- entailed(A).
evalCC(A, B, callA) :- entailed(B).
evalCC(A, B, true) :- ground(B).
evalCC(A, B, cannot_prove).

```

A conditional constraint is satisfied if its consequent is entailed by the current state of the evaluation. In this case, the partial evaluation processes the next goal. On the other hand, if the antecedent is entailed by the current state of evaluation, then the consequent must be true for the conditional constraint to be satisfied and hence it is added to the current collection of output constraints. The third rule states that if the antecedent is false, then the conditional constraint is satisfied. However, the failure of the antecedent can be checked only if it is ground. In case none of the above rules is applicable, the conditional constraint is added to the list of output constraints to be evaluated at a later stage (when more information might be available). The predicate `entailed/1` is a SICStus Prolog builtin which evaluates to true if a given constraint is entailed by the current constraint store or state of the evaluation.

- A *constraint_atom* (see grammar in Appendix A) is detected using the builtin SICStus Prolog predicate `predicate_property`, which returns the value `imported_from(clpr)` for constraints. Equivalently, a constraint atom can be detected by the presence of surrounding parenthesis (Property 5.3). Constraints of the form $X = \textit{number}$ are evaluated (not collected) because they represent variable initialization. All other constraints are added to the existing list of output constraints (they are not evaluated).
- If the input goal is not handled by any of the above rules, it is resolved into a collection of goals using the rules in program P , i.e., goal reduction is performed with respect to the input CLP program. This is done using the CLP builtin predicate `clause/2`. The goals that fall in this category (i.e., they are reduced) are: calls on the user-defined predicates as well as the Cob builtin predicates for quantification and aggregation (e.g. `foralli`, `existsi`, `sumi`), etc predicates, there are goals

```

V1 = I1 * 10 = I2 * 20 = V2
V3 = I3 * 20 = I4 * 20 = V4
V1 = S.I * P1.R
S.I = I1 + I2
1/P1.R = 1/10 + 1/20
V3 = S.I * P2.R
S.I = I3 + I4
1/P2.R = 1/20 + 1/20
30 = S.I * S.R
30 = V1 + V3
S.R = P1.R + P2.R

```

Figure 5.8: Constraints obtained by partial evaluation of samplecircuit

of the form $A \rightarrow B;C$. This means that the partial evaluator performs loop unraveling to generate the set of constraints a quantified constraint represents.

Since there are no non-logical constructs, we do not have any case for them. The scheme for partial evaluation described above terminates on all inputs because, by Property 5.1, we know that the CLP program has only finite depth recursion and all invocation of the predicates in the CLP program terminate.

As an example of the result of the above scheme of partial evaluation, consider the samplecircuit class defined in Section 3.2.4. A query `S1 = new samplecircuit()` creates an instance of the sample circuit shown in Figure 3.3. This query is translated to `samplecircuit([],S1)`, and is partially evaluated to obtain the constraints shown in Figure 5.7. The full name of every variable above is actually prefixed by “S1.” since each variable is a local variable of S1. To avoid repetition, we do not show the prefix above. These constraints can now be solved to obtain values for the variables.

The partial evaluator described above generates optimized code and enables the handling of conditional as well as non-linear constraints. We give two examples to illustrate this point. The first example shows that the code resulting from partial evaluation is more

efficient than the original CLP program. It also demonstrates how the optimized code can be used for incremental constraint solving. The second example shows how partial evaluation can facilitate solving Cob models with non-linear and conditional constraints which otherwise cannot be handled by the CLP engine.

5.2.2 Optimization

Heat Transfer in a Plate. We give below a different Cob formulation of the Heatplate problem than the one given in Section 3.2.2. The problem is to model a plate in which the temperature of any point in the interior of the plate is the average of the temperature of its neighboring four points. This can be stated mathematically by using 2D Laplace's equations. In this formulation, we model every point on the plate as a `cell` with four neighboring cells. The value of the `Flag` attribute of the `cell` class indicates whether it is an interior cell (`Flag=1`) or not (`Flag=0`). If a cell is on the interior, its temperature `T` is related to its neighbors. This is modeled by the conditional constraint in class `cell`. The Cob representation of a 2D plate is shown below in a class called `heatplate`. Compared to its CLP(R) representation, the Cob representation of this problem is more comprehensible, and the quantified and conditional constraints make it reusable for different sizes of the plate. Note that we can also give an alternate model for this example using inheritance instead of conditional constraints to model the difference in behavior of a border and an interior cell. A description of such an alternate model is provided in Appendix D.

```
class cell {
  attributes
    cell Left, Right, Up, Down;
    real T, Flag;
  constraints
    (T = (Left.T+Right.T+Up.T+Down.T)/4) :- Flag=1;
  constructor cell(T1, F1) {
    T = T1; Flag = F1;
  }
}
class heatplate {
  attributes
    int Size;
    cell [Size][Size] Plate;
```

```

constraints
forall I in 2..Size-1:
  (forall J in 2..Size-1:
    (Plate[I,J].Up = Plate[I-1,J];
    Plate[I,J].Right = Plate[I,J+1];
    Plate[I,J].Left = Plate[I,J-1];
    Plate[I,J].Down = Plate[I+1,J]););
constructors heatplate(S, A,B,C,D) {
  Size = S;
  forall I in 2..Size-1:
    forall J in 2..Size-1:
      (Plate[I,J] = new cell(_, 1)););
  forall K in 1..Size:
    (Plate[1,K] = new cell(A,0);
    Plate[Size,K] = new cell(B,0)););
  forall L in 2..Size-1:
    (Plate[L,1] = new cell(C,0);
    Plate[L,Size] = new cell(D,0)););
}
}

```

The Cob compiler translates the above program to the CLP program P using the translations given in Sections 4.1.2 and 5.2.1. Suppose we want to compute the heat at all the interior points of a 4x4 grid and suppose that the temperatures along the top, bottom, left and right border of the grid are A , B , C and D respectively. We can partially evaluate the query

```
parEval(heatplate(4,A,B,C,D),Constraints).
```

to obtain the constraints:

$$X_1 = (A + X_2 + C + X_3) / 4,$$

$$X_3 = (A + X_4 + X_1 + D) / 4,$$

$$X_2 = (X_1 + C + B + X_4) / 4,$$

$$X_4 = (X_3 + B + X_2 + D) / 4$$

where X_i are the Cob program variables ($\text{Plate}[I, J].T$). The array operations (declaration and indexing) are processed away by the partial evaluator. The conditional constraints in the cell class are simplified by the partial evaluator since the value of the flag variable is known for all cells. Suppose the above set of constraints is denoted as $C(A, B, C, D)$. We can now obtain solution for any 4x4 grid by solving these constraints. For example, a

4x4 grid, with the top row set to 0 and the rest of the border set to 100, can be evaluated simply by solving the constraints $C(0, 100, 100, 100)$.

These constraints can also be displayed in terms of the original Cob program variables as:

```

Plate[3][3].T = (Plate[2][3].T+100+Plate[3][2].T+100)/4
Plate[3][2].T = (Plate[2][2].T+100+100+Plate[3][3].T)/4
Plate[2][3].T = (0+Plate[3][3].T+Plate[2][2].T+100)/4
Plate[2][2].T = (0+Plate[3][2].T+100+Plate[2][3].T)/4

```

By solving these constraints we get the values:

```

Plate[3][3].T = 87.5
Plate[3][2].T = 87.5
Plate[2][3].T = 62.5
Plate[2][2].T = 62.5

```

A clear advantage of partial evaluation is that the Cob program need not be recompiled for different instances of the query `heatplate(4,A,B,C,D)`. This is especially advantageous in the context of the domain specific interfaces. Once the modeler creates the drawing of a structure and it is compiled, subsequent changes to the attributes of the drawing can be solved without need for re-compilation of the diagram. Thus partial evaluation saves time in the *modify* and *resolve* phases of the Cob modeling environment.

Another source of performance improvement during goal evaluation is the separation of the goal reduction and constraint solving phases. Compared to the evaluation of the CLP-translation P , of the above program on SICStus Prolog directly, the partial evaluation of P followed by the solving of the collection of constraints takes less time.

Performance Results We compare the performances of the CLP-translation P of the above program with and without partial evaluation (PE). The points of the heatplate along the top, left, right and bottom border were initialized to 0, 100, 100 and 100 respectively for all queries. Table 5.1 shows the runtimes of the program for different sizes (N) of the heatplate. These tests were performed on a Sun Ultra Enterprise 450 Model 4400.

The first column in Table 5.1 specifies the size ($N \times N$) of the heatplate, the second column gives the runtime (in seconds) of the query `heatplate(N, 0, 100, 100, 100)` with respect to the CLP program P on SICStus Prolog, and the third column gives

Size of Heatplate (square)	Running time on SICStus Prolog (in seconds)	
	without PE	with PE
20x20	1	1 (0 + 1)
25x25	5	1 (0 + 1)
30x30	14	3 (1 + 2)
35x35	42	4 (1 + 3)
40x40	113	8 (1 + 7)
45x45	203	15 (2 + 13)
50x50	428	25 (2 + 23)
51x51	651	32 (2 + 30)
52x52	Resource error: insufficient memory	

Table 5.1: Comparison of Runtimes with and without Partial Evaluation

the runtime (in seconds) of the same query using partial evaluation and run on SICStus Prolog. The figures in the braces in the third column give the individual times for partial evaluation and the time for solving of the constraints. A 0 time indicates that the computation took a small amount of time which could not be measured (hence almost zero). Clearly, for all sizes of the heatplate, the total time taken for the partial evaluation and the constraint solving is less than the time taken to run the program without partial evaluation. The table shows that the time taken for partial evaluation (first element inside parenthesis) does not change significantly with the size of the heatplate. This indicates that even for large values of indices, the array accessing operation takes constant time. The figures in Table 5.1 indicate that partial evaluation leads to a performance improvement of 5 to 17 fold.

5.2.3 Conditional and Non-linear Constraint Solving

Truss Design. For the truss problem in section 3.2.1, we encounter several non-linear constraints. A concrete example of a truss and its Cob model from [68] is given in Appendix C. A sample truss is created using five steel beams each having a square cross-section. The length of the beams, the angle at which they are placed at a joint and the load at each joint is given. The problem is to determine the thickness required of each beam to be able to support the load. The query `T = sampletruss()` is partially evaluated to obtain a set of constraints. We show below the constraints for one of the instances of the beam (AB) and joint (JA) class:

```

Iab = -Fab*(12*10.4)^2/(3.141)^2*3.0E+07) :- Fab < 0
Fab = 3.0E+04*(W*H) :- Fab > 0
Iab = W1^4 / 12
Fab * cos(3.141/4) + Fac * cos(0) +
Fav * cos(3.141/2) + Fah * cos(0) = 0
Fab * sin(3.141/4) + Fac * sin(0) +
Fav * sin(3.141/2) + Fah * sin(0) = 0

```

These non-linear constraints cannot be solved in the CLP(R) system. Hence we make use of Maple [112], which is a computing system for interactive algebra, calculus, discrete mathematics, graphics, numerical computation and many other areas of mathematics. From the above set of non-linear constraints, we give the non-conditional constraints to Maple. The answers obtained include the value of `Fab` which is -13440.80489. Using this value, we can simplify the conditional constraints. The second conditional constraint is satisfied because its antecedent is false. The first conditional constraint simplifies to yield the constraint

```

Iab = -(-13440.80489)*(12*10.4)^2/(3.141^2*3.0E+07)

```

With this value substituted in the set of constraints above, we get a simplified set of constraints which is then given to Maple to obtain the following solutions.

```

S1.AB.W = 1.70684719
S1.AB.H = 1.70684719

```

Thus the conditional constraints are evaluated alternately with the set of non-conditional

constraints since the simplification of one may give sufficient information to solve the other. Such a scheme for handling conditional constraints and non-linear constraints is possible due to the partial evaluation which returns the set of constraints underlying the Cob model.

5.2.4 Queries that can be Partially Evaluated

In general, the evaluation of a Cob query involves the creation of instances of constrained object classes and the solving of their internal as well as interface constraints. In the context of engineering modeling, the creation of object instances and aggregating them to form other complex objects represents the process of assembling the components of a complex engineering structure together to form a specific configuration. We would like to identify the class of Cob queries for which the scheme for partial evaluation given in Section 5.3.1 will return a correct answer (set of constraints).

The most important aspect of the partial evaluation is loop unraveling and processing array accessing operations. For the partial evaluator to process away these operations or loops, it must have sufficient information about them. For example, array accessing requires the value of the index. Similarly, the enumeration (list or array) over which quantified constraints and aggregation terms iterate must be a known finite (ground) list. Thus, queries that can be partially evaluated have a groundness requirement that allows the processing of loops and array operations.

In the context of engineering structures, we can understand the above groundness requirement as follows. A call to a Cob class constructor may represent either a specific configuration or a collection of different configurations depending upon the value or groundness of its arguments. In general, the arguments to a constructor may determine the type, arrangement and attributes of the components of a configuration. Queries that can be partially evaluated represent a family of structures all having the same configuration but different values for attributes. Thus, every instance of the query must result in the same assembly but possibly different values for the attributes of the assembly.

5.3 Interactive Execution

The result of partial evaluation of a Cob program or model is a list of constraints. For large programs, if a constraint fails, it can be difficult to trace the error back to the object that that gave rise to the constraint. We introduce the notion of a constrained object graph which serves as a meaningful representation of constraints and can be used for interactive execution and debugging of partially evaluated Cob programs.

Constrained Object Graph. A complex object may be depicted by an object graph. Each node represents an object (i.e., instance of some class) and each directed edge represents an aggregation relationship. Replacing each node of the object graph with the constraints of the object it represents results in the formation of the constrained object graph. In the case of Cob models of engineering artifacts, the structure of this graph implicitly represents the assembly of components of the artifact.

We have implemented a tool that generates the constrained object graph for Cob programs. Figure 5.8 shows the constrained object graph for the `samplecircuit` defined in Section 3.2.4 and shown in Figure 3.3. In this example, all instances of resistors, battery, etc are created in the class `samplecircuit`. In general this need not be the case. The components `R1` and `R2` might be created inside the `parallel` class. In either case, the object graph will be identical to the one in Figure 5.8. This is because, the edges of the object graph denote aggregation and do not indicate where the instance was created.

Augmented Translation. In order to build an object graph, the translation must include information regarding object creation and aggregation a map of Cob variable names and their corresponding CLP variables. The translation shown in Sections 4.1.2 and 5.2.1 is augmented in order to store this information. We show below only the modified parts of the translation. The rest of the translation is identical to that given in Sections 4.1.2 and 5.2.1.

The translator inserts calls to predefined predicates for building the object graph, collecting constraints at every node and maintaining information about which node is currently being evaluated (to indicate the node to which subsequent constraints will belong).

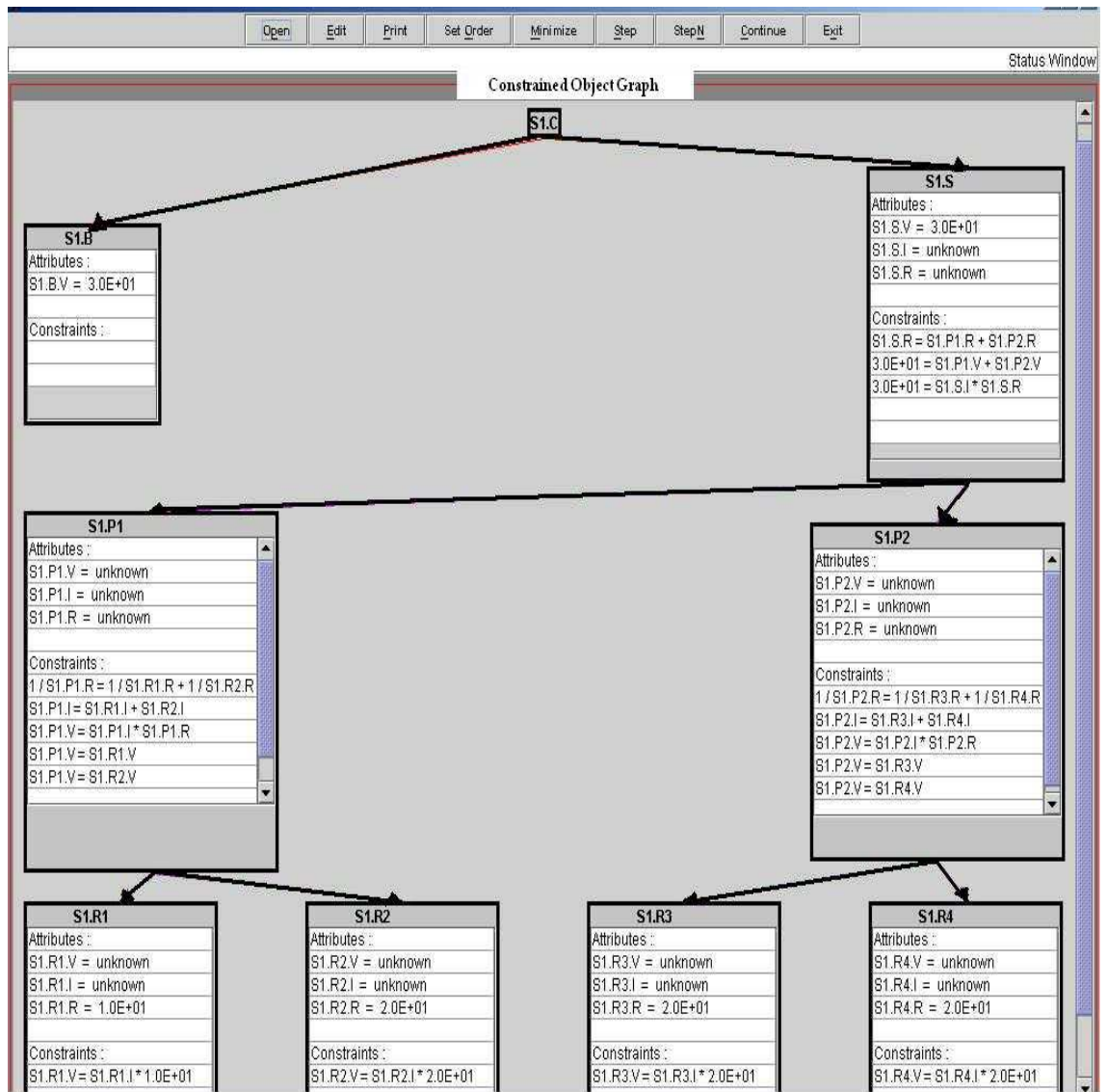


Figure 5.9: Object Graph of Samplecircuit

<pre> class c { attributes V₁ constraints C₁ predicates P₁ constructor c(V₂){ C₂ } } </pre>	\implies_T	<pre> P₁ ∪ {defn(p_c)} where defn(p_c) is: p_c(V₂, V₁) :- edge(V₁) , mapvar(V₁, NamesV₁) , C₁, C₂. </pre> <p style="text-align: center;">and</p> <p style="text-align: center;">NamesV₁: the names of the variables in V₁</p>
--	--------------	---

Creational constraints are translated as shown below. Calls to the predefined predicates `in` and `out` are placed before and after the creational constraints. This indicates to the partial evaluator that a new node is being created and that all the subsequent constraints between these two calls belong to one node.

$$\begin{aligned}
X = \text{new } c(\bar{t}) \quad &\implies_T \quad \text{in}(X_T), \\
&\quad p_c(\bar{t}_T, X_T) \\
&\quad \text{out}(X_T).
\end{aligned}$$

where p_c is the translation of class c

X_T is the translation of X

and \bar{t}_T is the translation of \bar{t}

Partial Evaluator built_ins: We refer to the predicates `edge`, `mapvar`, `in`, `out`, as the partial evaluator `built_ins`. `edge` is used for building the edges of the object graph. `mapcompoundvar` is used for building the CobCLP variable map for variables of the kind `X.Y`

`in`, `out` are used for building the Cob to CLP variable map and as the beginning and end markers for distinguishing the constraints of each node.

Augmented Partial Evaluator. The partial evaluator is also modified in order to build the object graph. The predicate `parEval/3` defined in section 4 is changed changed to `parEval/7`. It maintains a map (`Min`, `Mout`) of variable names and keeps track of the current node (`Nin`, `Nout`) of the object graph.

parEval(G, Cin, Cout, Min, Mout, Nin, Nout) :-

par_eval_built_in(G),

call_approp(G).

On all goals except the partial evaluator *built_ins*, the partial evaluator behaves as before. The partial evaluator *built_ins* are called appropriately using the current state of the object graph, the variable map, and the current node being visited. The output of this partial evaluator is used to build the constrained object graph.

5.3.1 Interactive Solving

We have developed a tool that takes as input a Cob program and displays its constraint object graph (COG) through a visual interface that can be used to interactively solve and debug. Once the constrained object graph is created, the *solve* phase of the Cob model can be observed and/or controlled via this interface. The solving of constraints is done on a per node basis, i.e., the constraints of node are solved then the next and so on. This graphical user interface of this tool was implemented by the following persons under the author's supervision: (i) Anuroopa Shenoy, a recent graduate student of the Department of Computer Science and Engineering, University at Buffalo, who implemented the basic graphical user interface and provided a visual display of the text-based representation of a graph; (ii) Anantharaman Ganesh, currently a graduate student of the Department of Computer Science and Engineering, University at Buffalo, who enhanced the graphical user interface by providing more features for user inputs, improving the graph display and suggesting ways of making the constraint solving process more interactive.

The tool is equipped with the following features and menu buttons.

- *Node Information*: Each node displays the constraints as well as the attributes (and their values) for the instance it represents.
- *Solve*: The constraint solving phase (after partial evaluation) can be traced on the COG. The tool highlights the node whose constraints are being solved. During the solving, every time a variable gets a value, it is displayed on the COG.

- *Step*: The modeler can step through the solve phase by clicking on the *Step* button, i.e., for every click one node of the COG is solved. The tool uses a default order in which to solve the nodes which is a bottom-up order.
- *StepN*: By clicking on this button, the modeler can specify a number (N) of nodes that should be solved without displaying the intermediate state of the COG.
- *Continue*: By clicking on this button, the modeler indicates that the solving should continue to completion, i.e., all nodes should be solved and the end result displayed.
- *SetOrder*: The programmer can specify the order in which the nodes should be solved, or that the constraints of only a subgraph should be solved. The programmer can choose from top-down, bottom-up, left-right and right-left or explicitly specify an ordering of the nodes.
- *Minimize*: For large graphs, a display of the entire graph with all the details of each node may be cluttered and incomprehensible. The *Minimize* button can be used to hide the details of a node or a subgraph of the COG.

5.3.2 Interactive Debugging

In general, an error in constraint solving might be caused due to an incorrect initialization of a variable, or an incorrect constraint. The tool described in the previous subsection facilitates a novel means by which the modeler can be assisted in understanding the possible cause of error in an over-constrained system. The following features are provided in the tool for aid in debugging. The visual aspects of these features were developed by Anantharaman Ganesh, currently a graduate student of the Department of Computer Science and Engineering, University at Buffalo.

- The programmer can step back or forward through the solve process by any number of steps. This can help the modeler understand the effect of solving as well as help detect an unexpected value for a variable.
- If an error occurs while solving, the node whose constraint fails is flagged.

- Simply knowing which node's constraint(s) failed may not be sufficient to determine the cause of error. By providing different orders of solving, which may result in different locations of the error, the programmer will be in a better position to understand the source of error.

Currently the tool can handle only acyclic constrained object graphs. Although this tool currently serves as a simple aid for understanding execution and errors of a Cob program, it sets the stage for a giving a unique view of the running and debugging of constrained object programs. For example, as a part of ongoing work, we will be providing a trace of the nodes which are involved in assigning a value to a variable. We will also need to address

5.4 Comparison with Related Work

Partial evaluation of logic programs has been studied widely [1, 74, 97, 59, 53]. The techniques used to perform partial evaluation include program transformations using unfolding, folding and definition [97]. Given a program and a query, they return a program specialized for all instances of the query. The program returned by partial evaluation is expected to be more efficient than the original program. A scheme for partial evaluation of constraint logic programs [59] is discussed in [53] which uses the fundamental transformations: specialization, unfolding and factorization. Specialization replaces a query with a call to a specialized version of the predicate. Unfolding is done by resolving a goal with the body of the specific clause it matches. Redundant code is eliminated and constraint simplification is done by detecting tautologies and contradictions.

Specialization techniques can be applied to constraint logic programs to obtain determinism, i.e., only one clause is applicable for goal reduction at a time. Again, this too uses specialized fold, unfold and partitioning operations [28]. A description of the fold and unfold operations for CLP with dynamic scheduling is given in [25].

Some of the operations in our scheme for partial evaluation can be considered as instances of the more general program transformations such as unfolding. However, our partial evaluator differs from other implemented schemes for partial evaluation of constraint

logic programs because we deal with a special class of CLP programs. These are programs obtained as a result of translating Cob programs. The different categories of predicates that appear in such programs is known apriori: Cob built-in predicates, conditional constraints, and quantified constraints, and this knowledge tells us where to expect constraints and which subgoals to process away.

5.5 Summary

The implementation of constrained objects involves two main steps: translation of Cob classes to CLP predicates, and partial execution of the translated program. Cob classes essentially represent abstract data types, and the translation to CLP predicates is quite natural, compact, and straightforward. The translated predicate can be considered as a procedure to enforce the constraints of a class. The translated programs however, are not suitable for immediate execution. There are overheads arising from the representation of arrays by lists and indexing by list traversal; and also due to the loop iterations caused by quantified constraints. Given a query, partial evaluation allows us to process away these operations and returns a set of constraints. A solution to these constraints is a solution to the query. Different instances of the query can be evaluated by solving the corresponding instances of this set of constraints.

Based on partial evaluation and the concept of constrained object graph, we provide a unique view of Cob program execution which can also be used for understanding the source of errors in an over-constrained system. We have developed an implementation of all the techniques presented in this chapter. Our initial experiments with partial evaluation shows that there is a 5-17 fold improvement in performance. We have also found the interface to Maple to be a very effective means of handling nonlinear constraints.

Chapter 6

Constrained Objects with Preferences

This chapter extends the paradigm of constrained objects to model optimization problems. Often a constrained object model of a complex system has multiple solutions, but some solutions may be considered better than others. In our constrained object paradigm, we provide constructs for the modeler to express a preference between solutions. The preferences may specify minimization/maximization criteria (as in linear programming) or they may be of a more general form discussed in [40]. In the presence of a preference, the resultant optimal state of the constrained object is obtained by employing constraint satisfaction and optimization techniques.

We extend the basic paradigm of constrained objects given in Chapter 4 to include preferences, along the lines of preference logic programs [39, 40, 66, 41] described in Section 2.3. A preference logic program is a logic program with two types of predicates: ordinary predicates and optimization predicates. The latter are defined using a combination of ‘optimization clauses’ and ‘preference clauses’. We map a Cob class with preferences to a PLP predicate and give a set-theoretic semantics of a Cob program in terms of the preferential consequences of the corresponding PLP program.

When two optimization problems are combined or interdependent, or an optimization problem has multiple preference criteria, an optimal solution to one may not form an optimal solution to another. In such cases, and also in general, one may be interested in the optimal as well as suboptimal solutions to a problem. Thus an optimization problem may be subject to relaxation, which is a technique for finding suboptimal solutions. This chapter

also addresses the computational problems arising from relaxation of constrained objects with preferences as well as relaxation in the paradigm of preference logic programming (PLP) [38]. We explore different forms of relaxation, introduce a more general the concept of a relaxation goal, and present two different variations of this concept: relaxation with respect to the underlying preference order as well as relaxation with respect to additional constraints on the optimization goal (i.e., ‘what-if’ relaxation).

The execution of preference logic programs is based upon SLD search trees extended with a preference structure which is determined from the preference clauses of the program [38]. This model crucially depends upon memo-tables for efficiency. We present a reformulation of the above model using rewrite rules that facilitates an extension of the model to include the evaluation of relaxation goals.

The material in this chapter is organized as follows. In Section 6.1 we extend the syntax of the Cob language to express preferences and give an example of its use in engineering modeling. This section also describes the declarative and operational semantics of Cob programs with preferences. We also give a reformulation the operational semantics of PLP programs in terms of rewrite rules. Section 6.2 introduces the notion of relaxation of constrained objects with preferences and gives the syntax and motivating examples of the different relaxation constructs. In Section 6.3 we propose an operation semantics for PLP programs with relaxation goals. Section 6.4 gives a comparison with related work.

6.1 Cob Programs with Preferences

6.1.1 Syntax

We extend the syntax described in Section 3.1 to express preferences in Cob programs.

Class. A class definition now has the following structure.

$$\begin{aligned} \textit{class_definition} & ::= [\textit{abstract}] \textit{class } \textit{class_id} [\textit{extends } \textit{class_id}] \{ \textit{body} \} \\ \textit{body} & ::= [\textit{attributes } \textit{attributes}] \\ & \quad [\textit{constraints } \textit{constraints}] \end{aligned}$$

```

[ predicates pred_clauses ]
[ preferences pref_clauses ]
[ constructors constructor_clause ]

```

There are two forms of preference clauses:

1. A preference clause can express an objective function (essentially an arithmetic expression) and state whether it is to be minimized or maximized.

```

pref_clauses ::= pref_clause . | pref_clause . pref_clauses
pref_clause ::= min arithmetic_expr.
pref_clause ::= max arithmetic_expr.

```

2. Another form of expressing preferences augments a predicate definition with preference criteria that define the optimal solution(s) to goals that make use of this predicate. Such a preference clause has the following syntax in Cob.

```

pref_clause ::=  $p(s1) \leq p(s2) :-$  clause_body

```

This clause states that the solution $s1$ is less preferred than solution $s2$ for predicate p if the condition specified by *clause_body* is true. The non-terminal *clause_body* is defined as before in Section 3.1. This form of preference clause is discussed in [38] in the context of preference logic programs.

The non-terminals *attributes*, *constraints*, *pred_clauses* and *constructor_clause* are defined as before in Section 3.1. We now give an example from the engineering domain that involves multiple solutions and illustrates the use of the preference syntax in Cob. This example was developed with the help of Pratima Gogineni, a former graduate student of the Departments of Chemical Engineering and Computer Science and Engineering, University at Buffalo.

Separators and Mixers. A common problem in chemical engineering is the use of a combination of mixers and separators to produce a chemical that has specific ingredients in a certain proportion. The arrangement of mixers and separators in Figure 6.1 has two input

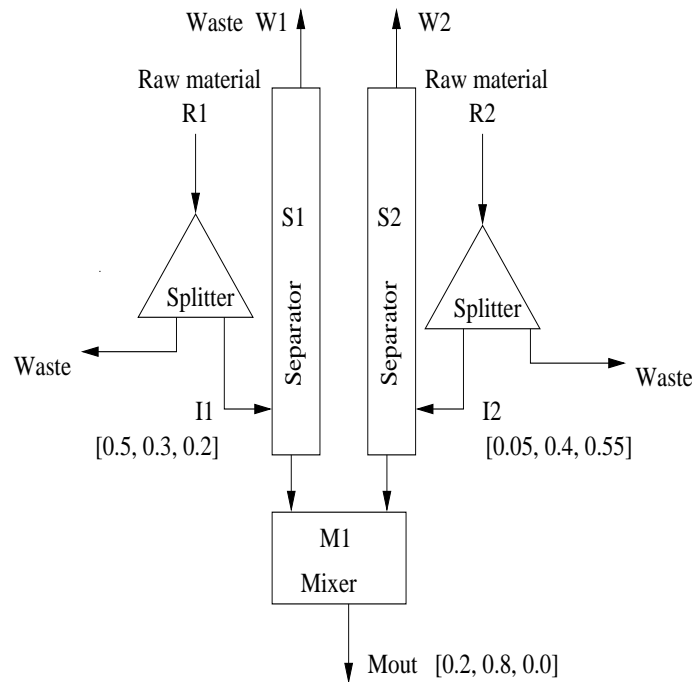


Figure 6.1: A Separation Flow Sheet

raw material streams R1 and R2. Each of these streams has certain ingredients in different concentrations. R1 and R2 are split and a part of each (I1 and I2 respectively) is sent to a separator which separates its ingredients. Each separator supplies certain proportion of each ingredient to the mixer which combines them to produce Mout, the desired chemical. W1 and W2 are waste streams from the separators. The problem is to produce Mout while minimizing I1 and I2 thereby minimizing the cost of processing material in the separators.

Figure 6.1 describes a typical scenario, and we present below some of the key classes needed for this example. A stream is modeled by the class `stream` with attributes for its rate of flow `FlowRate` and the concentrations of its ingredients (`Concentrations` is an array of reals indexed by the ingredients of the stream). The concentrations of all the ingredients of a stream must sum up to 1.

```
class stream {
  attributes
    real FlowRate;
    real [] Concentrations;
  constraints
    sum C in Concentrations:C = 1;
```

```

constructors stream(Q, C) {
  FlowRate = Q; Concentrations = C;
}
}

```

The class equipment models any piece of equipment having some input streams and output streams. Every equipment that processes streams is constrained by the law of mass balance. Separators, mixers and splitters are instances of the equipment class.

```

class equipment {
  attributes
    stream [] InStream, OutStream;
    int NIngredients;
  constraints          % law of mass balance
    forall I in 1..NIngredients :
      (sum J in InStream : (J.FlowRate * J.Concentrations[I])) =
      (sum K in OutStream: (K.FlowRate * K.Concentrations[I]));
  constructors equipment(In,Out,NumIng) {
    InStream = In; OutStream = Out; NIngredients = NumIng;
  }
}

```

The class sampleFlowSheet representing Figure 6.1 creates instances of separators and a mixer. The variables representing output streams of one equipment are made the input streams of another appropriate equipment. The class also has the preference: $\min I1.FlowRate + I2.FlowRate$. This states that the constraints of the model should be satisfied while minimizing the amount of raw material processed (which in turn will minimize the cost of the process). Given the the concentration of materials in each input stream and their desired composition in the output stream, such a Cob model can be used to determine the optimal amount of input raw material.

```

class sampleFlowsheet {
  attributes
    stream I1, I2, Slout, S2out, Mout, W1, W2;
    equipment S1, S2, M1;
    real Q1, Q2;
  constraints
    Mout.FlowRate = 150;
    Mout.Concentrations = [0.2,0.8,0.0];
    I1.FlowRate ≤ 500;
    I2.FlowRate ≤ 600;

```

```

preferences
  min (I1.FlowRate + I2.FlowRate).
constructors sampleFlowsheet() {
  I1 = new stream(Q1, [0.5, 0.3, 0.2]);
  I2 = new stream(Q2, [0.05, 0.4, 0.55]);
  S1 = new equipment([I1], [S1out, W1], 3);
  S2 = new equipment([I2], [S2out, W2], 3);
  M1 = new equipment([S1out, S2out], [Mout], 3);
}
}

```

6.1.2 Declarative Semantics

A Cob program containing preference clauses can be translated to a Preference Logic Program (PLP) in a natural way. To obtain a PLP program from a Cob program, we augment the translation of Cob programs to CLP (described in Chapters 3 and 4) with the following transformation for Cob classes that contain preferences.

In Cob programs with more than one correct answer, preferences are a way of specifying the criteria for the optimal answer(s). There are two ways of specifying preferences:

- using the min/max construct (see examples in Sections 6.1 and 7.1).
- using an explicit preference clause: $p(\bar{t}) \preceq p(\bar{u}) :- \bar{L}$.

The preference clauses of the first kind above (min/max clauses) are translated as shown below. We use V_i , C_i , P_i to denote collections of variables, constraints and predicates respectively.

class c {	\implies_T	$P_1 \cup \{defn(p_c)\}$
attributes V_1		where $defn(p_c)$ is:
constraints C_1		$p_c(V_2, V_1) :- C_1, C_2.$
predicates P_1		$p_c(X, V_1) \preceq p_c(X, V'_1) :- v \geq v'$
preferences min v		
constructor $c(\bar{V}_2)$ {		
C_2		
}		
}		

Thus we translate a Cob class with \min/\max preferences to an *O-predicate* with an arbiter clause whose body captures the preference criterion. The class c is translated to a predicate p_c whose first argument corresponds to the list of arguments of the constructor of class c and second argument corresponds to the list of attributes of class c . The body of p_c contains the class constraints and constructor constraints of c . This ensures that the attributes of the class satisfy these constraints. Until this point the translation is similar to the CLP-translation defined in Chapter 4. However, if a \min preference is present in the definition of class c , then it gives rise to an arbiter clause for p_c as shown above. For the same list of constructor arguments X , if the values V_1 for the list of attributes of class c give a value v for the objective function and if the values V'_1 amount to a value v' for the same objective function, and v' is less than v , then the set of values in V_1 is less preferred to V'_1 . The comparison in the body of the arbiter clause is reversed if the objective function is being maximized.

Preference clauses of the second kind are arbiters for the predicates of a Cob class. Since we translate predicates of a Cob class without any transformation, their arbiter clauses are also translated as is (with the necessary translation of subterms). We now give the declarative semantics of Cob programs with preferences.

Definition 6.1: *Suppose C is a Cob program containing preference clauses. The PLP program P obtained by translating each class of C using the above scheme is called the **PLP-translation** of C . For each class c in C containing preferences, the predicate p_c obtained by translating c is called the PLP-translation of c .*

The paradigm of PLP is capable of expressing both hierarchic as well as recursive optimization problems [38, 40]. References [38, 40] give a possible-worlds semantics for preference logic programs. We describe these semantics here briefly. Consider a predicate p with one or more preference clauses associated with it. The different models of p form different worlds, and an ordering over the worlds is determined by the preference clauses. The preferential consequences of the program are the set of atoms that are true in the *strongly optimal worlds*, i.e., worlds such that there are no other worlds that are better according to the preference clauses. This is in contrast to *logical consequence* which refers to truth in *all* worlds. A *strongly optimal answer* to the query G is a valuation θ such that

$G\theta$ is a *preferential consequence* of the preference logic program.

When preferences are specified in a Cob class definition, we determine the set-theoretic semantics of the class from the set of preferential consequences of the corresponding PLP predicate.

Definition 6.2: *Given a Cob program C containing preferences, if its PLP-translation is P , then the **declarative semantics** of C is defined to be the set of preferential consequences of the program P .*

Definition 6.3: *Given a Cob program C with preferences and a query $X = \text{new } c(\bar{t})$, if P is the PLP-translation of C and the goal corresponding to the constructor call is $G \equiv p_c(\bar{t}, X)$, then the **correct optimal answer** to G is a valuation θ such that $G\theta$ is a preferential consequence of P .*

Note that the PLP-translation of a Cob program C without preferences is equivalent to the CLP-translation of C .

6.1.3 Operational Semantics

The operational semantics of a Cob program with preferences can be understood as the operational semantics of its PLP-translation. The operational semantics of PLP given in [38, 40] was based upon a scheme called Pruned Tree SLD resolution (PTSLED). In essence, PTSLED-derivations incrementally construct the search tree from a goal and use the preference clauses as advice to prune paths that compute non-optimal answers. Essentially a tree T_1 **derives** a tree T_2 , written $T_1 \Rightarrow T_2$, if T_2 is obtained from T_1 by reducing a goal in a leaf of T_1 in all possible ways and creating the corresponding offspring nodes. Paths that compute sub-optimal solutions are pruned by consulting the preference clauses. We refer the reader to [38, 40] for the details of the operational semantics.

In this section we reformulate these semantics by describing them in terms of rewrite rules. This enables us to give a concise operational semantics for PLP with relaxation goals (introduced later in Section 6.2). Recall the overview of preference logic programs given in Section 2.3. A PLP program P is represented by the tuple $\langle \mathcal{T}_C, \mathcal{T}_O, \mathcal{A} \rangle$, where \mathcal{T}_C is the set of *C-predicates*, \mathcal{T}_O is the set of *O-predicates* and \mathcal{A} is the set of arbiter clauses of the PLP program. Given a goal G , we represent a PTSLED derivation of G with respect to P as

a sequence of states.

Definition 6.4: A **state** is a 2-tuple $\langle S, P \rangle$ of sets. The first element S represents the set of non-failed, non-pruned paths in the PTSLD resolution tree. The second element P of the tuple represents the pruned paths of the PTSLD resolution tree.

Definition 6.5: A **non-pruned, non-failed path** is represented by a 3-tuple $\langle L, C, \wp \rangle$. The first element L is the list of the literals at the leaf node of the path. The second element C is the constraint store, i.e., the collection of constraints encountered along the path. The third element \wp represents the ordered sequence of O -predicates that were expanded along the path (the leftmost element representing the first and the rightmost representing the most recent).

Each element in the sequence \wp uniquely identifies the name and occurrence of the O -predicate in the PTSLD tree.

Definition 6.6: A **pruned path** is represented by a 4-tuple $\langle L, C, \wp, \ell \rangle$. The first 3 elements are identical to those in a non-pruned path and the fourth element ℓ is a label indicating the name, level, occurrence and depth (in the PTSLD search tree) of the O -predicate by which the path was pruned.

The label corresponding to an occurrence of a predicate also indicates the depth of the predicate occurrence in the search tree and its level in the PLP program. Each predicate in a PLP program is associated with a positive integer called its level which is the least such integer that satisfies the following condition. If predicate p appears in the definition of a predicate q , then the level of p is less than the level of q . The start state of a PTSLD derivation for a goal G with respect to a PLP program P is

$$G_o \equiv \langle \{ \langle G, \in, \in \rangle \}, \phi \rangle$$

We now describe the transition rules that transform one state into another.

Let $G_i \equiv \langle U, P \rangle$ represent the state of a PTSLD derivation where

$$U \equiv \{ \langle S_1, C_1, \wp_1 \rangle, \dots, \langle S_n, C_n, \wp_n \rangle \} \text{ and}$$

$$P \equiv \{ \langle S'_1, C'_1, \wp'_1, \ell_1 \rangle, \dots, \langle S'_m, C'_m, \wp'_m, \ell_m \rangle \}$$

The different types of transitions are :

1. **Expand O -predicate:**

$$\langle \{ \langle S_1, C_1, \wp_1 \rangle, \dots, \langle \{a\} \cup S_i, C_i, \wp_i \rangle, \dots, \langle S_n, C_n, \wp_n \rangle \},$$

$$\begin{aligned}
& \{ \langle S'_1, C'_1, \wp'_1, \ell_1 \rangle, \dots, \langle S'_m, C'_m, \wp'_m, \ell_m \rangle \} \rightarrow_r \\
& \langle \{ \langle S_1, C_1, \wp_1 \rangle, \dots, \langle S_{i-1}, C_{i-1}, \wp_{i-1} \rangle, \\
& \quad \langle B_1 \cup S_i, (C_i \cup \alpha_1), (\wp_i p_u) \rangle, \dots, \langle B_k \cup S_i, (C_i \cup \alpha_k), (\wp_i p_u) \rangle, \\
& \quad \langle S_{i+1}, C_{i+1}, \wp_{i+1} \rangle, \dots, \langle S_n, C_n, \wp_n \rangle \}, \\
& \{ \langle S'_1, C'_1, \wp'_1, \ell_1 \rangle, \dots, \langle S'_m, C'_m, \wp'_m, \ell_m \rangle \} \rangle
\end{aligned}$$

if $a = p(\bar{t})$ is an atom selected by the computation rule, and p is an *O-predicate* such that the following holds. Let $h_1 \leftarrow B_1, \dots, h_k \leftarrow B_k$ be all the rules of program P such that a and h_i have the same outermost predicate. If \bar{t} represents the terms t_1, \dots, t_n , we use the notation $\bar{s} = \bar{t}$ to denote the set of constraints $s_1 = t_1, \dots, s_n = t_n$. Now suppose that for $j \in 1..k$, $h_j = p(\bar{t}_j)$, then α_j is the set of constraints $\bar{t} = \bar{t}_j$ and p_u uniquely identifies the occurrence of atom a .

2. Expand C-predicate:

$$\begin{aligned}
& \langle \{ \langle S_1, C_1, \wp_1 \rangle, \dots, \langle \{a\} \cup S_i, C_i, \wp_i \rangle, \dots, \langle S_n, C_n, \wp_n \rangle \}, \\
& \quad \{ \langle S'_1, C'_1, \wp'_1, \ell_1 \rangle, \dots, \langle S'_m, C'_m, \wp'_m, \ell_m \rangle \} \rightarrow_r \\
& \langle \{ \langle S_1, C_1, \wp_1 \rangle, \dots, \langle S_{i-1}, C_{i-1}, \wp_{i-1} \rangle, \\
& \quad \langle B_1 \cup S_i, (C_i \alpha_1), (\wp_i) \rangle, \dots, \langle B_k \cup S_i, (C_i \alpha_k), (\wp_i) \rangle, \\
& \quad \langle S_{i+1}, C_{i+1}, \wp_{i+1} \rangle, \dots, \langle S_n, C_n, \wp_n \rangle \}, \\
& \quad \{ \langle S'_1, C'_1, \wp'_1, \ell_1 \rangle, \dots, \langle S'_m, C'_m, \wp'_m, \ell_m \rangle \} \rangle
\end{aligned}$$

if $a = p(\bar{t})$ is an atom selected by the computation rule, and p is a *C-predicate* such that the following holds. Let $h_1 \leftarrow B_1, \dots, h_k \leftarrow B_k$ be all the rules of program P such that a and h_i have the same outermost predicate. Suppose that for $j \in 1..k$, $h_j = p(\bar{t}_j)$, then α_j is the set of constraints $\bar{t} = \bar{t}_j$.

3. Prune using non-pruned path:

$$\begin{aligned}
& \langle \{ \langle S_1, C_1, \wp_1 \rangle, \dots, \langle S_i, C_i, \wp_i \rangle, \dots, \langle S_n, C_n, \wp_n \rangle \}, \\
& \quad \{ \langle S'_1, C'_1, \wp'_1, \ell_1 \rangle, \dots, \langle S'_m, C'_m, \wp'_m, \ell_m \rangle \} \rightarrow_p \\
& \langle \{ \langle S_1, C_1, \wp_1 \rangle, \dots, \langle S_{i-1}, C_{i-1}, \wp_{i-1} \rangle, \langle S_{i+1}, C_{i+1}, \wp_{i+1} \rangle, \dots, \langle S_n, C_n \rangle \}, \\
& \quad \{ \langle S'_1, C'_1, \wp'_1, \ell_1 \rangle, \dots, \langle S'_m, C'_m, \wp'_m, \ell_m \rangle, \langle S_i, C_i, \wp_i, \ell' \rangle \} \rangle
\end{aligned}$$

if there exists a j such that $1 \leq j \leq n$ and $p(\bar{t})$ is an element of \wp_i and \wp_j such that $i \neq j$ and both S_i and S_j have finished solving for $p(\bar{t})$ and the following holds.

Suppose p is an O -predicate subject to an arbiter of the form:

$$p(\bar{u}) \preceq p(\bar{u}_1) \leftarrow L_1, \dots, L_k.$$

The above transition holds if the following goal is satisfiable:

$$\leftarrow p(\bar{t})\theta_i = p(\bar{u}), p(\bar{t})\theta_j = p(\bar{u}_1), L_1\theta_i\theta_j, \dots, L_k\theta_i\theta_j.$$

where θ_i and θ_j are the valuations obtained by restricting the constraints in C_i and C_j respectively to the variables in \bar{t} . Also $p(\bar{t})$ should be the rightmost such element of \wp_i and \wp_j . This ensures that we are pruning based on the expansion of the closest ancestor of the two leaf nodes i and j (this is a requirement of the semantics of PLP [40]). The label ℓ' denotes p and its level and its depth. The path $\langle S_i, C_i, \wp_i \rangle$ is said to be pruned.

4. Prune using pruned path:

$$\begin{aligned} &< \{ \langle S_1, C_1, \wp_1 \rangle, \dots, \langle S_i, C_i, \wp_i \rangle, \dots, \langle S_n, C_n, \wp_n \rangle \}, \\ &\quad \{ \langle S'_1, C'_1, \wp'_1, \ell_1 \rangle, \dots, \langle S'_m, C'_m, \wp'_m, \ell_m \rangle \} \rightarrow_p \\ &< \{ \langle S_1, C_1, \wp_1 \rangle, \dots, \langle S_{i-1}, C_{i-1}, \wp_{i-1} \rangle, \langle S_{i+1}, C_{i+1}, \wp_{i+1} \rangle, \dots, \langle S_n, C_n \rangle \}, \\ &\quad \{ \langle S'_1, C'_1, \wp'_1, \ell_1 \rangle, \dots, \langle S'_m, C'_m, \wp'_m, \ell_m \rangle, \langle S_i, C_i, \wp_i, \ell_j \rangle \} > \end{aligned}$$

if there exists a j such that $1 \leq j \leq m$ and $p(\bar{t})$ is an element of \wp_i and \wp'_j such that both S_i and S'_j have finished solving for $p(\bar{t})$, the label ℓ_j denotes p and its level and depth, and the following holds. Suppose p is an O -predicate subject to an arbiter of the form:

$$p(\bar{u}) \preceq p(\bar{u}_1) \leftarrow L_1, \dots, L_k.$$

The above transition holds if the following goal is satisfiable:

$$\leftarrow p(\bar{t})\theta_i = p(\bar{u}), p(\bar{t})\theta_j = p(\bar{u}_1), L_1\theta_i\theta_j, \dots, L_k\theta_i\theta_j.$$

where θ_i and θ_j are the valuations obtained by restricting the constraints in C_i and C_j respectively to the variables in \bar{t} . Also $p(\bar{t})$ should be the rightmost such element of \wp_i and \wp'_j . The path $\langle S_i, C_i, \wp_i \rangle$ is said to be pruned.

5. Fail:

$$\begin{aligned} &< \{ \langle S_1, C_1, \wp_1 \rangle, \dots, \langle a \cup S_i, C_i, \wp_i \rangle, \dots, \langle S_n, C_n, \wp_n \rangle \}, \\ &\quad \{ \langle S'_1, C'_1, \wp'_1, \ell_1 \rangle, \dots, \langle S'_m, C'_m, \wp'_m, \ell_m \rangle \} \rightarrow_f \end{aligned}$$

$$\begin{aligned} &< \{ < S_1, C_1, \wp_1 >, \dots, < S_{i-1}, C_{i-1}, \wp_{i-1} >, < S_{i+1}, C_{i+1}, \wp_{i+1} >, \dots, < S_n, C_n, \wp_n > \}, \\ &\{ < S'_1, C'_1, \wp'_1, \ell_1 >, \dots, < S'_m, C'_m, \wp'_m, \ell_m > \} > \end{aligned}$$

if a is an atom selected by the computation rule and there does not exist a rule of P such that a unifies with the head of that rule.

Definition 6.7: A state is said to be **final** if it cannot be transformed by any of the above transition rules.

Definition 6.8: A derivation for a goal G is said to be **successful** if its final state is:

$$< \{ < \{\}, C_{o_1}, \wp_1 >, \dots, < \{\}, C_{o_t}, \wp_t > \}, \{ < S'_1, C'_1, \wp'_1, \ell_1 >, \dots, < S'_m, C'_m, \wp'_m, \ell_m > \} >$$

and the **optimal computed answers** to G are the constraints: C_{o_1}, \dots, C_{o_t} .

We refer to the above derivation scheme as \mathcal{A} . The operational semantics described above are essentially a reformulation (with a different representation of a state) of those described in [40]. The soundness of PTS LD-derivations requires, among other things, that the predicates be sufficiently non-ground when invoked. To show the soundness of PTS LD-derivations, the notion of stratified preference logic programs is introduced [38]. Informally, a program is stratified if its *O-predicates* can be arranged in a linear order such that predicates that appear in the body of an *O-predicate* come below it in the linear order. If a program is stratified, then it can be shown that the above derivations do not prune any answers that are in a strongly optimal world [38]. Also, answers to an *O-predicate* goal of level i computed along different paths (i.e., present in different elements of U in the above derivations) correspond to instances of $p(\bar{t})$ present in distinct worlds in the intended preference models at level i [38]. Also if the arbiter applies to these answers, then the corresponding worlds are also related by the preference criterion.

The completeness of PTS LD-derivations requires that the search tree for every optimization goal be finite [38]. This is because the optimal solution cannot in general be computed unless the search tree is exhaustively explored. If an instance of an *O-predicate* goal is present in the intended preference model at the same level as the goal, then it can be shown that there is a PTS LD-derivation that computes that instance. This is because the candidate solutions to an *O-predicate* goal are obtained from the SLD-derivations which have the completeness property [73]. It can then be shown that if an arbiter applies between two computed answers, then the preference criterion applies to the corresponding worlds

in which these instances are present. The proof of both soundness as well as completeness is given by induction on the level of the *O-predicate* goals [38].

6.2 Relaxation in Cob

It is often the case that in constrained objects with preferences, one may be interested in the optimal as well as certain suboptimal solutions to the problem. For example, a designer may be interested in the five most efficient models of a gear train or say the three most cost effective designs for a car. In the case of multiple optimization criteria, a solution that is optimal based on one criterion may not be optimal with respect to another. Such cases naturally give rise to a need for relaxing the preference criterion to obtain a suboptimal solution.

A simple form of relaxation of optimization goals is discussed in [38] and we describe it here. Consider the `sh_path` predicate defined in Section 2.3.2. We can use this predicate along with the relaxation construct `RELAX_WRT`, to determine the second-shortest distance between a pair of nodes in a graph. Determining the second-shortest distance is a problem of sub-optimization, i.e., relaxation of the preference criterion. Suppose the shortest distance between nodes `a` and `b` in a graph is d . The following goal specifies the second shortest distance `C` and corresponding path `P` (if it exists) between nodes `a` and `b`.

?- RELAX sh_path(a,b,C,P) WRT C > d.

The above is a relaxation goal and states that the optimization goal `sh_path(a,b,C,P)`, should be relaxed with respect to (WRT) the constraint $C > d$. Note that if there are multiple paths having the same shortest distance, the second-shortest path(s) are the path(s) with next shortest distance.

To illustrate the power of the above `RELAX` clause, consider the recursive definition for the n^{th} -shortest path in a graph given in [38].

`n_sh_path(1,X,Y,C,P) ← sh_path(X,Y,C,P) .`

`n_sh_path(N+1,X,Y,C,P) ← n_sh_path(N,X,Y,D,-) ,`

`RELAX sh_path(X,Y,C,P) WRT C > D .`

The first clause states that the 1st shortest path is the same as the shortest path between two

nodes. The second clause states that an $(N+1)^{th}$ shortest path P from X to Y has distance C if there is an N^{th} shortest path of distance D between X and Y and P is a path obtained by the relaxation goal $RELAX \text{ sh_path}(X, Y, C, P) \text{ WRT } C > D$. In other words the optimization goal $\text{sh_path}(X, Y, C, P)$ is relaxed to give a path whose distance, C , is greater than the shortest distance D . For example, given a query, $?- \text{n_sh_path}(2, a, b, C, P)$, the computed answer for C will be the second-shortest distance between a and b .

The above program gives a concise, modular and declarative specification of the problem. Although the evaluation of such a goal makes multiple calls to the sh_path predicate, re-computation is avoided through the use of memoization. Memoization is a technique by which solutions to goals once computed are stored in a memo-table and solutions to subsequent calls to the same goal are obtained by looking up this memo-table.

The above is one form of preference relaxation. In this section we first present different forms of relaxation: relaxation with respect to a constraint, relaxation with respect to the underlying criterion, and relaxation to the n^{th} level (n^{th} suboptimal answers). We present examples motivating the use of these kinds of relaxation in constrained objects as well as in PLP. Since the semantics of constrained objects with preferences are given in terms of their PLP-translation, when giving the operational semantics of the relaxation goals in later sections, we restrict our attention to *relaxation goals* in preference logic programs (PLP). We follow the PLP paradigm outlined in the papers [39, 40, 41, 66], as it provides a reasonably general framework for exploring this concept.

In this section, we present the computational issues arising from relaxation goals. The simplest strategy for defining operational semantics of the $RELAX \text{ } p(\vec{t}) \text{ WRT } C$ goal is to dynamically augment the optimization clause for p with the condition C and then evaluate the augmented clause under the preference criteria for p . Such a strategy was outlined in [40]. However, this simple strategy is correct when the optimization predicate is defined in terms of ordinary predicates but not other optimization predicates. In the latter case, one might consider “pushing” the condition C down further, but we show that such a scheme could result in the inner optimization predicates to flounder.

6.2.1 Syntax

A **relaxation goal** can have the form

$$\text{RELAX } p(\bar{t}) \text{ WRT } c(\bar{u}),$$

where p is an *O-predicate* and c is a *C-predicate* or a constraint as in CLP and was given in [38]. The predicate p is said to be a **relaxation predicate** and c is said to be the **relaxation criterion**. Our semantics also captures the meaning of relaxation goals where c is any *O-predicate* that is not defined in terms of p . However, in this chapter, we will only deal with relaxation goals where c is a *C-predicate*. A relaxation goal may appear in the body of an *O-predicate*, a *D-predicate* or in a top level query.

As explained earlier, the above relaxation goal finds the best solutions to $p(\bar{t})$ that satisfy $c(\bar{u})$. We also provide a form of relaxation for *O-predicates* in which the relaxation is with respect to the underlying preference criterion for the predicate. Such a goal has the form

$$\text{RELAX}^* p(\bar{t})$$

This form of relaxation generates solutions (from the set of feasible solutions to the optimization problem) in the order of decreasing preference. This form of relaxation is motivated by the need for sub-optimization when the relaxation criterion is not known beforehand. This form of relaxation will permit backtracking into an optimization goal to get suboptimal solutions. In this case, p can be an *O-predicate* or a *D-predicate*.

In optimization problems, it is often the case that we are interested in not just the optimal solution(s), but the n best solutions. For example, in searching for flights from source a to destination b , we are not just interested in the cheapest flight but in say the first 5 cheapest flights so that we may pick one that best suits our frequent flyer program or stopover preference. To model such problems, we provide the construct

$$\text{RELAX}^n p(\bar{t})$$

Note that this goal will not provide n optimal solutions, but instead, it will first generate all the optimal solutions, then all the next-best solutions, and so on till the n^{th} level.

These constructs are motivated by their need in modeling optimization problems. We illustrate their usefulness via the examples in Section 6.2.2.

6.2.2 Motivation

In this section, we give examples to motivate the need for the $\text{RELAX}^* p(\bar{t})$, $\text{RELAX}^n p(\bar{t})$ and $\text{RELAX } p(\bar{t}) \text{ WRT } c(\bar{u})$ constructs in constrained objects.

We formulate the problem of putting together a meal plan given a person's dietary preferences. This problem illustrates different scenarios for relaxation depending upon the different forms of optimization.

Combining Multiple Optimizations

The meal class below, defines the optimal breakfast, lunch and dinner based on a person's taste.

```
class meal {
  attributes
    brkfst B; lunch L; dinner D;
    real Cals;
  constraints
    B.Cals + L.Cals + D.Cals = Cals;
  constructors meal(B1, L1, D1) {
    B = B1; L = L1; D = D1;
  }
}
```

The class dinner aggregates the bread, meat, vegetable and drink that the person prefers most (likes the taste of) for dinner. The constraint `din_bread(B, Bc)` ensures that B is a bread that the person likes (stated as preferences on `din_bread`) to have for dinner. The variable Bc represents the calorie content of B.

```
class dinner {
  attributes
    foodItem B, M, V, D;
    real Cals, Bc, Vc, Mc, Dc;
  constraints
    din_bread(B,Bc); din_meat(M,Mc);
    din_veg(V,Vc); din_drink(D,Dc);
    Cals = Bc + Mc + Vc + Dc;
  predicates
    din_bread(B,Bc) → bread(B,Bc).
    bread(french,10).
    bread(foccacia,20).
```

```

    :
preferences
  din_bread(B,_)  $\preceq$  din_bread(garlic,_) :- B  $\neq$  garlic.
  din_bread(french,_)  $\preceq$  din_bread(foccacia,_).
  :
constructors dinner(B1, M1, V1, D1) {
  B = B1; M = M1; V = V1; D = D1;
}
}

```

The first preference clause for the `din_bread` predicate indicates a preference for garlic bread over any other bread. The second preference clause indicates a preference for foccacia over french bread. We have only listed two preferences, though in general there may be more. The rest of the predicates `din_meat`, `din_veg` and `din_drink` are defined similarly. The classes `brkfst` and `lunch` are defined along the lines of the `dinner` class.

The predicate `din_bread` above is a basic *O-predicate* since it is defined in terms of *C-predicates*. Note that the class `meal` when translated to a PLP predicate, will not be an optimization predicate itself (it does not have a preference clause), but is defined using a combination of multiple *O-predicates*. Now, it may happen that given the preferences of the person, the optimal lunch and dinner may end up looking identical. So instead of simply solving the query `meal([_], [B,L,D,Cals])`, a more appropriate query would be

```
?- RELAX meal([_], [B,L,D,Cals]) WRT L  $\neq$  D
```

in which the preferences on the food items in either the lunch or dinner will be relaxed in case the optimal dinner and lunch are identical. Note that since a meal aggregates a breakfast, a lunch, and a dinner, and each of these consist of atleast one food item, the calorie content of a meal will always be non-zero (except for the unlikely case when all the selected food items are of zero calories).

Hierarchic Optimization

Suppose that in addition to the basic preferences in a meal defined above, the person has to adhere to certain other restriction, e.g., cutting down on calories. The class `low_cal_meal` below defines the person's preferred meal having lowest calories. The constraints in the

`low_cal_meal` class ensure that the total calories in a day's meal meet the standards recommended by the US RDA (Recommended Daily Allowances) for an average healthy person. The preference clause in `low_cal_meal` states that the lowest calorie meal that meets the person's taste is the optimal answer.

```
class low_cal_meal extends meal {
  constraints
    Cals <= 2500;
    Cals >= 1200;
  preferences
    min Cals;
  constructors low_cal_meal(C) {
    Cals = C;
  }
}
```

For simplicity, we have not modeled other nutrient requirements (vitamins, minerals, etc.) and assume that each item is served in 1 serving size. It is possible to extend the above Cob model to accommodate such details. Note that class `low_cal_meal` performs *hierarchical optimization*: its PLP-translation is an optimization predicate defined in terms of other optimization predicates. The above example brings up a number of interesting issues regarding relaxation.

Suppose the person's optimal breakfast, lunch and dinner do not meet the RDA requirements (of say the maximum calorie content). Surely, we do not want the goal `low_cal_meal(M, C)` to fail. Instead, we would like the optimization of either the breakfast, lunch or dinner (or possibly all three) to be relaxed so that the calorie requirements are met. In order to carry out such a relaxation, we make use of the `RELAX*` construct. The goal

```
?- RELAX* low_cal_meal([_], [B,L,D,Cals]).
```

will recursively relax one or more of the `brkfst`, `lunch`, `dinner` predicates (PLP-translations of the corresponding classes). The solution might involve an optimal breakfast combined with a suboptimal lunch and dinner.

The classes `low_sod_din` and `high_iron_din` shown below, optimize a dinner based on its nutritional content. The predicate `low_sod_din` returns only those optimal (based on the person's taste) dinners that have the least sodium content and the predicate `high_iron_din` returns the optimal dinner that has the highest iron content.

```

class low_sod_din extends dinner {
  attributes
    real Na, BNa, MNa, VNa, DNa;
  constraints
    Na = BNa + MNa + VNa + DNa;
    sodium(B,BNa); sodium(M,MNa);
    sodium(V,VNa); sodium(D,DNa);
  preferences
    min Na;
  constructors low_sod_din(Na1) {
    Na = Na1;
  }
}

class high_iron_din extends dinner {
  attributes
    real Fe, BFe, MFe, VFe, DFe;
  constraints
    Fe = B.Fe + M.Fe + V.Fe + D.Fe;
    iron(B,BFe); iron(M,MFe); iron(V,VFe); iron(D,DFe);
  preferences
    max Fe;
  constructors high_iron_din(Fe1) {
    Fe = Fe1;
  }
}

```

Suppose the person has nutritional requirements of low sodium and high iron. In that case, we would like to solve

```
?- low_sod_din([_], D), high_iron_din([_], D).
```

This is another form of *combining multiple optimizations*. It is quite possible that none of the optimal solutions to the first goal is an optimal solution to the second goal. Therefore, we would once again like to make use of relaxation. Either one or both of the optimization subgoals needs to be relaxed and so the query we are interested in is

```
?- RELAX* (low_sod_din([_], D), high_iron_din([_], D)).
```

Recursive Optimization

We now give an example of a PLP program which illustrates the issues involved in relaxation of recursive *O-predicates* and the need for recursive relaxation. Consider the dynamic

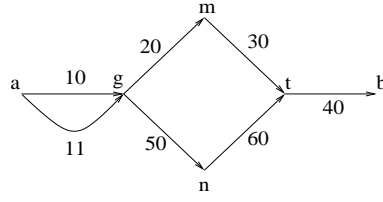


Figure 6.2: Sample Graph.

programming formulation of the shortest distance problem [40]. The predicate `sh_dist` is defined recursively as follows.

```

sh_dist(X,X,N,0).
sh_dist(X,Y,1,C) → X <> Y | edge(X,Y,C).
sh_dist(X,Y,N+1,C1+C2) → N > 1, X <> Y |
    sh_dist(X,Z,1,C1), sh_dist(Z,Y,N,C2).
sh_dist(X,Y,N,C1) ≤ sh_dist(X,Y,N,C2) ← C2 < C1.

```

Consider the sample graph shown in Figure 6.2. Suppose that the shortest distance between nodes `a` and `b` is `d`. Now consider the goal

```
?- RELAX sh_dist(a,b,N,C) WRT C > d.
```

We expect that the answers to the above goal give us the suboptimal shortest distance between `a` and `b`. Consider the graph in Figure 6.2.

The result of the goal `sh_dist(a,b,4,C)` is `C = 100`. Now suppose we want to get the next shortest distance. We evaluate the goal

```
?- RELAX sh_dist(a,b,4,C) WRT C>100.
```

Although this gives us the next best solution to the optimization goal, the above goal is a roundabout way to achieve relaxation of the underlying preference criterion: we first obtain the optimal answer and then use it to sub-optimize. In general it is more useful to have a builtin construct that will automatically relax an optimization goal with respect to the underlying preference criterion. The `RELAX*` construct is provided for this purpose and can be used directly to obtain relaxation with respect to different criterion. For example, we may want to sub-optimize the shortest distance goal so that a particular node is (or is not) visited. With suitable modifications to the `sh_dist` predicate to include the path corresponding to a distance, such a relaxation goal can be simply stated as

`RELAX* sh_dist(a,b,4,C,P), member(n, P).`

where the argument P is the path corresponding to the shortest distance. The relaxation goal returns the distances between nodes a and b in increasing order of cost and the first of these distances whose path does not go through n is the answer (as expected). We would like to note here that the above example suggests a general scheme for the operational semantics of relaxation goals involving basic, hierachic as well as recursive optimization. We define this scheme in the next section.

6.3 Operational Semantics

Having given several examples motivating the need for different forms relaxation goals in constrained objects and in Preference Logic Programs, we assume that the reader is now familiar with the different forms of relaxation goals and their expected answers. Assuming this informal description of the meaning (declarative semantics) of relaxation goals through examples, we now describe their abstract operational semantics. We first show why the approach to operational semantics of relaxation goals given in [38] cannot be applied to all the different forms of optimization. Then we describe our approach which first gives the operational semantics for $RELAX^*$ goals and subsequently defines the operational semantics of general relaxation goals.

6.3.1 Naive Approach

It would appear that the operational semantics of a relaxation goal such as $RELAX\ p(\bar{t}_1)$ WRT $c(\bar{u})$ can be defined simply by embedding the relaxation criterion into the definition of the *O-predicate*. This approach was used in [38]. We show below how such an approach while useful for relaxing basic *O-predicates*, cannot satisfactorily compute the answers to relaxation goals for hierarchic and recursive *O-predicates*.

Basic O-predicates. Basic *O-predicates* are non-recursive and non-hierarchic, i.e., they are defined in terms of *C-predicates* only. Suppose $p(\bar{t})$ is an *O-predicate* defined in terms of one or more *C-predicates*. The predicate `sh_path` defined in the previous section is an

example of such a p . Suppose p is defined as follows, where q_i are C -predicates.

$$\begin{aligned} p(\bar{t}) &\rightarrow q_1(\bar{u}), \dots, q_m(\bar{u}). \\ p(\bar{x}) &\preceq p(\bar{y}) \leftarrow d(\bar{x}, \bar{y}). \end{aligned}$$

To evaluate the goal RELAX $p(\bar{t}_1)$ WRT $c(\bar{u})$, we first define a predicate R_p as

$$\begin{aligned} R_p(\bar{t}_1) &\rightarrow q_1(\bar{u}), \dots, q_m(\bar{u}), c(\bar{u}). \\ R_p(\bar{t}) &\rightarrow \bar{t} \neq \bar{t}_1 \mid p(\bar{t}). \\ R_p(\bar{x}) &\preceq R_p(\bar{y}) \leftarrow d(\bar{x}, \bar{y}). \end{aligned}$$

where \bar{u} is obtained by applying the unifier of \bar{t}_1 and \bar{t} on \bar{u} . The second clause is given so that the definition of R_p is complete. Now the goal RELAX $p(\bar{t}_1)$ WRT $c(\bar{u})$, is evaluated as $R_p(\bar{t}_1)$. In the above transformation, the original O -predicate is transformed into another O -predicate by embedding the condition $c(\bar{u})$ into its body while leaving the preference criterion unchanged. For example, to evaluate

$$?- \text{RELAX } \text{sh_path}(a, b, C, P) \text{ WRT } C > d.$$

we first transform the definition of `sh_path` to define another predicate `R_sh_path` by embedding the condition $C > d$ into the body of `sh_path` but leave the preference condition on `sh_path` unchanged.

$$\begin{aligned} \text{R_sh_path}(a, b, C, P) &\rightarrow \text{path}(a, b, C, P), C > d. \\ \text{R_sh_path}(X, Y, C, P) &\rightarrow (X, Y) \neq (a, b) \mid \text{sh_path}(X, Y, C, P). \\ \text{R_sh_path}(X, Y, C_1, P_1) &\preceq \text{R_sh_path}(X, Y, C_2, P_2) \leftarrow C_2 < C_1. \end{aligned}$$

By pushing the condition $C > d$ into the body, we are able to reject paths with cost more than d before the preference clause orders them.

Hierarchic and Recursive O-predicates. The above scheme of embedding the relaxation condition $c(\bar{u})$ into the body of the definition [38] will not work if the O -predicate is hierarchic or recursive. For example, suppose the goal `sh_dist(a, b, N, C)` (see previous section) is to be relaxed with respect to the condition $C > d$ where d is the cost of the optimal path. Now suppose there is a single optimal path of cost d . If we were to embed the condition $C > d$ in the the body of the definition of `sh_dist`, we would not get any solution at all, since the shortest path will be rejected and no other path found.

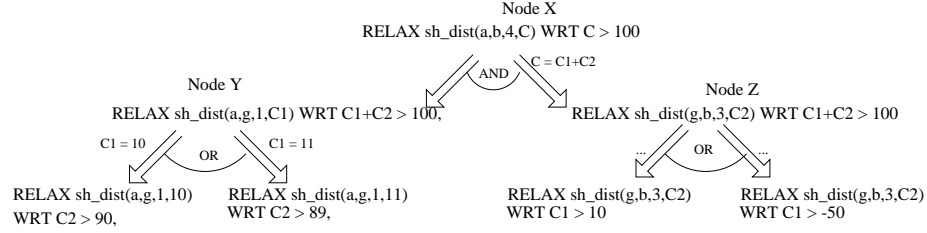


Figure 6.3: Naive evaluation of a relaxation goal.

Thus, to achieve relaxation of hierarchic or recursive predicates with respect to a condition, we must induce relaxation on the subgoals. At least one of the recursive calls in its body must be relaxed. Now suppose we recursively relax subgoals with respect to the same condition. The evaluation then proceeds as shown in Figure 6.3. The evaluation cannot proceed beyond any of the leaf nodes shown in Figure 6.3. This is because in each of the sub goals the relaxation constraint (criterion) has a free variable and the optimization goal is not sufficiently non-ground. For example, in the goal:

RELAX sh_dist(a,g,1,10) WRT C2 > 90.

the variable $C2$ does not appear in the optimization goal (which is being relaxed) and thus constraint $C2 > 90$ cannot be verified. If we ignore the constraint and optimize based on the cost, we will in fact get an incorrect answer. To see this, suppose the goal $\text{sh_dist}(a,g,1,10)$ with $C2 > 90$ is returned as the optimal choice at node Y of Figure 6.3. This will cause the optimal choice at Node Z to be the path $g \rightarrow n \rightarrow t \rightarrow b$ with $C2 = 150$. Thus the answer will be the path $a \rightarrow g \rightarrow n \rightarrow t \rightarrow b$ with cost 160 which is indeed greater than 100. However, this is not the shortest path with cost greater than 100. The correct answer is the path $a \rightarrow g \rightarrow m \rightarrow t \rightarrow b$ with cost 101. The error in the above goal reduction occurs because of the presence of free variables in the relaxation criterion. Thus we see that the operational semantics of $\text{RELAX } p(\bar{t}) \text{ WRT } c(\bar{u})$ for a recursive predicate p , cannot be obtained by simply relaxing its subgoals.

Thus the scheme of relaxing subgoals with the same/modified condition may work in some cases, for example, relaxing the shortest path with respect to the condition that it does not go through a particular vertex. But it cannot be used in general to evaluate relaxation of hierarchical *O-predicates*. Still, there is a well defined notion of relaxation of the subgoals,

viz., with respect to their underlying preference ordering. This motivates the need for a different approach to the operational semantics of relaxation goals that is based on the relaxation with respect to the underlying preference criterion. This approach first defines the operational semantics of $\text{RELAX}^* p(\bar{t})$ giving answers to $p(\bar{t})$ in decreasing order of the underlying preference as specified for $p(\bar{t})$. To obtain the answers to $\text{RELAX } p(\bar{t}) \text{ WRT } c(\bar{u})$, we first evaluate $\text{RELAX}^* p(\bar{t})$, followed by applying the condition $c(\bar{u})$.

6.3.2 Improved Approach

Intuition. We regard the problem of relaxation with respect to the underlying preference criterion as a specialization of the optimization problem in which the optimal solutions to the original problem have been removed from the set of feasible solutions. In other words, the constraints and the preference criteria remain the same but there are additional constraints that disqualify the optimal solutions. We first define the operational semantics of the $\text{RELAX}^* G$ construct and use it to define the semantics of the $\text{RELAX } p(\bar{t}) \text{ WRT } c(\bar{u})$ construct.

As stated earlier, the goal $\text{RELAX}^* G$ returns solutions in order of decreasing preference. It first returns the optimal answers to G , then the next-best answers, then the next to the next-best answers and so on. In this way, the goal $\text{RELAX}^* G$ exhaustively enumerates all the feasible solutions to the optimization goal G . We refer to the optimal solutions to G as $\text{RELAX}^0 G$, i.e., there is no relaxation. In other words, $\text{RELAX}^0 G \equiv G$. In general, the goal $\text{RELAX}^n G$ returns the n best solutions to the goal. We refer to n as the level of relaxation. Note that there may be more than one solutions at each level of relaxation just as there can be more than one optimal solutions to an optimization problem.

Since, $\text{RELAX}^0 G \equiv G$, the operational semantics of $\text{RELAX}^0 G$ are already well defined (see Section 6.1.3). For all $n > 0$, we define the operational semantics of $\text{RELAX}^n G$ in terms of the operational semantics of $\text{RELAX}^{n-1} G$. These operational semantics are in the form of rewrite rules and make use of the transitions described in Section 6.1.3, in scheme \mathcal{A} .

For $n = 1$, the operational semantics of $\text{RELAX}^1 G$ are defined as follows. Suppose the final state at the end of the derivation for $\text{RELAX}^{n-1} G \equiv G$ is $\langle S_0, P_0 \rangle$. We know that

this state is obtained by applying the transitions of \mathcal{A} to the start state $\langle \{ \langle G, \in, \in \rangle \}, \phi \rangle$. Therefore, the set $\{C \mid \langle L, C, \wp \rangle \in S_0\}$ represents the set of solutions to $\text{RELAX}^{n-1} G$, i.e., the optimal solutions to G . In order to get sub-optimal solutions, we must: remove those paths in S_0 from the PTSLD search tree that represent the optimal solutions to G ; unprune the paths that were pruned by the optimal solutions; and then continue the derivation. To do this, we introduce a transition rule that constructs a new state from the final state of the derivation for G .

$$\langle S_0, P_0 \rangle \rightarrow_{up} \langle S'_0, P'_0 \rangle$$

where $S'_0 = \{p \in P_0 \mid p \text{ was pruned by a path in } S_0\}$

and $P'_0 = P_0 - S'_0$

The derivation now proceeds from state $\langle S'_0, P'_0 \rangle$ using the transition rules of scheme \mathcal{A} . If the final state at the end of this derivation is $\langle S_1, P_1 \rangle$, then the answers to $\text{RELAX}^n G$, i.e., $\text{RELAX}^1 G$ are given by the set $\{C \mid \langle L, C, \wp \rangle \in S_1\}$.

We now define the transition \rightarrow_{up} more formally and the operational semantics of $\text{RELAX}^n G$ for any $n \geq 1$ as follows. Suppose that

$$\langle S, \mathcal{P} \rangle \equiv \langle S, \{ \langle S'_1, C'_1, \wp'_1, \ell_1 \rangle, \dots, \langle S'_m, C'_m, \wp'_m, \ell_m \rangle \} \rangle$$

is the final state at the end of the derivation for $\text{RELAX}^{n-1} G$. This means that the set

$$\{C \mid \langle L, C, \wp \rangle \in S\}$$

forms the $(n-1)^{th}$ sub-optimal answer constraints to G . From this final state, we obtain the next state by the application of the \rightarrow_{up} transition rule as follows

$$1. \langle S, \mathcal{P} \rangle \equiv \langle S, \{ \langle S'_1, C'_1, \wp'_1, \ell_1 \rangle, \dots, \langle S'_m, C'_m, \wp'_m, \ell_m \rangle \} \rangle \rightarrow_{up} \langle S', \mathcal{P}' \rangle$$

if the following holds. We first identify the following subset UP of pruned paths:

$$UP := \{ \langle S'_j, C'_j, \wp'_j, \ell_j \rangle \mid 1 \leq j \leq m, \$$

$$\forall k \in 1..m \text{ level}(\ell_k) < \text{level}(\ell_j), \$$

$$\forall k \in 1..m \text{ if } \ell_j \text{ and } \ell_k \text{ refer to the same predicate,} \$$

$$\text{then } \text{depth}(\ell_j) \leq \text{depth}(\ell_k) \}$$

where $\text{level}(\ell_j)$ and $\text{depth}(\ell_j)$ refer respectively to the level of the predicate that pruned the path $\langle S'_j, C'_j, \wp'_j \rangle$ and its depth in the PTSLD search tree. This information is already incorporated in the label ℓ_j (see Section 6.1.3).

The above set UP gives us the set of paths that should be unpruned, i.e., derivation down these path should be continued in order to find sub-optimal solution. The set S of successful paths at the end of the derivation for $RELAX^{n-1} G$ is now replaced by the set

$$NewS := \{ \langle L, C, \wp \rangle \mid \langle L, C, \wp, \ell \rangle \in UP \}$$

This set forms the new set of non-pruned, non-failed paths and the derivation proceeds with the state

$$\langle S', \mathcal{P}' \rangle \equiv \langle NewS, \mathcal{P} - NewS \rangle$$

while the set of successful paths that dominated (pruned) it is removed from the set of feasible solutions.

The selection of path that should be unpruned uses the level and depth of the predicate that pruned it. By selecting those paths whose label has the highest level, we ensure that an *O-predicate* is relaxed before relaxing the *O-predicates* in its definition. This also ensures that while relaxing an *O-predicate*, we explore the feasible solutions to its subgoals before further relaxing them. When two labels refer to the same predicate (and level), then we choose to unprune those with lower depth. This ensures that for recursively defined predicates, the outer most invocation is relaxed before the recursive call.

We then continue the derivation by applying the transitions of scheme \mathcal{A} to this new start state $\langle S', \mathcal{P}' \rangle$. If the final state at the end of this derivation is $\langle S_n, \mathcal{P}_n \rangle$, then the set

$$\{ C \mid \langle L, C, \wp \rangle \in S_n \}$$

forms the n^{th} sub-optimal answer constraints to G , thus giving us the solutions to $RELAX^n G$.

To summarize, relaxation is performed by first removing the paths corresponding to optimal solutions in the PTSLO search tree, then by systematically (with respect to level and depth) unpruning paths that were pruned by the optimal solutions and then continuing the search for optimal solutions.

Soundness and Completeness. In the previous sections we have given an informal definition of relaxation goals by illustrating their meaning through examples. A formal definition of the declarative semantics of relaxation goals of the form $RELAX \ p(\bar{t}) \text{ WRT } c(\bar{u})$

can be given in terms of the subset of the intended preferential consequences of the program that satisfy the constraint $c(\neg u)$. An approach along these lines has been discussed in [38]. The declarative semantics of general relaxation goals of the form $\text{RELAX } n$ and $\text{RELAX } *$ may require a recursive definition using the relaxed preferential consequences for $\text{RELAX } n-1$ to define those for $\text{RELAX } n$. However, for the purposes of this dissertation, we rely on the informal meaning of relaxation goals given via the examples in Section 6.2.2.

For an informal justification for the soundness of the operational semantics of Section 6.3 we rely on the soundness of the derivations for optimization goals [38]. Using this, we argue that if the sub-optimal solutions to a goal are the “next-best” solutions, i.e., they are considered as the best solutions in the absence of the optimal solutions, then removing the optimal solutions from the search tree, and again optimizing with respect to the same preference criteria will indeed lead us to these next-best solutions. A formal proof for soundness must use formal declarative semantics of relaxation goals and show that the unpruning step is sound for basic, hierarchic as well as recursively defined *O-predicates*. Such a proof lies beyond the scope of this dissertation.

The completeness result for the above operational semantics requires finite-depth recursion. As with the completeness of PTS LD derivations, the search tree must be finite, since all possible paths are explored before determining the optimal or suboptimal solutions. The proof for the completeness result relies on the completeness of PTS LD resolution which in turn relies on the completeness of SLD resolutions and is beyond the scope of this dissertation.

In addition to the above informal arguments for soundness and completeness, we have also successfully implemented the above scheme of operational semantics for relaxation goals for certain test programs including the `sh_dist` program. This implementation uses the technique of memoization or tabling [96, 93] to reduce the time for computing repeated calls to the same goal. In this technique, when a goal is evaluated for the first time, the system stores the computed answer(s) in a table (memo-table). All subsequent evaluations of the same goal are performed by simply retrieving its answer from this table, thus making the computation more efficient. In the case of optimization goals, we use this scheme to store not only the optimal but also the feasible solutions. The feasible solutions are useful

if the optimization goal is later subject to relaxation.

6.4 Related Work

Optimization in logic programming is discussed in [26, 82] and is given a semantics based on negation. Unlike these approaches where optimization can be stated only via a minimizing or maximizing objective function, PLP provides a more general framework for optimization and relaxation and gives the programmer explicit control over stating the preference criterion. Preferences have also been used in databases [19] and the reference [29] gives a survey of different forms of preference structures and their numerical representations.

The PLP paradigm crucially depends on memoization for efficient computation as well as termination of computation. Memoization or tabling has been used with logic programming and deductive databases for efficiency and termination [101, 96, 92, 111, 93]. Tabling is typically used to store answers to a goal for retrieval in case the same goal is evaluated again. In the PLP context, the memo-table stores only the optimal solutions to an optimization goal. But in the operational semantics of relaxation goals that we described in the previous section, the memo-table stores the optimal as well feasible solutions to an optimization goal. This reduces computation time when such a goal is subjected to relaxation.

Some relevant contributions in the area of preferences and relaxation within the framework of logic programming include Courteous Logic Programs [42], Relaxable Horn Clauses (RHC) [13, 80] and Hierarchical Constraint Logic Programming (HCLP) [11, 113]. Courteous logic programs are an extension of logic programs with a *Overrides* predicate that states a pairwise preference order between the clauses of a predicate definition. Program rules can contain negated literals in the head as well as the body and hence the logical consequence of a query may contain contradictions. Stating a priority between pairs of rules allows the resolution of such contradictions or conflicts and leads to a unique set of logical inference of a program that does not have contradictions. Preferences in PLP on the other hand, do not address the problem of conflict handling (since the head of a rule cannot be a

negated literal). They are used instead, to order the feasible solutions to a goal (or the set of logical consequences of a programs) and generate them in decreasing order of preference using the relaxation constructs. Although the *Overrides* predicate of courteous logic programs, in the absence of conflicts, may be viewed as a means for ordering or prioritizing the solutions to a goal, the priority cannot be relaxed. RHC defines a relaxation clause as a definite clause with a partial order over the goals in the body. In case of a failure, the partial order determines the order in which the goals should be relaxed. This form of relaxation can be thought of as providing an order on the solutions to a goal. It cannot be used to state or relax the underlying preference criterion of an optimization problems.

Hierarchical constraint logic programming (HCLP) is an extension of CLP in which constraints are organized into a hierarchy by the tags *required*, *strong*, *weak*, etc. to indicate that certain constraints must be satisfied and those that can be relaxed. Each of the tags is associated with a weight and an error function that determines how well a valuation satisfies the constraints of the problem. A *comparator* function is then used to compare two valuations based on their error values. Thus the HCLP scheme is parameterized by the domain of constraints (similar to CLP) and also by the choice of error function and comparator. Although the comparators can be thought of as stating a preference between multiple solutions to the required constraints, not all forms of optimization problems (in particular global optimization such as shortest distance) can be stated in the HCLP paradigm [38]. The PLP paradigm along with the relaxation constructs introduced in this chapter is a more expressive paradigm for constraint optimization and relaxation.

Chapter 7

Case Studies

In this chapter we present three case studies illustrating the application of the Cob programming language and environment to engineering modeling as well as non-engineering domain. The case studies required an understanding of the problem domain to elicit details of the structure being modeled and to design an appropriate class hierarchy for the problem. In all the problems presented in this chapter, the Cob model can be developed textually or through the domain dependent and independent interfaces. After specifying the Cob model, we illustrate its use for analyzing various types of scenarios. We also compare our model with other existing approaches to modeling.

The first case study in Section 7.1 is from the electrical engineering domain and illustrates the simplicity and power of the constrained object paradigm. We define a Cob model for electrical circuits and present a domain specific visual interface for drawing electrical circuits (AC as well as DC). We show how the underlying Cob model of the diagram along with the Cob debugging tool provides a powerful and flexible computational model for simulating electrical circuits and for pedagogic purposes. The second case study in Section 7.2 is from the mechanical engineering domain from the area of variant product design. We design a gear train based on the specifications and preferences of the client. In coming up with a model, we show the use of the CUMML tool for drawing class diagrams. The third case study in Section 7.3 discusses the use of constrained objects to model documents from the simple spreadsheets to books. In particular we give a Cob model for the problem of formatting the contents of a book.

7.1 Electric Circuits

The classical problem of modeling electrical circuits illustrates the simplicity and power of the Cob programming language and modeling environment. The behavior of the components of a circuit varies depending upon the voltage supply (AC or DC). In Section 3.2.4 we defined Cob classes for modeling DC circuits. In this case study we use the Cob language and environment to model AC as well as DC circuits.

We would like to model resistance, inductance, and capacitance circuits (RLC circuits) as constrained objects. The components of such circuits include resistors, capacitors, inductors, diodes, transformers, wires, batteries, and AC voltage sources. We would like to be able to model any assembly of such components and use this model to: (i) compute the current or voltage drop at any point in the circuit; (ii) specify voltage across two points and compute what the input voltage should be; (iii) specify a model giving values for all the attributes of the circuit and verify the model; (iv) debug an over-constrained model; (v) specify the model through a visual drawing tool and observe the results of the underlying Cob computation through this tool; (vi) use the drawing tool to edit, modify and resolve the model any number of times.

7.1.1 Cob Model of RLC Circuits

We model the components and connections of RLC circuits as objects and their properties and relations as constraints over the attributes of these objects. Given below is the code that defines these classes. The features or attributes of analog circuits (current, voltage, etc.) are represented as complex numbers. We first define a class `complex` which represents the set of complex numbers and the predicates of this class represent some basic mathematical operations on complex numbers.

```
class complex {
  attributes
    real Re,Im;
  predicates
    c_add([R1r, R1i], [R2r, R2i], [R3r, R3i]):-
      R3r = R1r + R2r,
      R3i = R1i + R2i.
    c_mult([R1r, R1i], [R2r, R2i], [R3r, R3i]):-
```

```

        R3r = (R1r*R2r) - (R1i*R2i),
        R3i = (R1r * R2i) + (R2r * R1i).
c_sub([R1r, R1i], [R2r, R2i], [R3r, R3i]):-
    R3r = R1r - R2r,
    R3i = R1i - R2i.
c_neg([R1r, R1i], [R2r, R2i]):-
    R2r = - R1r,
    R2i = - R1i.
makeReal([R1r, R1i], R):-
    R=(R1r*R1r+R1i*R1i).
constructors complex(Re1,Im1) {
    Re = Re1;
    Im = Im1;
}
}

```

The component class below models any electrical entity with two terminals. Its attributes $V1$, $V2$ are complex numbers and represent the voltages at the two terminals. Similarly the attributes $I1$, $I2$ represent the corresponding currents at the two terminals. The currents at the two terminals are equal and opposite to each other and the potential difference across the component (V) is the difference between $V1$ and $V2$. These two relations are represented as constraints of the component class. The class component is an abstract class and hence cannot be instantiated.

```

abstract class component {
    attributes
        complex V1, V2, I1, I2, V;
    constraints
        c_neg(I1, I2);
        c_sub(V1, V2, V);
}

```

A resistor is a component characterized by its resistance R . It is represented by the `resistor` class which is a subclass of the component class. The values of current, voltage and resistance of a resistor are governed by Ohm's law which is represented as a constraint of this class.

```

class resistor extends component{
    attributes
        complex R;
    constraints

```

```

        c_mult(I1, R, V);
    constructors resistor(R1) {
        R = new complex(R1,0);
    }
}

```

A diode is a component that allows current to flow only in one direction. The current flow is a piecewise linear function of the voltage across the diode and represents the transition from a state of reverse breakdown to reverse bias to forward bias. This relation between the voltage and current can be modeled either through predicates or as conditional constraints. We give the latter formulation below. The diode class is a subclass of component.

```

class diode extends component{
    attributes
        real Vr, Ir;
    constraints
        V = new complex(Vr, 0);
        I1 = new complex(Ir, 0);
        diodel(Vr,Ir);
    predicates
        diodel(V, I) :- V < -100, DV = V + 100, I = 10*DV.
        diodel(V, I) :- Vr >= -100, Vr < 0.6, I = 0.001*V.
        diodel(V, I) :- Vr >= 0.6, DV = V - 0.6, I = 100*DV.
    constructors diode(){
}

```

Electrical components such as voltage sources, inductors, and capacitors are two terminal components whose behavior depends upon the frequency of the applied AC voltage. The class of frequency dependent components is defined below as the `freq_component` class, a subclass of component. The attribute `W`, a function of the frequency, is known as omega and is used in defining the characteristic of frequency dependent components.

```

abstract class freq_component extends component{
    attribute
        real Freq; % frequency of voltage
        real W; % omega
    constraints
        Pi = 3.141;
        W = 2 * Pi * Freq;
}

```

The behavior of a voltage source is frequency dependent. If Omega (ω), the frequency, is zero, then it is interpreted to be a DC voltage source.

```
class voltage_source extends freq_component{
  constraints
    V2 = Zero :- W = 0;
  constructors voltage_source(Va, W1) {
    V1 = new complex(Va, 0);
    W = W1;
  }
}
```

A capacitor resists quick changes (high frequency) in voltage whereas an inductor resists quick changes in current. The resistance is represented by the capacitance and inductance of the capacitor and inductor respectively. While resistors vary only the magnitude of the voltage, capacitors and inductors introduce a phase lag in the AC voltage. Hence frequency dependent characteristics are modeled using complex numbers (resistors are represented by complex numbers with the imaginary part zero). The voltage and current of a capacitor (or inductor) are related to the capacitance (or inductance) and Omega by the constraints shown in the following classes.

```
class inductor extends freq_component{
  attributes
    complex L; % inductance
    complex T; % temporary variable
  constraints
    c_mult(T, I, V);
  constructors inductor(L1,W1){
    L = L1;
    W = W1;      % Omega
    T = new complex(0.0, W*L);
  }
}

class capacitor extends freq_component{
  attributes
    complex C; % capacitance
    complex T; % temporary variable
  constraints
    c_mult(T, V, I);
  constructors capacitor(C1,W1){
    C = C1;
  }
}
```

```

        W = W1;      % Omega
        T = new complex(0.0, W*C);
    }
}

```

The class `end` represents a particular end of a component. We use the convention that the voltage at end 1 of a component is `V1` (similarly for current). A node aggregates a collection of ends. When the ends of components are placed together at a node, their voltages must be equal and the sum of the currents through them must be zero (Kirchoff's law). Notice the use of the quantified constraints (`forall`) to specify these laws.

```

class end {
  attributes
    component C;
    real E, V, I;
  constraints
    V = C.V1 :- E = 1;
    V = C.V2 :- E = 2;
    I = C.I1 :- E = 1;
    I = C.I2 :- E = 2;
  constructors end(C1, E1)
    { C = C1; E = E1; }
}
class node {
  attributes
    end [] Ce;
    real V;
  constraints
    sum C in Ce: C.I = 0;
    forall C in Ce: C.V = V;
  constructors node(L) {
    Ce = L; }
}

```

7.1.2 Cob Diagrams of Electrical Circuits

We have developed a domain-specific tool for modeling DC circuits. The visual aspects of this tool were implemented by Abhilash Dev and Narayan Mennon, recent graduate students of the Department of Computer Science and Engineering, University at Buffalo. This modeling tool provides a palette of buttons for creating instances of resistors, batteries

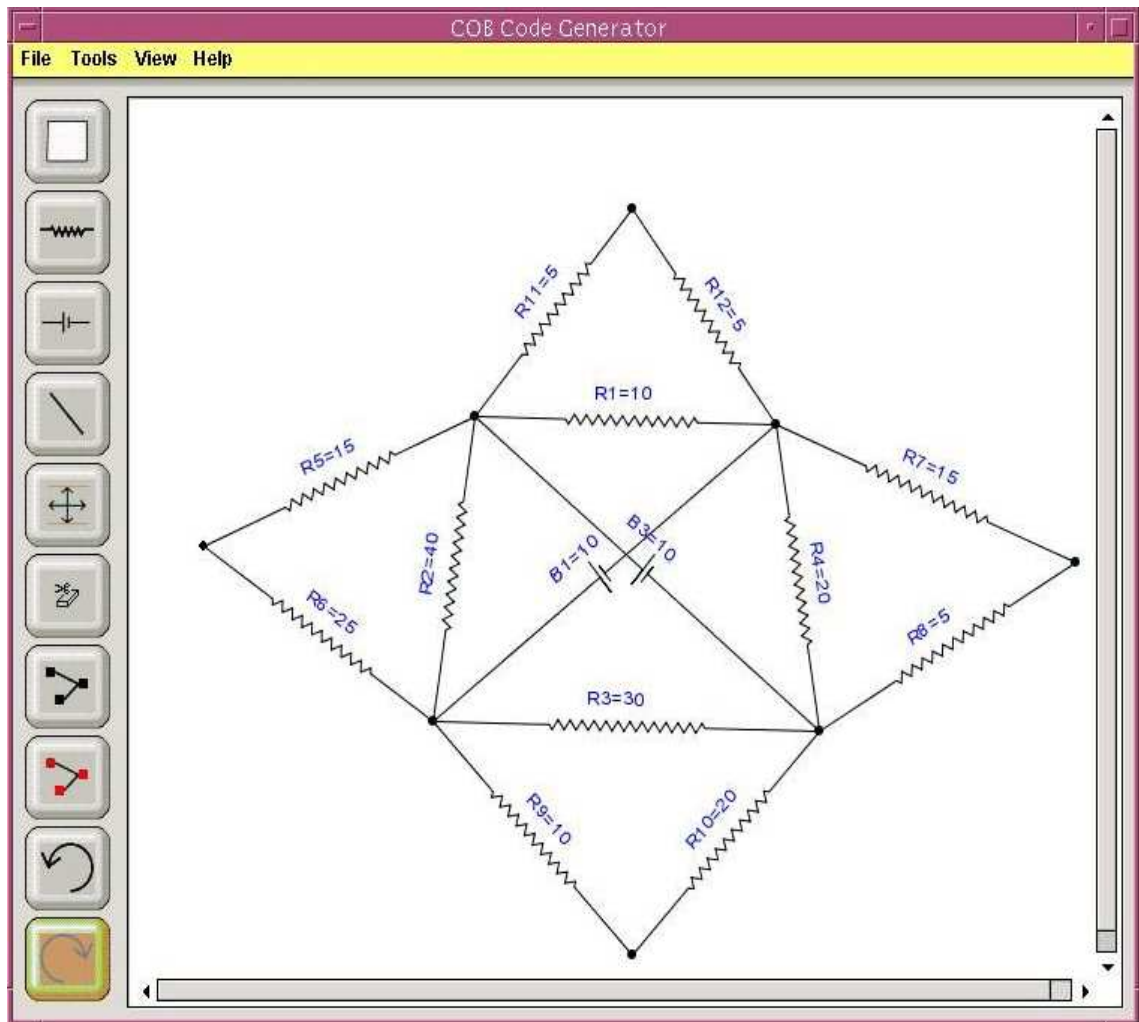


Figure 7.1: Snapshot of the Circuit Drawing Tool

and wires. Electrical joints can be created by merging the ends of two or more electrical components.

A similar domain-specific tool is provided for modeling RLC analog circuits. The visual aspects of this tool were developed by Palaniappan Sathappa and Prakash Rajamani, graduate students in the department of Computer Science and Engineering, University at Buffalo.

DC Circuits. Figure 7.1 shows a circuit drawn using the DC circuit drawing tool. The interface has a predefined library of classes corresponding to each component. Placing the icon of a component on the canvas creates an instance of the corresponding class. To create

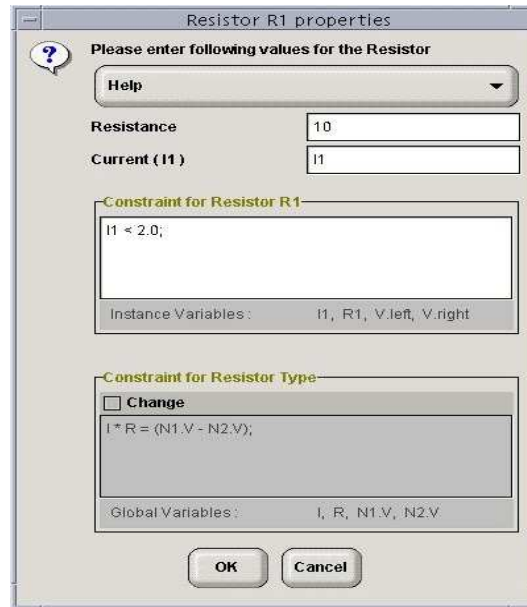


Figure 7.2: Snapshot of the input window for a resistor in the circuit drawing tool

an instance of a resistor, the user clicks on the resistor icon. This pops open a window (Figure 7.2) through which the user can enter values for the attributes of a resistor (e.g. its resistance) and any instance level constraints on the attributes (e.g. current must be less than 10Amps). Any or all attributes may be left uninstantiated or given a variable value (e.g. X). A similar interface is provided for creating an instance of a battery. Every instance of a component is labeled with a default name.

Once the drawing is completed, the user can compile the drawing to generate the textual Cob code corresponding to the diagram. This is done by clicking on the **Compile** button from the **Tools** menu. By clicking on the **Run** button from the **Tools** menu, the generated Cob code is compiled and executed. The answers resulting from this execution are displayed on the diagram. The user can get different views of the structure: the diagram; the Cob code; the translated CLP(R) code; and the window showing the script of the execution. The modeler can click on any component of the diagram and get information about the instance, its input constraints and output constraints as shown in Figure 7.3.

RLC Circuits. The interface for drawing analog circuits works much the same way and has extra icons for drawing inductors, capacitors and AC voltage sources. Figure 7.4 shows

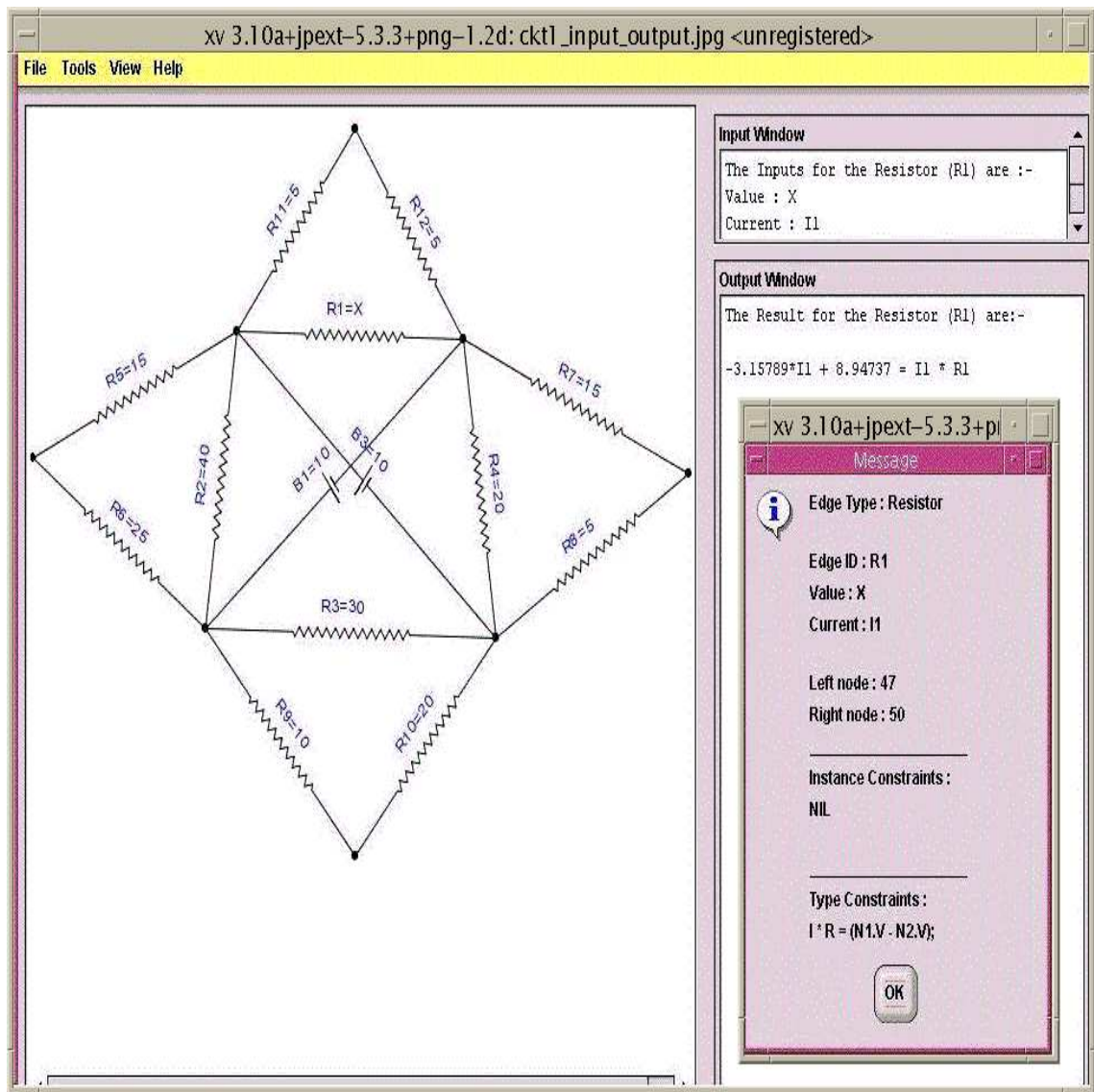


Figure 7.3: Snapshot of the Circuit Drawing Tool showing input and output windows and details of resistor

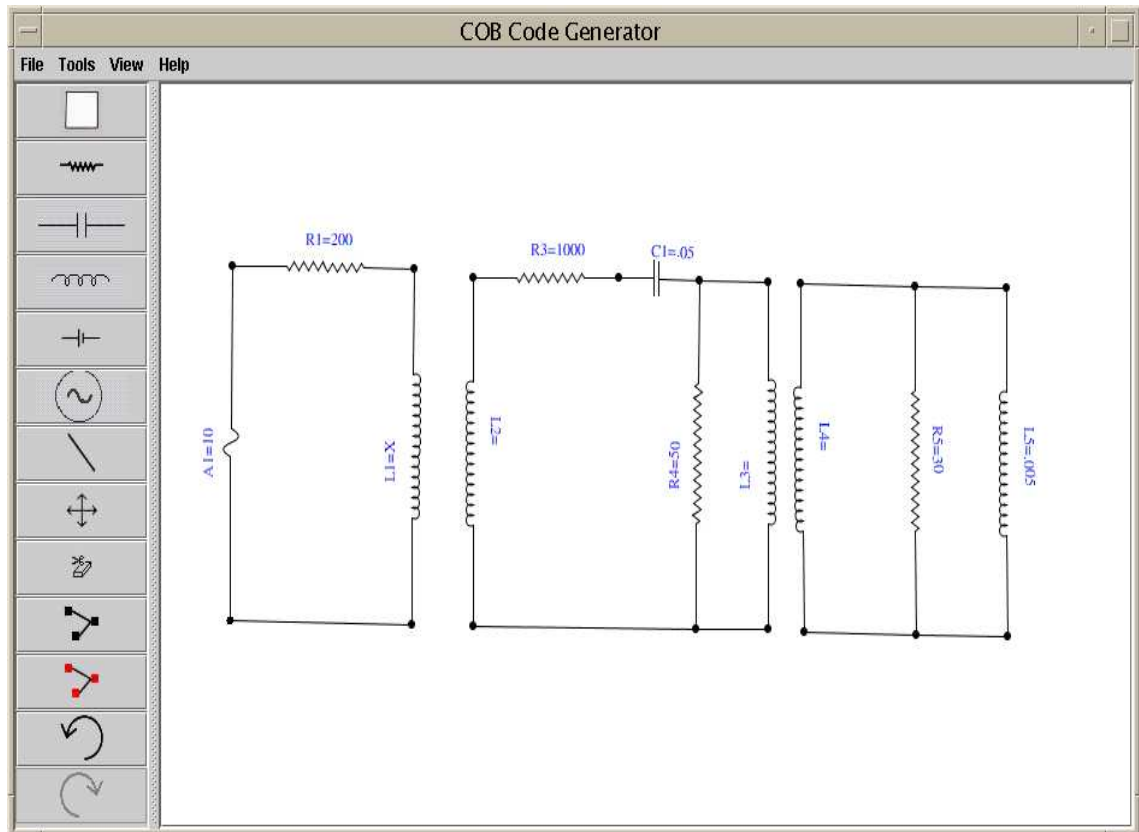


Figure 7.4: Snapshot of the Analog Circuit Drawing Tool

a snapshot of this tool.

7.1.3 Uses of Cob Model

Using the Cob classes of Section 7.1.2, we can model any analog circuit containing any number of the above components. Given initial values for some attributes such models can be used to calculate values of the remaining attributes (e.g. the current through a particular component or voltage drop at a particular node). The Cob model may be used to solve different kinds of problems. The designer may model the circuit leaving some of the attributes unspecified. The system can then be asked to run the model, i.e., solve the constraints and arrive at values or constraints for all the unspecified fields. After a solution has been obtained, the designer may modify the value of some of the attributes to see the effect of the modification on the rest of the model. Instead of giving initial values

to a component, a modeler may specify the constraints on the value of a component (e.g., the current through a component must be between 5Amps to 10Amps. The Cob model will then returns answers for the rest of the components in the form constraints on their values. The debugging tool of Section 5.5 can be used to locate the source of error in an over-constrained model.

Conclusions and Future Work. The Cob code presented in this case study provides a simple and intuitive model for electrical circuits. The CLP(R) models of such circuits are given in [45]. We have shown a part of such CLP(R) code in Section 2.2.2. In comparison to the CLP(R) code, the above Cob model is more intuitive, concise, readable and amenable to modifications. When modeling, thinking in terms of objects leads to better design. Directions for future work include combining the drawing tool with the debugging tool so that the source of error can be displayed on the diagram instead of the Cob code generated from the diagram.

7.2 Product Design: Gear Train

The process of **product design** begins with the specification of the functional requirements of the product. This specification is given by the client. Along with the functional requirements, the client also places constraints and preferences on the performance of the product. The domain mapping process of product design involves mapping the functional requirements to concepts in the physical world that realize them. The original function specified by the client is broken down into smaller functions and in the process, the corresponding concepts get refined. There can be more than one concept that can achieve the same function. The designer must build the product by choosing the appropriate concepts and assigning absolute values to their physical attributes. The design, i.e., choice of concepts and values of physical attributes, must be such that building such an object is feasible in the real world. The aim of this case study was to explore the use of **Cob**, in product design.

A **gear train** is a special kind of drive whose purpose is to transmit rotary power between shafts while reducing angular speed. It consists of an assembly of pairwise engaged gears (meshes) enclosed in a gear box. The efficiency of a drive depends on its input and

output torques, angular speeds and transmission powers. An acceptable design of the gear train must meet the functional specifications and constraints of the designer as well as preferences like maximizing its efficiency. The problem is to create a constrained object model for gear trains that can be used to come up with a design for the gear train, validate an existing design, etc. The above problem was proposed by Prof. Li Lin of the Department of Industrial Engineering, and by Prof. Bharat Jayaraman, Department of Computer Science and Engineering, University at Buffalo. The details of the Cob model were worked out with the help of Jason Chen, a former graduate student of the Department of Industrial Engineering, University at Buffalo.

7.2.1 Structure of Gear Train

Drive: A `drive` is an engineering entity that transmits rotary power between shafts while reducing angular speed. There are different types of drives based on the medium used for transmitting power between the shafts, e.g., `belt_drive`, `gear_train`, `chain_drive`. The attributes describing a drive are its input(T_{in}) and output(T_{out}) torque (turning forces on its input and output shaft), input (H_{in}) and output(H_{out}) transmission powers, input(N_{in}) and output(N_{out}) angular speeds of the shafts, efficiency(E) and speed reduction ratio(R). These attributes are related by the following equations:

$$E = H_{out} / H_{in}.$$

$$R = N_{out} / N_{in}.$$

$$T_{in} = 63025 * H_{in} / N_{in}.$$

$$T_{out} = 63025 * H_{out} / N_{out}$$

Gear Train: A `gear_train` is a drive that uses gears to transmit rotary power between shafts. A pair of gears engaged together form a mesh. A gear train consists of an assembly of meshes enclosed in a gear box. The speed reduction ratio(R) of the gear train is related to the speed reduction ratio(r) of the meshes by the relation:

$$R = \prod (M \in Mesh) M.R$$

There are different types of gear trains based on the shape of the teeth of the gears, the angle between the shafts and the number of gears inside a gear box.

Gear: The attributes of a gear include its diametral pitch(P), number of teeth(N), pitch

diameter(D), face width(F), pressure angle(PA), size(S). The following relations hold on these attributes

$$P = N/D.$$

$$N \geq 18.$$

$$PA \in \{14.5, 20, 25\}.$$

$$9 / P \leq F \leq 13 / P$$

Mesh: For placing two gears say G1 and G2 together in a mesh, their diametral pitch, face width and pressure angles have to be equal. Therefore the following relations must hold.

$$G1.P = G2.P.$$

$$G1.F = G2.F.$$

$$G1.PA = G2.PA.$$

The speed reduction ratio of a mesh is related to the number of teeth of its gears by the following relation:

$$R = G2.N / G1.N.$$

Two adjacent meshes share a shaft, hence the output angular speed of one mesh is the input angular speed of the next. Hence,

$$\forall M \in \text{Meshes} : M.G2.S = \text{next}(M).G1.S$$

Depending upon the type of gear used inside a mesh, the engineer may be able to provide more information specific to the gear used. For example, if it is known in advance that spur gears will be used in the design, then the following constraints can be added to the properties of a mesh: $R \leq 5$ and $G1.N \geq 17$.

Gear Box: A gear box is the physical box around an assembly of meshes that encloses the meshes but leaves part of the input and output shafts of the first and last mesh outside. The attributes of a gear box are its length, width and height and the number of gears (N_g), bearings(N_b) and shafts(N_s) inside it. Since a mesh has two gears and there are two gears on one shaft, the following relation holds:

$$N_b = 2 * N_s$$

The problem of designing a gear train that meets client specification, involves coming up with values for the physical attributes of a gear train. These values have to be such that the performance and certain other attributes meet the client's constraints. There are two

stages in the design. In stage I, given values for T_{in} , H_{in} , N_{in} , calculate the # of meshes and # of teeth required to maximize the efficiency of the drive. In stage II, given the # of meshes and # of teeth from stage I, the values of P , D and F need to be calculated.

7.2.2 Cob Model of Gear Train

We now give a constrained object representation of a gear train in **Cob** and describe how the computational engine of Cob can be used to come up with a specific instance of a gear train that meets the client's constraints and preferences.

The structure of a gear train described above gives an intuitive class hierarchy for defining Cob classes for a gear train. We begin with the `drive` class that represents any generic drive. The attributes and constraints on these attributes are derived from the properties of a drive. In addition to the relations among attributes mentioned above, the client has specified that gear train be built with the maximum efficiency possible. A drive is given as a relation between its input and output torques, angular speeds and transmission powers. We model this as a constraint in the `drive` class. The designer can specify a preference that the transmission power be as close to 12hp (horse power) as possible. The problem is to maximize the efficiency which is also stated as a preference. For simplicity we do not show the constructors in the classes below. We give a part of a constrained object hierarchy that can be used to models different gears. Figure 7.5 shows the entire class diagram.

```
class drive {
  attributes
    real  $T_{in}$ ,  $T_{out}$ ,  $N_{in}$ ,  $N_{out}$ ,  $H_{in}$ ,  $H_{out}$ ,  $E$ ,  $R$ ;
  constraints
     $E < 1$ ;
     $E = H_{out} / H_{in}$ ;
     $R = N_{out} / N_{in}$ ;
     $T_{in} = 63025 * H_{in}/N_{in}$ ;
     $T_{out} = 63025 * H_{out}/N_{out}$ ;
  preferences
    max  $E$ .
}
class gear_train extends drive{
  attributes
    mesh [] Meshs;
    gear_box Gbox;
```

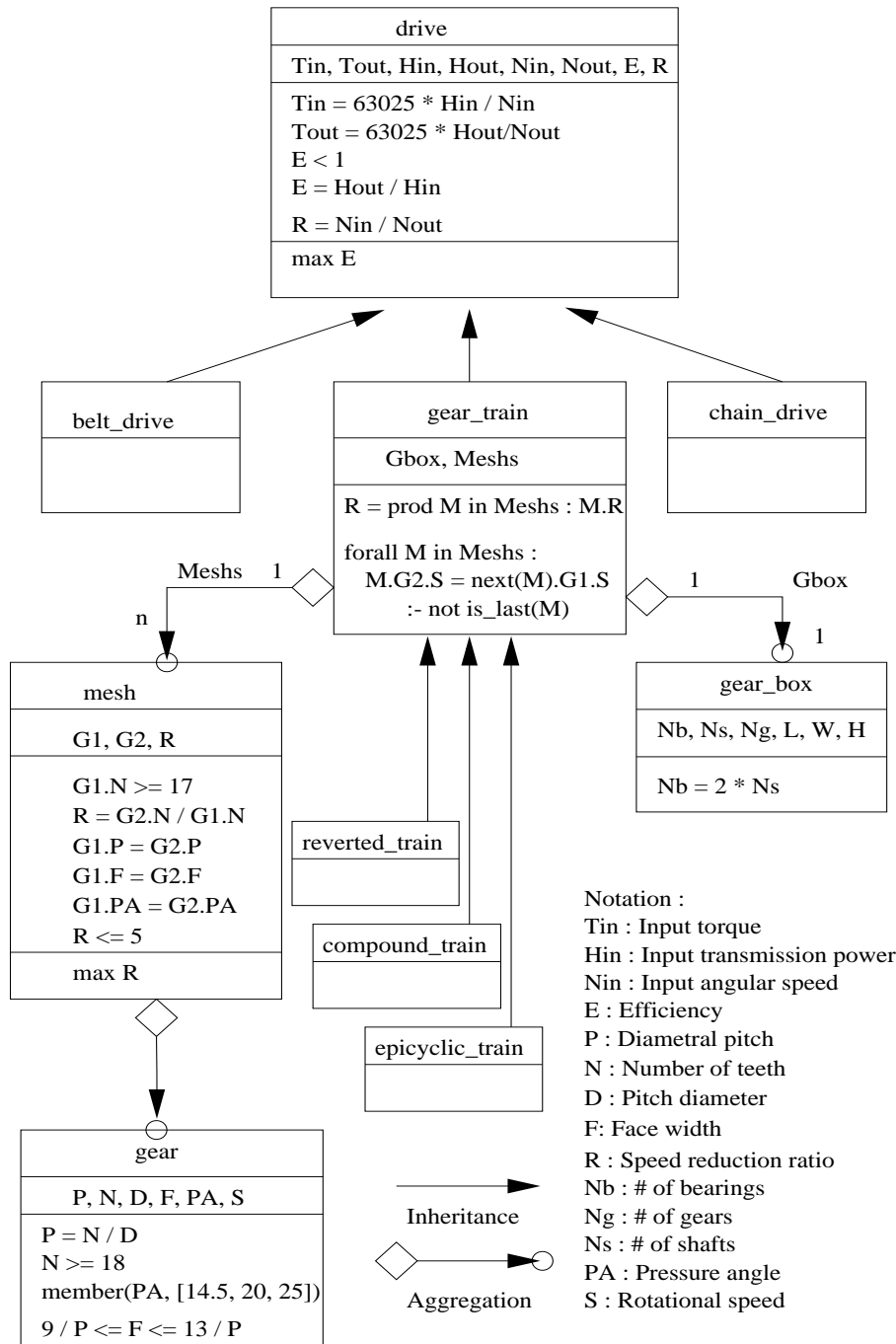



Figure 7.5: Gear Train as Constrained Objects

```

constraints
    size(Meshs, NumMesh);
    R = prod M in Meshs: M.R;
    forall M in 1..(NumMesh-1): M.G2.S = next(M).G1.S ;
}

class mesh {
    attributes
        gear G1, G2;
        real R;
    constraints
        G1.N >= 17;
        R = G2.N / G1.N;
        G1.P = G2.P;
        G1.F = G2.F;
        G1.PA = G2.PA;
        R <= 5;
    preferences
        max R.
}

class gear {
    attributes
        real P, N, D, F, PA, S;
    constraints
        P = N/D;
        N >= 18;
        PA ∈ {14.5, 20, 25};
        9 / P <= F <= 13 / P;
}

class gearbox {
    attributes
        real Nb, Ns, Ng, L, W, H;
    constraints
        Nb = 2 * Ns;
}

```

7.2.3 Uses of Cob Model

The above Cob model of a gear train can be used by the designer in different ways.

1. *Design*: Given values of some attributes, the Cob model can be used to calculate values of remaining attributes. For example, given the values for T_{in} (or H_{in}), N_{in} , R , the Cob model can be used to compute the value of N_{out} .

2. *Verification*: A designer can specify an existing design in Cob and the underlying Cob computational engine can verify whether design satisfies the constraints of every component of the gear train. The design that needs to be verified need not be complete (i.e., all the attribute values need not be specified). The designer can specify a partial model and the Cob engine can verify or solve this partial model returning answer constraints.
3. *Optimization*: Preferences such as maximum efficiency for the drive can be stated directly in `drive` class as `max E`. The constraint $E < 1$ (which prevents H_{out} from becoming equal to H_{in}) along with the constraints of the rest of the Cob classes forms a linear programming problem with the objective function maximizing E . The underlying Cob engine can solve such optimization problems to give the values for the attributes of the gear train.
4. *Interactive Design*: Very often some of the design decisions are based purely on the experience of the design engineer. For example, a partial design of a gear train may be developed based on the constraints and functional specifications of the problem but some choices such as the type of gear (spur, helical, or double helical gears) are made by the engineer. The Cob engine can be used to come up with a partial model of a gear train, the engineer can then make some design choices and the Cob engine can verify/solve the resulting model. Thus at any point in the design process, the engineer can modify the (partial) model and observe the effect of the change on the rest of the model.

Conclusions and Future Work. It is natural to model gear trains and engineering artifacts in general as an assembly of constrained objects:

- Each physical component of the engineering structure has a counterpart in the Cob model. The attributes of each physical object can be represented as the attributes of the corresponding Cob object. Thus the compositional nature of an actual gear train can be mapped onto the class hierarchy that represents it in Cob. The CUMIL tool provides a convenient way of thinking and designing with an object-oriented approach.

- Compared to a the traditional paradigm of imperative objects, a constrained object representation is better since the behavior of the structure can be modeled through the constraints, and the underlying Cob computation engine does the constraint handling. The relations that hold between the attributes as well as the restrictions on their values as given by client specifications can be declaratively stated as Cob constraints. The computational engine of Cob can be used to verify these constraints and/or use them to obtain values for some of the uninitialized attributes of the complex object.

In comparison with other object oriented languages with constraints, e.g. Modelica [88], the Cob model of a gear train is easier to understand, develop and debug. Future directions for work include developing a domain specific visual interface for drawing assemblies of gears and gear trains, investigating the use of constrained objects with predicates in variant product design, etc. When designing an engineering artifact, there may be several choices for a physical component of the structure. For example, there may be different brands of a part, differing in quality, color, cost, etc. The choice may depend upon the engineers experience as well as compatibility constraints between various parts. In particular, we have explored the use of Cob in representing *shape grammars* [2] and *bill of material* and feel that constrained objects with predicates may provide a powerful tool for specifying and solving such problems in variant product design.

7.3 Documents as Constrained Objects

The use of constrained objects is not restricted to modeling engineering structures alone. Any complex system that is compositional in nature and whose characteristics are governed by some rules can in general, be modeled using constrained objects. For example, constrained objects are ideal for modeling the class of structured documents such as spreadsheets, telephone bills, tax forms, transcripts, etc.

7.3.1 Structured Documents

By structured documents we mean documents whose contents are organized or arranged in a certain way conforming to some format and are related to each other via some constraints.

The contents of such documents can be modeled as objects and the relation between the contents can be stated as constraints. Constrained object models of such documents may be used for their analysis and recognition, and can be queried for information or consistency. We give some examples of structured documents below.

Document Analysis.

Student Transcript. As an example of a structured document, consider a student's transcript or grade sheet. This document is kind of a spreadsheet that contains a list of courses that the student has taken in each semester, the number of credits for each course and the grade received in each course. There are different kinds of constraints on the contents of such a grade sheet. For example, some courses must be taken for exactly 3 credits, some other can be for variable number of credits not more than 6 and so on. The overall grade point average (GPA) on the grade sheet is related to the grades in all the courses by a certain formula. Another constraint may limit the number of credits that have a non-letter grade. The contents of a transcript and the constraints on them can be modeled using constrained objects. In order to determine if the student meets the degree requirements, we can pose the various queries to the Cob model, e.g., has the student taken certain course and electives, is the total number of credits sufficient, is the grade at least a B+ and so on. A student can use such a model to determine the courses and the minimum grade in each such that he/she must take in order to graduate in a certain number of semesters.

Tax Forms. A tax form can also be thought of as a spreadsheet but the constraints are much more complicated. The numerical figures in some rows of a tax form depend on the contents of previous rows only by simple arithmetic expressions such as addition, subtraction, percentage, etc. But the numeric figure of other rows depend on the person's income and can be determined only after consulting some tax tables. Conditional constraints are very appropriate for modeling such dependencies. A constrained object model of a tax form will be a modular representation of its contents and will provide a declarative specification of the tax rules. The tax payer can simply provide his or her personal and financial information such as name, date of birth, income, number of dependents, etc. and the Cob model can compute the tax based on the tax rules. Although there are existing softwares that perform such tasks, an advantage of the Cob model is that it can handle queries that

contain constraints and/or ranges for variables. Such queries can be useful in determining the optimum set of claims, exemptions or deductions such that the tax is minimized.

Document Recognition. An interesting problem in document recognition consists of scanning an old document and filling in the faded parts or contents that could not be read by the scanner. Examples of such documents are bill of materials (BOM), schedules, expense reports, or even a phone bill.

Bill of Material. A BOM is an inventory of the components of an engineering artifact such as a plane, a building, etc. A BOM contains information about the name, cost, physical dimensions, quantity, brand, code etc. of the parts used. BOMs are used in scheduling, resource planning, manufacturing control, etc. Scanning and storing such documents in a format which can be queried later is valuable. Constrained objects are ideal for modeling BOMs because they give a compositional/hierarchical description of an engineering product, i.e., they list the product, its parts, their subparts, and so on. Also, each category of parts and their product information can be stored represented as objects and their attributes respectively. Any relation between parts can be stored as a constraint. Such a representation of a BOM can be queried for the cost or the product, the raw material required for its manufacture, etc. Computation of such queries will involve constraint solving, e.g., the cost of a part will be a sum total of the cost of its subparts each multiplied by the number representing their quantity.

Phone Bill. As another example, consider a phone bill. The contents of a phone bill list in chronological order, phone calls, their time, duration, cost, etc. The contents of a phone bill are ideally suited for modeling as constrained objects since there several constraints on attributes of every phone call. For example, the end time of a call must be less than the start time of the next call, and the cost of a call depends on the duration, destination, time of the call and the rate for such calls depends on the calling plan. A constrained object model allows specification of such constraints declaratively into the document recognizer. Such a model is also conducive to being adapted for recognizing other kinds of documents.

7.3.2 Book as a Constrained Object

One of our first case studies in developing the paradigm of constrained objects involved the representation of a book and its layout as a constrained object. Constrained objects can model not only the logical contents of a document, as shown by the examples in the previous section, but also its physical layout. The problem of layout or formatting is typically a combination of “must satisfy” constraints and “should satisfy” preferences. Often an optimal layout which satisfies certain esthetic preferences may not satisfy some of the basic layout constraints. The problem of formatting a compound document has another interesting aspect, viz., though the structure of the document and its overall layout is compositional, the problem of optimizing its layout is not. For example, the optimal layout of a chapter is not strictly composed of the optimal layouts of its sections. This is a problem of hierarchic optimization which may require relaxation (sub-optimization) of preference criteria in order to compute a satisfactory solution (layout). This case study investigates the application of constrained objects to represent the contents and layout of a simple book, with the main focus being the use of predicates, preferences and relaxation to compute a satisfactory layout.

A book is a complex compound document which is made of several logical (chapters, sections, figures, footnotes, etc.) and physical (font and its size, page size, layout of figures, margins, etc.) components. For the purposes of this case study we take a simplified form of a book, one which has chapters, chapter titles, sections, section titles, and paragraphs. We show how the concept of constrained objects can be used to model the contents and formatting of such a book. We discuss, at the end of the case study, how our model for a simple book can be extended to accommodate, figures, footnotes, references, etc. for use in modeling a more complex book.

Problem Definition. Given a simple book whose contents are grouped into the logical components: chapters (and their titles), sections (and their titles), and paragraphs, we are to format the contents of the book onto pages. Some global dimensions for the physical layout of the contents are provided, e.g., page size, size of the margins, space between two lines, and font size. We are given one or more algorithms for formatting the contents of a

paragraph that takes care of the indentation and alignment with the margin. This algorithm takes a sequence of words and returns a sequence of lines that contain the words. We are also given that there can be at most say 30 lines per page. In the setting of this example, there are certain important constraints that we must deal with while formatting, e.g.: “No section shall begin on the last line of a page.” and “No section shall end on the first line of a page.” These are called the orphan and widow constraints.

In general there can be several guidelines for obtaining an appealing format: a line should not have only one word. There should be not be too many hyphenated words in a paragraph, the white space between words should be at least or at most of a certain length, etc. Since these are guidelines and not strict constraints, we regard them as preferences. Our task is to provide a constrained object model for a simple book that expresses the above constraints and preferences and, more importantly, can also be used to compute a format satisfying these constraints and preferences.

In Section 3.1 we presented the syntax of Cob programs. In order to keep the discussion relatively simple at that point, we did not give certain details of the syntax of predicates. We now provide some enhancements to the syntax of Section 3.1 that will be used in this case study: (i) Object attributes may appear in user-defined predicates, and (ii) Selection operation may be performed on these object attributes to invoke a predicate defined in their corresponding class. In relation to the syntax of constraints and predicates given in Section 3.1.2, this means that the non-terminal *term'* can generate *complex_id* and that the non-terminals *goal* and *constraint_atom* can reduce to predicate calls of the form *X.predicate_id(terms')*. The complete syntax of Cob programs including these enhancements is given in Appendix A. It is possible to translate these enhancements into CLP or PLP code using a mechanism for name resolving along the lines of the translation of selection terms given in Section 5.1.1.

A book may be thought of as a complex object that aggregates many chapters, each of which aggregates many sections. Each section in turn aggregates many paragraphs, and each paragraph aggregates many words (see Figure 7.6). Thus, the elements of a book, viz., chapter, section, and paragraph become the basic classes of objects in the constrained object model. The physical attributes of a logical component, e.g., the page number on

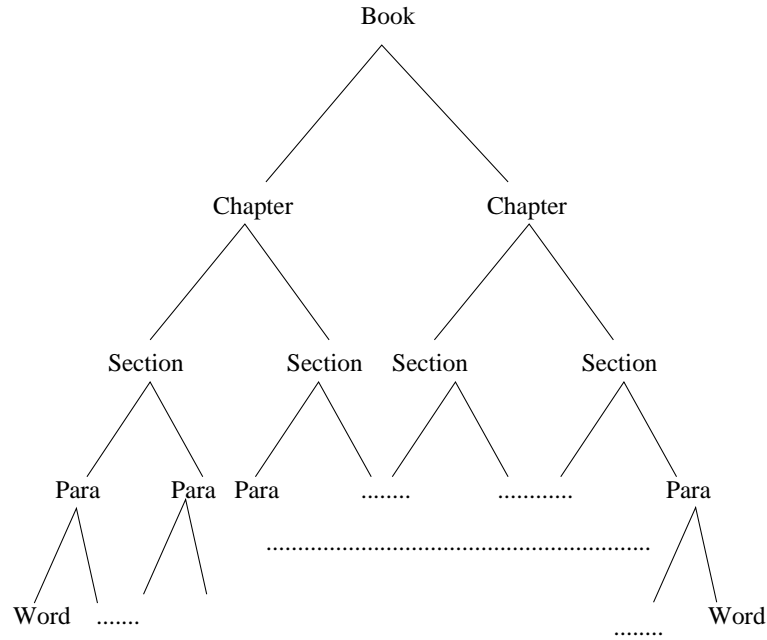


Figure 7.6: Structure of a Simple Book

which a chapter begins, also become attributes of the appropriate class.

We may think of a `format` predicate within each class of book-element (chapter, section, etc.), whose purpose is to obtain the optimal layout for the element (the reason for having a *predicate*, instead of a method, will become clear below.) In a compositional formulation, the `format` predicate of one element should invoke the `format` predicate of its constituent elements in order to construct its optimal layout.

Although we do not focus on the formatting algorithm itself, we are interested in the quality of the resultant format. Hence, each formatted element is associated with a measure of how good or bad its formatting is, viz., its so-called *badness*. We assume that the `format` predicate incorporates some such method for calculating its badness. Suppose there are several possible formats for the book using the same algorithm. The possibility of multiple solutions necessitates a preference criterion for choosing the best (most preferred) one. This is shown in the class `para`, where we state that the format with lesser badness is the preferred one:

```
format(W, L1, B1) < format(W, L2, B2) :- B1 > B2.
```

The restrictions on the formatting (mentioned in the problem definition) are better specified separately from the formatting algorithm, since the algorithm may be modified inde-

pendently while these still hold. These are specified as constraints within the class definitions of a section. We now define constrained object classes representing the logical components of a book. For simplicity we do not show the constructors in the classes below.

Book. The class `book` aggregates a sequence of `Chapters` and has an attribute representing its title. Every chapter of the book begins on a new page and the first chapter begins on page number 1. These relations are specified as constraints of the `book` class. The quantified constraint in the `book` class states that every chapter begins on an odd-numbered page: either immediately after or leaving a blank page after the previous chapter (indicated by the conditional constraints inside the quantification). The format predicate for the book calls the format predicate on each of its chapters.

```
class book {
  attributes
    chapter[] Chapters;
    string Title;
  constraints
    Chapters[1].Begin_pg = 1;
    size(Chapters, NChap);
    forall C in 1..NChap-1:
      (Chapters[C].End_pg + 1 = Chapters[C+1].Begin_pg :-
        even(Chapters[C].End_pg);
       Chapters[C].End_pg + 2 = Chapters[C+1].Begin_pg :-
        odd(Chapters[C].End_pg));
  predicates
    format(Bpgs) :- format(Chapters, Bpgs).
    format([], []).
    format([C1|C], [CP1|CPgs]) :-
      C1.format(CP1), format(C, CPgs).
}
```

Chapter. The class `chapter` aggregates a sequence of `Sections` and has attributes denoting its title and its beginning and end pages (`Begin_pg` and `End_pg` respectively). Let us assume that the title of a chapter is always placed on line number 8 and the first section begins on line number 10 of the first page of the chapter. These restrictions are represented by the first two constraints of the `chapter` class. In an alternate formulation, the title and first section's line number may be left uninitialized in the constraints clause and

their values may be passed as arguments to the constructor, thus allowing these formatting guidelines to be decided at runtime. Each section of a chapter begins on a new line, the one immediately following the end line of the previous section. The above relations are stated as constraints of the `chapter` class. The format predicate for a chapter calls the format predicate of each its sections.

```
class chapter {
  attributes
    section[] Sections;
    string Title;
    int Begin_pg, End_pg, Numpgs, Title_ln;
  constraints
    Title_ln = 8;
    Sections[1].Begin_ln = 10;
    size(Sections, NSec);
    End_pg = Begin_pg + Numpgs - 1;
    forall S in 1..NSec-1:
      Sections[S].End_ln + 1 = Sections[S+1].Begin_ln;
  predicates
    format(Cpgs) :-
      format(Sections, SLines),
      form_pgs(SLines, Cpgs, Numpgs).
    format([], []).
    format([S1|S], [SL1|SLines]) :-
      S1.format(SL1), format(S, SLines).
}
```

Section. The class `section` aggregates a sequence of paragraphs and has attributes representing its title (`Title`), the maximum number of lines per page (`Size`), and its beginning and end lines (`Begin_ln` and `End_ln` respectively). The first paragraph of a section begins on the line immediately following the title of the section. All other paragraphs of a section begin on the line following the ending line of the previous paragraph. A section ends on the same line on which its last paragraph ends. These relations are stated as simple and quantified constraints in the `section` class. The restriction that a section cannot begin on the last line of a page is modeled by the disequation $\text{Begin_ln} \bmod \text{Size} \neq 0$ where the attribute `Size` represents the maximum number of lines per page. This attribute is initialized in the constructor of the `section` class. The restriction that a section cannot end on the first line of a page is modeled similarly. Note that since the maximum number

of lines per page does not vary per section or chapter, this information may be better placed in the book class. In that case, a section will need to have access to this value through an attribute referring to the book object. The format predicate of a section calls the format predicate of its paragraphs.

```
class section {
  attributes
    para[] Paras;
    int Begin_ln, End_ln, Size, Title_ln;
    string Title;
  constraints
    size(Paras, NPara);
    Begin_ln = Title_ln;
    Begin_ln + 1 = Paras[1].Begin_ln;
    End_ln = Paras[NPara].End_ln;
    Begin_ln mod Size != 0;
    End_ln mod Size != 1;
    forall P in 1..NPara-1:
      Paras[P].End_ln + 1 = Paras[P+1].Begin_ln;
  predicates
    format(Sec_lines) :- format(paras, Sec_lines, Badness).
    format([], [], 0).
    format([P|Paras], [L|Lines], Badness) :-
      P.format(L,Badness'),
      format(Paras, Lines, Badness''),
      Badness = Badness' + Badness''.
}
```

Paragraph. The class para aggregates a sequence of words and has attributes representing its first and last lines and the total number of lines. Assume that the width of a paragraph is given to be 13 centimeters. The format predicate of a paragraph takes a list of words as input, formats them into a list of lines and computes the measure of badness for the format. This formatting algorithm may take the font size, width of paragraph, inter-word spacing, etc. into account and will usually have more than one solution for a list of words depending on how much apart the words are stretched (more or less inter-word spacing). We are not interested in the exact details of this formatting algorithm but only in the solution(s) it gives. Each solution is associated with a measure of badness which may depend upon the inter-word space, the number of hyphenated words, the number of words

on the last line of the paragraph, etc. Again, we are not interested in how the badness is measured, but only in its value (which can be different for different formats).

```
class para {
  attributes
    int Begin_ln, End_ln, Numlines;
    char[] Words;
  constraints
    End_ln = Begin_ln + Numlines;
    Width = 13;
  predicates
    format(Words, Lines, Badness) :- ... details omitted ...
  preference
    format(Words, L1, B1)  $\preceq$  format(Words, L2, B2) :- B1 > B2.
}
```

Since there can be more than one format for the same set of words, we state a preference to indicate what is an optimal format. The preference clause above states that among two formats (L1 and L2) for the same sequence of Words, the format with lesser badness (B2) is preferred over the one with more badness (B1).

Although we have not shown the constructors for any of the classes above for the sake of brevity, their definition is not difficult to understand. The constructors would essentially initialize the contents of each component, e.g., the constructor for the `chapter` class will assign a value to the `Sections` array and so on.

7.3.3 Creating and Formatting a Simple Book

Having defined the classes needed for modeling a simple book, we now describe how a sample book is created and a format for its contents is computed. A class `samplebook` can be defined along the lines of the `samplecircuit` class of Section 3.2.4. In it, first instances of paragraphs are created, by aggregating lists of words. These instances of the paragraph class are aggregated to create an instance of a section. In this way, several sections are instantiated and then chapter objects are created by aggregating these instances of the section class. Finally a book object is created by aggregating the chapters and the top level query to build a book is `B = new samplebook(VarList)`. The parameters for formatting a book, e.g., line spacing, line number at which title of a chapter appears,

size of page, width of paragraph, etc., can be passed as arguments to the constructor of the `samplebook` class.

Conceptually, the above constrained object program is translated to a PLP program along the lines given in Sections 4.1.2., 5.1.1. and 6.1.2. Subsequently, there can be two ways to format the book. Either using the query `B = new samplebook(Var_List)`, `B.format(Bpgs)`, which results in `Bpgs` containing the format; or by placing the constraint `B.format(Bpgs)` in the `samplebook` class and simply invoking the query `B = new samplebook(Var_List)`. In either case, the query is translated to a PLP query and its evaluation proceeds along the lines described in Section 6.1.3. When `B.format(Bpgs)` is evaluated, it calls the `C1.format(CP1)`, where `C1` iterates over the list of chapters of the book. Similarly when the format predicate for each chapter is evaluated, it invokes the format predicate of each of its section. Similarly a section formats each of its paragraphs. So far most of the constraints of the book, chapter, section, and paragraph objects are not sufficiently ground to be solved or verified. Once the format predicate for a paragraph is invoked these constraints start to get evaluated. For example, when a paragraph's words are formatted and its last line is determined, this tells the next paragraph where it should begin and so on. The paragraphs are formatted in a sequential order beginning with the first paragraph of the first section of the first chapter.

The format predicate of a paragraph is an optimization predicate. Its evaluation proceeds along the lines described in Section 6.1.3. The preference criterion is used to prune sub-optimal formats, resulting in an optimal format for the paragraph. In this way all paragraphs get an optimal format. Subsequently, the layout attributes of all the other objects (sections, chapters and book) get instantiated by the evaluation of their constraints. Thus a set of pages are associated with every chapter and lines with every section and paragraph of the book. This is, however, only the best-case scenario. The more interesting cases occur when the optimal layouts of all the paragraphs do not together satisfy all the constraint of the section class. We discuss two such scenarios next and explain how the preferences together with relaxation in constrained objects can address the issues raised by these cases.

Preferences and Relaxation. Suppose the optimal format of a paragraph violates one of the orphan or widow constraints. In this case the evaluation of format query `B.format(Bpgs)`

backtracks to produce obtain another optimal format for the paragraph which will satisfy all constraints. This is possible because the query is a PLP query and is evaluated using the goal evaluation scheme described in Section 6.1.3. along with the traditional backtracking of logic programming. This is why having a format predicate is more convenient than a method. If all the optimal formats for a paragraph violate the constraint, then backtracking proceeds to find an alternative optimal format for the previous paragraph, and so on till a satisfactory optimal format is found. Backtracking to obtain another optimal solution assumes that such a format exists; otherwise we need some means of relaxing the constraints.

Consider the case when the none of the optimal formats for the paragraphs of a section satisfy the orphan or widow constraint. In such a case backtracking will be unable to find a satisfactory format for the book. Hence what is really needed is a query of the form

`RELAX* B.format(Bpgs) .`

This query will first try to find a solution (a format) by backtracking through all the optimal solutions. If a solution is not found, the evaluation will proceed to relax the optimization subgoals involved in this query, viz., subgoals of the form `P.format(W,L,B)` where `P` is a paragraph object. These are optimization subgoals because the format predicate of the paragraph class is an optimization predicate. Recall the scheme for relaxing multiple optimization and hierarchic optimization goals described in Section 6.3.2. By this scheme, when formatting a paragraph, first the optimal solutions will be returned and subsequently, they will be removed from the search tree to obtain suboptimal solutions. The relaxation is with respect to the underlying preference criterion, which in this case is the badness level of a format. Thus the query `RELAX* B.format(Bpgs)` will invoke the query `RELAX* P.format(W,L,B)` and find suboptimal formats for a paragraph in the order of increasing badness.

Suppose a chapter has n sections: S_1, \dots, S_n . According to the above formulation, the paragraphs of S_1 will be formatted first, then the contents of S_2 , and so on. Within a section S_i , the contents of its first paragraph are formatted, then the second, and so on. Suppose the last paragraph of S_i ends on the last but one line of a page. No matter what format is selected for S_{i+1} , it will always begin on the last line of a page. It may seem that the computation will have to iterate through all formats of S_{i+1} to conclude that none can satisfy the orphan

constraint and hence backtrack to get an alternate format for S_i . However, this is not the case. Due to the constraint

```
forall S in 1..NSec-1:
    Sections[S].End_ln + 1 = Sections[S+1].Begin_ln
```

in the chapter class, the optimal format for S_i is rejected since it implies that the next section will begin at the bottom of a page. Thus the evaluation will try to find a suboptimal format for S_i and then proceed to format the section S_{i+1} .

Similarly, if a section S_i ends on the first line of a page, the formatting algorithm will backtrack to produce an alternate format (suboptimal or another optimal solution) for its last paragraph. If none of them satisfy the widow constraint, the preference criterion of the last but one paragraph will be relaxed, and so on.

Consider a reformulation of the problem such that the orphan and widow constraints are moved down to the class `para` and the preference on the format is moved up to the class `section`. This is a more rigid constraint since no paragraph is to begin on the last line of a page and it automatically implies that no section will violate this condition either. Such a formulation always returns the optimal solution since the constraint is in a way embedded in the definition of the format predicate of a paragraph. Suppose there were some further constraints on the section, and suppose that an optimal format violated these constraints. One may ask how the section will know which paragraph to reformat. The computational model of preferences and relaxation described in Chapter 6 accounts for such scenarios. The global preference criterion of the section class compares all possible formats for its paragraphs to compute the next best overall format of the section. Thus in both cases above, the correct query should be `RELAX* B.format(Bpgs)`.

Advantages of the Constrained Object Model. The constrained object representation of a book allows a natural and compositional specification of the logical as well as physical structure of a book. Classes provide a compact means of representing a concept and its attributes. While the formatting of the contents of a book is defined using predicates and preferences, the constraints on the format are specified separately. This allows the programmer to change the formatting algorithm without having to alter the constraints. The use of predicates within class definitions promotes a compositional or modular definition

of the formatting algorithm which is easy to understand.

Preferences allow the programmer to specify a criterion for selecting between different formats. This is a valuable construct that facilitates optimization and, along with backtracking and relaxation, can be used to compute sub-optimal solutions. Thus the constrained object model of a book shown above is better than a pure object-oriented as well as better than a purely logic programming representation.

Extending the Model. The above model of a simple book can be extended to account for details such as figures, tables, footnotes, references, etc. For example, a `figure` class may be defined that aggregates a figure, its size, orientation, etc. The `section` class may be suitably modified to aggregate paragraphs and figures and its formatting algorithm modified to take the size of the figure into account. Similarly, tables can also be incorporated in the above model. Footnotes may be associated with a word or a line and the class representing them should be aggregated by a paragraph. In that case, a paragraph class may incorporate a constraint that states that the footnote appears on the same page as its associated word or line. Thus, we can say that it is feasible to extend the model to cater for more logical components and their layout.

By extending the model with the above mentioned features, we do not greatly alter the evaluation scheme for its format predicate. The format predicate will have to be changed to incorporate the formatting algorithm and preferences for figures, tables, etc. But the evaluation scheme will essentially remain the same, viz., if a table's format violates a constraint, an alternate optimal format will be computed or the preference will have to be relaxed.

The constraints mentioned so far are related to the physical layout of the book. There is an entire separate category of constraints that are concerned with the integrity of the contents of the book. For example, checking the existence of a reference and the validity of its format. Such constraints can also be modeled using constrained objects. These type of constraints, referred to as integrity constraints in the context of databases, may use the Cob quantified existential constraint `exists` (see Cob constraint syntax in Section 3.1.2).

We made several other simplifications to the book before modeling it, e.g., we assumed that the first section of a chapter comes immediately after its title. In general this need not be the case, there may be a few or several paragraphs before the first section begins. We feel

that such details can be accommodated by suitably altering the class definitions to contain attributes representing these details. The extensions will not give rise to any fundamental problem in the issues addressed in this case study.

Conclusions and Future Work. We have shown through several examples that constrained objects provide a modular and declarative specification that is suitable for modeling structured documents. Constrained object models of documents can be used for recognition (e.g. phone bill and BOM) and querying the document for information (e.g. transcript), computing the contents of the document (e.g. tax forms) or arranging the contents in a certain format (e.g. simple book).

The case study presented in this section focused on the use of predicates, preferences, and relaxation, in the context of constrained objects, to represent and compute the problem of document formatting for a simple book. We illustrated how the use of predicates and preferences facilitates a succinct specification of the optimization criterion and facilitates the relaxation of the preference criterion in case of constraint violation. The use of the RELAX* construct allows relaxation with respect to the underlying criterion, thus generating formats in increasing order of badness. Our conclusion is that constrained objects with predicates, preferences and relaxation can provide a compact representation of a complex problem and its solution.

As a part of the future work, we intend to provide an extension of the above constrained object model incorporating several more details of the contents and format of a book. We also intend to provide detailed constrained models for other documents including those given at the beginning of this section. We may also develop domain specific interfaces for certain kinds of documents such as transcripts and tax forms that allow the user to interactively create the document and or query it for information.

Chapter 8

Conclusions and Future Work

8.1 Summary and Contributions

This dissertation reported the development of a programming language and execution environment for modeling complex systems based upon three basic concepts: objects, constraints, and visualization. Our language, Cob, facilitates a compositional specification of the structure (through objects), declarative specification of the behavior (through constraints), and allows visual development of models of complex systems. A complex system is thus represented as the aggregation of several constrained objects and the underlying computational engine performs logical inference and constraint satisfaction to obtain the resultant behavior of the system.

Constrained Objects are better than Constraints and Objects. In modeling many complex structures (especially in the engineering domain), constrained objects are preferable to the traditional paradigm of imperative objects and also the traditional constraint/constraint logic programming language: (i) It is more natural to specify the behavioral laws governing the attributes of an object as constraints rather than through imperative methods. (ii) It is more natural to model an engineering artifact as a complex object than as a complex constraint. An object has a direct counterpart in the physical world. (iii) The diagrammatic visualization of a structure can be more easily traced back to an object representation rather than a constraint representation. (iv) Object structure helps in reporting the cause of model

inconsistency.

Cob Language and Environment. At the modeling level, Cob provides quantified and conditional constraints, logic variables and preferences. These facilitate compact specifications of a wide range of arithmetic and symbolic constraints, and optimization criteria. This set of modeling features along with constraint solving and optimization capabilities, specification and querying of partial models, visual interfaces for drawing constrained object class diagrams (CUMML) and domain-specific drawings, visual interfaces for interactive execution and debugging, makes Cob a comprehensive language for modeling complex systems. Compared to Cob, other constrained object languages [10, 55, 30, 34] provide only limited features for modeling only certain category or domain of constraints, no constructs for modeling general optimization problems and little or no facility for visual development of models.

Semantics of Constrained Objects. We have defined set theoretic semantics for constrained object programs where the meaning of a Cob class is defined as the set of values for its attributes that satisfy the constraints of the class. These semantics are based on a mapping from Cob programs to constraint logic programs (CLP). Essentially, a Cob class is mapped to a CLP predicate and the semantics of the class are defined as the logical consequences of the predicate. We have developed a novel implementation of constrained objects based on this mapping. CLP semantics however, cannot account for all features of the Cob language, viz., conditional constraints which model state dependent behavior. We have therefore defined conditional-CLP (CCLP) programs which are an augmentation of CLP programs with conditional constraints and have given their logical semantics. Cob programs with conditional constraints are translated to CCLP programs. The operational semantics of a Cob program with respect to a query are based on the rewrite semantics of CLP programs which are extended to account for conditional constraints. No other constrained object language or modeling system provides such rigorous theoretic foundations for constrained objects as the semantics provided in this dissertation.

Partial Evaluation. Due to the limitations of the CLP(R) (constraint logic programming over the real number domain) language, the basic implementation of constrained objects by translating them to CLP(R) predicates cannot handle non-linear constraints, conditional constraints and may not always yield adequate performance for large-scale systems. Hence we have developed a technique for partial evaluation of Cob programs that generates optimized code. This approach has enabled us to develop techniques for: (i) incremental constraint satisfaction and model revision (ii) visual debugging of Cob programs (iii) interfacing Cob with systems such as Maple for solving non-linear constraints and (iv) handling conditional constraints. During partial evaluation of a Cob program, we build its underlying *constrained-object graph*. This graph serves as a domain-independent visual representation for a Cob program that can be used for interactive execution and debugging.

Optimization and Relaxation. Modeling complex systems involves constraint satisfaction as well as optimization (minimize expenses, maximize efficiency, meet client's preferences, etc.). Therefore, we provide *preference* constructs within a Cob class definition for specifying optimization problems. The semantics of Cob programs with preferences are based on a translation to preference logic programs (PLP) and we presented a reformulation of PLP operational semantics [38] in the form of rewrite rules. We have shown through several examples that in optimization problems, there is often interest in the optimal as well as suboptimal solutions. Suboptimal solutions can be obtained by relaxation of the preference criterion either with respect to an external constraint or with respect to the underlying preference criterion itself. We feel that the latter form of relaxation introduced in this dissertation is the most basic form relaxation that can be used to compute suboptimal solutions in different kinds of optimization and sub-optimization problems. The reformulation of PLP operational semantics that we presented enables us to give an intuitive operational semantics for PLP with relaxation goals that accounts for hierarchic as well as recursive optimization predicates. Although languages such as [42, 13, 11] deal with relaxation issues in logic programming languages, they do not address the relaxation of the general optimization problems that can be modeled in PLP.

Application. We have shown a variety of examples to illustrate that the paradigm of constrained objects is very well-suited for modeling complex systems from different domains. A domain of particular interest is engineering structures such as circuits, gears, mixers, separators, etc. With the aid of domain-specific visual interfaces, the resulting paradigm has considerable potential for both pedagogic purposes as well as for more advanced applications. On a technical level, our language advances previous work by showing the use of a number of new features in modeling, especially conditional and quantified constraints, as well as preferences.

8.2 Directions for Future Work

8.2.1 Language Design

Dynamic Constrained Objects: Most of the systems we have modeled in Cob so far have been of a static nature, i.e., the values of attributes of the objects are time-invariant. We are currently investigating the use of Cob to model dynamic systems that can be described through behavioral laws, e.g, hydrological and biological processes, physical systems, sensor networks, etc. In the context of modeling hydrological processes the state of a geological surface can be its water content. This state varies with time depending on several factors: precipitation, water content of neighboring surfaces, run off, ground water level, and the previous state, etc. Currently the history of changes to an attribute is represented as an array, and we are exploring better and more efficient techniques for the representation of time varying behavior.

Mutable Constrained Objects: We are also interested in providing state change operations in Cob at the modeling level. Our technique for partial evaluation enables us to efficiently resolve constrained objects for a certain class of problems (see Section 5.2.4). But for other general Cob models, currently, if the state changes, the entire constrained object model must be re-evaluated. This is inefficient and thus a technique for associating a mutable state with a constrained object is desirable. Currently we are exploring the use of SICStus objects for this purpose. We are examining the problem of constraint-based information sharing

between objects whose state changes frequently and subsequently affects the sharing of information between them.

Modeling Features: The modeling features provided in Cob are motivated mostly by problems in the engineering domain. A continued exploration of problems in the engineering, organizational enterprises, hydrological domains will serve as a guide to adding new constraint constructs to the Cob language. Expanding the current set of features to include constructs such as differential equations, integrals, time varying constraints etc. will make Cob applicable to an even wider range of problems.

Constructive Negation: In Chapter 4 we described the operational semantics of conditional constraints. The case when a consequent is false entails the negation of the antecedent which may have more than one literal. Negation of a conjunction of non-ground literals is a problem that has been closely studied and involves issues of termination. Constructive negation is the process of obtaining solutions for negated non-ground goals and a practical implementation of it is needed in Cob.

8.2.2 Modeling Environment

Domain Specific Interfaces: As the application of Cob to different engineering domains is investigated, domain specific interfaces can be developed for each of these domains. A project under development is constraint-based information sharing in a battlefield scenario. Information about moving targets is shared between stationary sensors. The sensors that should exchange information depend upon the range of the sensors and their location as well as the location of the target. The underlying Cob model computes how the information is shared and between whom and the visual interface displays the model. There is also ongoing work to develop a domain specific interface for hydrological modeling that studies precipitation and the flow of water on geological surfaces.

Pedagogical Tools: There is growing interest from faculty in the departments of Architecture and Civil Engineering at University at Buffalo, to develop visual interfaces with underlying Cob representations that will be used for teaching students the fundamental concepts of architecture and engineering. One possible scheme is to have a 3-D visualization of, say,

different gears, trusses, beams etc. with which the students can interact. Through a visual interface, the student can examine the laws of physics and the various forces acting on the entity; place two entities together to see how their forces interact, etc. Problems that are usually solved using paper and pencil can be tested on this tool which will help students understand the source of their mistakes/miscalculations. The partial specification of Cob models is especially advantageous in this scenario since the modeler (in this case a student) is not fully aware of how the model should be initialized.

Model Abstraction: When complex system is very large (i.e., consists of many sub-objects), it may be very time-consuming to do solve a detailed model. In this case it may be necessary to work with a simplified model, i.e., a “cross-section” of the model so as to get an approximate answer in reasonable time. This problem may be called model abstraction. Thus it may be useful to expand the Cob modeling environment with facilities for viewing and evaluating a certain part of a Cob model.

8.2.3 Semantics

General Conditional Constraints: In Sections 4.1.3 and 4.1.4 we defined the formal semantics of Cob programs (with and without conditional constraints). In Sections 4.2.2 and 4.2.3 we defined the formal operational semantics of Cob programs with simple conditional constraints and gave proof of their soundness. We also discussed in brief the issues involved in showing their completeness. In Section 4.2.4 we gave a brief overview of the operational semantics of general conditional constraints. As part of future work, we would like to give a comprehensive operational semantics for Cob programs with rigorous proof of their soundness and completeness results.

Optimization: In Section 6.1 we gave a description of the declarative semantics of PLP programs and gave a reformulation of their operational semantics. It will be useful to provide a rigorous theoretic evaluation of this reformulation and formally prove its equivalence to the PLP semantics given in [38]. An implementation of these operational semantics will also prove to be interesting work.

Relaxation: In Section 6.2 we discussed two forms of relaxation: with respect to a given

constraint and with respect to the underlying preference criterion. The latter is a more general form of relaxation and we described an intuitive approach to its operational semantics in Section 6.3. As a part of future research, we would first like to formalize the declarative semantics of the different relaxation goals and then prove the soundness and completeness of the operational semantics that we proposed in Section 6.3.

8.2.4 Interactive Execution

Drawing: The Cob tool for interactive execution described in Section 5.3 displays the object graph of a complex constrained object. We currently have a simple drawing scheme that gives a satisfactory display for a tree. For more complicated graphs (i.e., those with cycles, large number of nodes, etc.) this scheme may not always give a very intuitive and legible display of the object graph. For example, the object graph for the heatplate example in Section 3.2.2 cannot be displayed using the existing algorithm. We need to develop graph drawing algorithms so that as far as possible, the object graph bears resemblance to the original structure that the program is modeling.

Explaining the Cause of Constraint Violation: When a constraint is violated, the Cob tool for interactive execution and debugging (Section 5.3) displays the object to which the constraint belongs. Although this is the first step in locating the immediate cause for error, it may not be sufficient for understanding the source of the constraint violation. This is because when a constraint such as $A = B$ fails, the reason may be an incorrect initialization of A or B either of which could be initialized due to the solving of another constraint. Hence the real cause of error may be located several such logical steps before the constraint $A = B$ is encountered. Thus, tracing the order in which nodes and their constraints were solved, i.e., giving a form of a proof tree for constraint solving, can be of importance to the modeler. We therefore feel that such a facility should be developed for the Cob tool for interactive execution.

Domain Specific Debugging: The current debugger works with the object graph of the Cob model. This is useful for debugging Cob programs that are created directly as textual code or through the CUMML tool. For Cob programs generated through the domain specific interfaces, the debugger should be linked to the diagram (since the modeler is not aware of

the Cob code generated from the diagram). A modeler is likely to find this debugging via diagrams to be more useful and easier to follow than tracing the generated Cob program.

Incremental Constraint Solving: Partial evaluation has enabled us to reuse computation when the attributes of a model change but its structure remains same. Traditional approaches to incremental constraint satisfaction focus on reducing re-computation when a constraint or value is modified. These algorithms use a dependency graph in which nodes correspond to variables and edges correspond to constraints. We feel that since some of this information is inherent to a constrained object graph, an investigation of its use in incremental constraint solving might lead to interesting results.

Bibliography

- [1] H. Abramson and M. Rogers. *Meta-Programming in Logic Programming*. MIT Press, 1989.
- [2] M. Agarwal and J. Cagan. A Blend of Different Tastes: The Language of Coffee-makers. *Environment and Planning B: Planning and Design*, 25:205–226, 1998.
- [3] A. Aho and J. Ullman. *Principles of Compiler Design*, chapter 12–13. Narosa Publishing House, 1999.
- [4] H. Ait-Kaci and A. Podelski. Towards a Meaning of LIFE. *Journal of Logic Programming*, 16(3):195–234, 1993.
- [5] C. Atay, K. McAloon, and C. Tretkoff. 2LP: A Highly Parallel Constraint Logic Programming Language. In *SIAM Conference on Parallel Processing for Scientific Computing*, 1993.
- [6] J. Avenhaus, R. Göbel, B. Gramlich, K. Madlener, and J. Steinbach. TRSPEC: A Term Rewriting Based System for Algebraic Specifications. In Stéphane Kaplan and Jean-Pierre Jouannaud, editors, *Conditional Term Rewriting Systems, 1st International Workshop (CTRS)*, volume 308 of *Lecture Notes in Computer Science*, pages 245–248. Springer, 1988.
- [7] B. De Backer and H. Beringer. Intelligent Backtracking for CLP Languages, An Application to CLP(R). In *International Logic Programming Symposium*, pages 405–419, 1991.

- [8] R. Barták. Constraint Programming - What is behind? In *Third Workshop on Constraint Programming in Decision and Control*, June 1999.
- [9] R. Barták. Theory and Practice of Constraint Propagation. In *Third Workshop on Constraint Programming in Decision and Control*, June 2001.
- [10] A. Borning. The Programming Language Aspects of Thinglab, A Constraint-Oriented Simulation Laboratory. *ACM TOPLAS*, 3(4):353–387, 1981.
- [11] A. Borning, M.J. Maher, A. Martindale, and M. Wilson. Constraint Hierarchies and Logic Programming. In *Proc. 6th Intl. Conf. on Logic Prog.*, pages 149–164, 1989.
- [12] A. Brodsky and Y. Kornatzky. The LyriC Language: Querying Constraint Objects. *Proc. ACM SIGMOD*, 1995.
- [13] A. Brown, S. Mantha, and T. Wakayama. Logical Reconstruction of Constraint Relaxation Hierarchies in Logic Programming. In *Proc. of 7th Intl. Symp. on Methodologies for Intelligent Systems*, LNAI 689, Trondheim Norway, 1993.
- [14] A. Brown, S. Mantha, and T. Wakayama. Preference Logics: Towards a Unified Approach to Non-Monotonicity in Deductive Reasoning. *Annals of Mathematics and Artificial Intelligence*, 10:233–280, 1994.
- [15] B. Cao and T. Swift. Preference Logic Grammars: Fixed-point Semantics and Application to Data Standardization. *Artificial Intelligence: An International Journal*, to appear.
- [16] M. Carlsson, G. Ottosson, and B. Carlson. An Open-Ended Finite Domain Constraint Solver. In *Programming Languages: Implementations, Logics, and Programs*, 1997.
- [17] W. Chen and D. Warren. Objects as Intensions. In R.A. Kowalski and K.A. Bowen, editors, *Fifth International Conference on Logic Programming*, volume 1, pages 404–419. The MIT Press, Cambridge, Mass., 1988.

- [18] J. Chinneck. Finding Minimal Infeasible Sets of Constraints in Infeasible Mathematical Programs. Technical Report SCE-93-01, Department of Systems and Computer Engineering, Carleton University, Ottawa, 1993.
- [19] J. Chomicki. Querying with Intrinsic Preferences. In *EDBT 2002, 8th International Conference on Extending Database Technology*, volume 2287 of *Lecture Notes in Computer Science*, pages 34–51. Springer, 2002.
- [20] J. Conery. Logical objects. In R.A. Kowalski and K.A. Bowen, editors, *Fifth International Conference on Logic Programming*, volume 1, pages 420–434. The MIT Press, Cambridge, Mass., 1988.
- [21] R. Dechter. Constraint Networks. In Stuart C. Shapiro, editor, *Encyclopedia of Artificial Intelligence*, volume 1. Addison-Wesley Publishing Company, 1992.
- [22] R. Dechter and J. Pearl. Network based heuristics for constraint satisfaction problems. *Artificial Intelligence*, 34:1–38, 1987.
- [23] J. Delgrande, T. Schaub, and H. Tompits. Logic Programs with Compiled Preferences. In *8th International Workshop on NonMonotonic Reasoning*. 2000.
- [24] M. Dincbas, P. Van Hentenryck, H. Simonis, A. Aggoun, T. Graf, and F. Berthier. The Constraint Logic Programming Language CHIP. In *International Conference on Fifth Generation Computer Systems*, 1988.
- [25] S. Etalle, M. Gabbrielli, and E. Marchiori. A transformation system for CLP with dynamic scheduling and CCP. In *Proceedings of the 1997 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 137–150. ACM Press, 1997.
- [26] F. Fages. On the Semantics of Optimization Predicates in CLP Languages. In *Proc. 13th Conf. on Foundations of Software Technology and Theoretical Computer Science*, 1993.

- [27] F. Fages, J. Fowler, and T. Sola. Handling Preferences in Constraint Logic Programming with Relational Optimization. In *Proc. 6th Intl. Symp. PLILP*, LNCS 844, pages 261–276, 1994.
- [28] F. Fioravanti, A. Pettorossi, and M. Proietti. Rules and Strategies for Contextual Specialization of Constraint Logic Programs. *Electronic Notes in Theoretical Computer Science*, 30(2), 1999.
- [29] P. Fishburn. Preference Structures and their Numerical Representations. *Theoretical Computer Science*, 217:359–383, 1999.
- [30] B. N. Freeman-Benson and A. Borning. Integrating Constraints with an Object Oriented Language. In *Proc. European Conference On Object-Oriented Programming*, pages 268–286, 1992.
- [31] E. Freuder. A Sufficient Condition for Backtrack-free Search. *Journal of the ACM*, 1982.
- [32] E. C. Freuder. Partial Constraint Satisfaction. In *Proc. 11th Intl. Jt. Conf. on Artificial Intelligence*, pages 278–283, 1989.
- [33] E.C. Freuder and R.J. Wallace. Heuristic Methods for Over-Constrained Constraint Satisfaction Problems. In *Proc. CP’95 Workshop on Overconstrained Systems*, 1995.
- [34] P. Fritzson and V. Engelson. Modelica - A Unified Object-Oriented Language for System Modeling and Simulation. In *12th European Conference on Object-Oriented Programming*, 1998.
- [35] T. Frühwirth. Constraint Handling Rules. In Andreas Podelski, editor, *Constraint Programming: Basics and Trends*, number 910 in *Lecture Notes in Computer Science*, pages 90–107. Springer Verlag, March 1995., 1995.
- [36] T. Frühwirth. Theory and Practice of Constraint Handling Rules. *Journal of Logic Programming, Special Issue on Constraint Logic Programming*, 37(1-3):95–138, October 1998.

- [37] P. Gärdenfors. *Belief Revision*. Cambridge Computer Tracts. Cambridge University Press, 1992.
- [38] K. Govindarajan. *Optimization and Relaxation in Logic Languages*. PhD thesis, Department of Computer Science, SUNY - Buffalo, 1997.
- [39] K. Govindarajan, B. Jayaraman, and S. Mantha. Preference Logic Programming. In *Proc. 12th International Conference on Logic Programming*, pages 731–745. MIT Press, 1995.
- [40] K. Govindarajan, B. Jayaraman, and S. Mantha. Optimization and Relaxation in Constraint Logic Languages. In *Proc. 23rd ACM Symp. on Principles of Programming Languages*, pages 91–103, 1996.
- [41] K. Govindarajan, B. Jayaraman, and S. Mantha. Preference Queries in Deductive Databases. *New Generation Computing*, 19(1), 2000.
- [42] B. Grosz. Courteous Logic Programs: Prioritized Conflict Handling for Rules. Technical report, IBM Research Division, T. J. Watson Research Center, 1997.
- [43] MATHLAB Group. *Macsyma Reference Manual*. MIT Press.
- [44] N. Heintze, J. Jaffar, C. Lim, X. Michaylov, P. Stuckey, R. Yap, and C. Yee. The CLP Programmer’s Manual. Technical Report 73, Department of Computer Science, Monash University, June 1986.
- [45] N. Heintze, S. Michaylov, and P. Stuckey. CLP(R) and Some Electrical Engineering Problems. In *Fourth International Conference on Logic Programming*, pages 675–703, 1987.
- [46] N. Heintze, S. Michaylov, and P. Stuckey. CLP(R) and Some Electrical Engineering Problems, 1992.
- [47] R. Helm, I. Holland, and D. Gangopadhyay. Contracts: Specifying Behavioural Compositions in Object-Oriented Systems. In *Proc. Object-Oriented Programming, Systems, and Applications (OOPSLA)*, 1990.

- [48] R. Helm, T. Huynh, C. Lassez, and K. Marriott. A Linear Constraint Technology for Interactive Graphics Systems. In *Graphics Interface*, pages 301–309, 1992.
- [49] P. Van Hentenryck. *Constraint Satisfaction in Logic Programming*. MIT Press, 1989.
- [50] P. Van Hentenryck, V. Saraswat, and Y. Deville. Constraint Processing in cc(fd). In A. Podelski, editor, *Constraint Programming: Basics and Trends*, number 910 in LNCS. Springer-Verlag, 1995.
- [51] P. Van Hentenryck and Vijay A. Saraswat. Strategic Directions in Constraint Programming. *ACM Computing Surveys*, 28(4):701–726, December 1996.
- [52] M. Hermenegildo and The CLIP Group. Some methodological issues in the design of CIAO - a generic, parallel concurrent constraint system. In *Principles and Practice of Constraint Programming*, LNCS 874, pages 123–133. Springer-Verlag, 1994.
- [53] T. Hickey and D. Smith. Toward the partial evaluation of CLP languages. In *Symposium on Partial Evaluation and Semantics-based Program Manipulation*, pages 43–51. ACM Press, 1991.
- [54] B. Horn. Constraint Patterns As a Basis For Object Oriented Programming. In *Proc. Object-Oriented Programming, Systems, and Applications (OOPSLA)*, 1992.
- [55] B. Horn. *Constrained Objects*. PhD thesis, CMU, November 1993.
- [56] H. Hosobe, K. Miyashita, S. Takahashi, S. Matsuoka, and A. Yonezawa. Locally Simultaneous Constraint Satisfaction. In *Workshop on Principles and Practice of Constraint Programming*, pages 51–62. Springer-Verlag LNCS, 1994.
- [57] J. Hsiang and M. Rusinowitch. A New Method for Establishing Refutational Completeness in Theorem Proving. In Jörg H. Siekmann, editor, *8th International Conference on Automated Deduction (CADE)*, volume 230 of *Lecture Notes in Computer Science*, pages 141–152. Springer, 1986.
- [58] International Business Machines Corporation. CLP(R) Version 1.2, 1992.

- [59] J. Jaffar and J. L. Lassez. Constraint Logic Programming. In *Proc. 14th ACM Symp. on Principles of Programming Languages*, pages 111–119, 1987.
- [60] J. Jaffar, M. Maher, K. Marriot, and P. Stuckey. The Semantics of Constraint Logic Programs. *The Journal of Logic Programming*, 37:1–46, 1998.
- [61] J. Jaffar and M. J. Maher. Constraint Logic Programming: A Survey. *Journal of Logic Programming*, 1994.
- [62] J. Jaffar, P. Michaylov, P. Stuckey, and R. Yap. The CLP(R) Language and System. *ACM Transactions on Programming Languages and Systems*, 14(3):339–395, July 1992.
- [63] J. Jaffar and S. Michaylov. Methodology and Implementation of a CLP System. In J. L. Lassez, editor, *Fourth International Conference on Logic Programming*, volume 1. MIT Press, 1987.
- [64] M. Jampel, E. Freuder, and M. Maher, editors. *Over-Constrained Systems*. Lecture Notes in Computer Science. Springer, 1996.
- [65] S. Janson and S. Haridi. Programming Paradigms of the Andorra Kernel Language. In V. Saraswat and K. Ueda, editors, *Logic Programming, Proceedings of the 1991 International Symposium*, pages 167–186. The MIT Press, 1991.
- [66] B. Jayaraman, K. Govindarajan, and S. Mantha. Preference Logic Grammars. *Computer Languages*, 24(3):179–196, 1998.
- [67] B. Jayaraman and P. Tambay. Semantics and Applications of Constrained Objects. Technical report, Department of Computer Science, SUNY - Buffalo, 2001.
- [68] B. Jayaraman and P. Tambay. Modeling Engineering Structures with Constrained Objects. In *Proc. Symposium on Practical Aspects of Declarative Languages*, 2002.
- [69] B. Kristensen, O. Madsen, B. Moller-Perdersen, and K. Nygaard. Object-Oriented Programming in the BETA Programming Language. Norwegian Computing Center, Oslo and ComputerScience Department, Aarhus University, Aarhus, Denmark, 1989.

- [70] V. Kumar. Algorithms for Constraint-Satisfaction Problems: A Survey. *AI Magazine*, 13(1):32–44, 1992.
- [71] J.H.M. Lee and P.K.C. Pun. Object Logic Integration: A Multiparadigm Design Methodology and a Programming Language. *Computer Languages*, 23(1):25–42, 1997.
- [72] W.J. Leler. *The Specification and Generation of Constraint Satisfaction Systems*. Addison-Wesley, 1987.
- [73] J. Lloyd. *Foundations of Logic Programming*, pages 6–7. Springer-Verlag, 2nd edition, 1987.
- [74] J. Lloyd and J. Shepherdson. Partial Evaluation in Logic Programming. Technical Report 87-09, Univ. of Bristol, 1987.
- [75] G. Lopez. *The Design and Implementation of Kaleidoscope, A Constraint Imperative Programming Language*. PhD thesis, University of Washington, 1997.
- [76] G. Lopez, B. Freeman-Benson, and A. Borning. Kaleidoscope: A Constraint Imperative Programming Language. In B. Mayoh, E. Tyugu, and J. Penjaam, editors, *Constraint Programming: Proceedings 1993 NATO ASI Parnu, Estonia*, pages 305–321. Springer, 1994.
- [77] G. Lopez, B. N. Freeman-Benson, and A. Borning. Constraints and Object Identity. In *Proc. European Conference On Object-Oriented Programming*, 1994.
- [78] A. Mackworth and E. Freuder. The Complexity of Constraint Satisfaction Revisited. *Artificial Intelligence*, 59(1-2):57–62, 1993.
- [79] L. Mandel. *Constrained Lambda Calculus*. PhD thesis, Ludwig-Maximilians-Universitat Munchen,, 1995.
- [80] S. Mantha. *First-Order Preference Theories and their Applications*. PhD thesis, University of Utah, November 1991.
- [81] K. Marriot and P. Stuckey. *Programming with Constraints*. The MIT Press, 1998.

- [82] K. Marriott and P. J. Stuckey. Semantics of Constraint Logic Programs with Optimization. *Letters on Programming Languages and Systems*, 2(1-4):181–196, 1993.
- [83] J. Martins. Belief Revision. In Stuart C. Shapiro, editor, *Encyclopedia of Artificial Intelligence*, pages 110–116. John Wiley & Sons, second edition, 1992.
- [84] R. Mayne and S. Margolis. *Introduction to Engineering*. McGraw-Hill, 1982.
- [85] B. Meyer. *Eiffel: The Language*. Prentice-Hall, 1992.
- [86] S. Minton, M. Johnston, A. Philips, and P. Laird. Minimizing Conflicts: A Heuristic Repair Method for Constraint Satisfaction and Scheduling Problems. *Artificial Intelligence*, 58(1–3):161–206, 1992.
- [87] C. Moss. *Prolog++: The Power of Object-Oriented and Logic Programming*. Addison-Wesley, 1994.
- [88] M. Otter, C. Schlegel, and H. Elmqvist. Modeling and Realtime Simulation of An Automatic Gearbox Using Modelica.
- [89] C. Oussalah and V. Puig. Integrating Constraints in Complex Objects. In *Fifth International Conference on Information and Knowledge Management*, pages 189–196, 1996.
- [90] R. Peak. Automating Product Data-Driven Analysis Using Multifidelity Multidirectional Constrained Objects. Invited Presentation, NASA STEP for Aerospace Workshop, Jet Propulsion Lab, Pasadena CA, Jan 2000.
- [91] J-F. Puget. A C++ implementation of CLP. In *Singapore Conference on Intelligent Systems*, September 1994.
- [92] I. Ramakrishnan, P. Rao, K. Sagonas, T. Swift, and D. Warren. Efficient Tabling Mechanisms for Logic Programs. In L. Sterling, editor, *Twelfth International Conference on Logic Programming*. MIT Press, 1995.
- [93] P. Rao, K. Sagonas, T. Swift, D. Warren, and J. Freire. XSB: A System for Efficiently Computing WFS. In Jürgen Dix, Ulrich Furbach, and Anil Nerode, editors, *Logic*

- Programming and Nonmonotonic Reasoning, 4th International Conference.*, volume 1265 of *Lecture Notes in Computer Science*, pages 431–441. Springer, 1997.
- [94] F. Rossi, V. Dahr, and C. Petrie. On the equivalence of constraint satisfaction problems. In *European Conference on Artificial Intelligence (ECAI-90)*, 1990.
 - [95] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1998.
 - [96] K. Sagonas, T. Swift, and D. Warren. XSB as an Efficient Deductive Database Engine. In *SIGMOD*, pages 442–453, 1994.
 - [97] D. Sahlin. *An Automatic Partial Evaluator for Full Prolog*. PhD thesis, Department of Telecommunication and Computer Systems, Stockholm, Sweden, March 1991.
 - [98] C. Sakama and K. Inoue. Prioritized Logic Programming and its Application to Commonsense Reasoning. *Artificial Intelligence*, 123((1-2)):185–222, 2000.
 - [99] V. Saraswat. Concurrent Constraint Programming. ACM Doctoral Dissertation Award and Logic Programming Series.
 - [100] E. Shapiro. The Family of Concurrent Logic Programming Languages. *ACM Computing Surveys*, 21(3), 1989.
 - [101] S. Shapiro and D. McKay. Inference with Recursive Rules. In *First Annual National Conference on Artificial Intelligence*, pages 151–153. Morgan Kaufmann, 1980.
 - [102] J. Shoenfield. *Mathematical Logic*. Reading MA. Addison-Wesley, 1967.
 - [103] R. Shostak. Deciding Linear Inequalities by Computing Loop Residues. *Journal of the ACM*, 28(4):769–779, October 1981.
 - [104] SICS-AB. SICStus Prolog, 2002.
 - [105] G. Smolka. The Oz Programming Model. In Jan van Leeuwen, editor, *Computer Science Today: Recent Trends and Developments*, volume 1000 of *Lecture Notes in Computer Science*, pages 324–343. Springer-Verlag, Berlin, 1995.

- [106] G. Smolka. Constraint Programming in Oz (Abstract). In *Proc. Intl. Conference on Logic Programming*, pages 37–38, 1997.
- [107] J. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. The MIT Press, 1979.
- [108] I. Sutherland. Sketchpad: A man-machine graphical communication system. In *Spring Joint Computer Conference*, pages 329–346. IFIPS, 1963.
- [109] The MathWorks, Inc. Matlab Version 6.1.0.450 Release 12.1, 2001.
- [110] J. Warmer and A. Kleppe. *The Object Constraint Language*. Addison-Wesley, 1999.
- [111] D. Warren. Programming with tabling in XSB. In David Gries and Willem P. de Roever, editors, *International Conference on Programming Concepts and Methods*, volume 125 of *IFIP Conference Proceedings*, pages 5–6. Chapman & Hall, 1998.
- [112] Waterloo Maple Inc. Maple 7, 2001.
- [113] M. Wilson and A. Borning. Hierarchical Constraint Logic Programming. *Journal of Logic Programming*, 16:277–318, 1993.
- [114] G. Winskel. *The Formal Semantics of Programming Languages, An Introduction*. Foundations of Computing. The MIT press, 1993.
- [115] S. Wolfram. *Mathematica: A System of Doing Mathematics by Computer*. Addison-Wesley, 1991.
- [116] S. Wu. Integrating Logic and Object-Oriented Programming. *OOPS Messenger*, 2(1):28–37, January 1991.

Appendix A: Complete Syntax of Cob Programs

```

program ::= class_definition+$
class_definition ::= [ abstract ] class class_id
                        [ extends class_id ] { body }
body ::= [ attributes attributes ]
           [ constraints constraints ]
           [ predicates pred_clauses ]
           [ preferences pref_clauses ]
           [ constructors constructor_clause ]
attributes ::= decl ; [ decl ; ]+
decl ::= type id_list
type ::= primitive_type_id | class_id | type [ ]
primitive_type_id ::= real | int | bool | char | string
id_list ::= attribute_id [ , attribute_id ]+
constraints ::= constraint ; [ constraint ; ]+
constraint ::= simple_constraint | quantified_constraint
                | creational_constraint
creational_constraint ::= complex_id = new class_id ( terms )
quantified_constraint ::= forall var in enum : ( constraints )
                        | exists var in enum : ( constraints )
simple_constraint ::= conditional_constraint | constraint_atom
conditional_constraint ::= constraint_atom :- literals
constraint_atom ::= term relop term | constraint_predicate_id ( terms )
                        | complex_id . predicate_id ( terms )
relop ::= = | != | > | < | >= | <=
term ::= constant | var | complex_id
           | arithmetic_expr | func_id ( terms )
           | [ terms ] | ( term )
           | aggreg_op var in enum : term

```

```

    aggreg_op ::= sum | prod | min | max
    terms ::= term [ , term ]+
    aggreg_op ::= sum | prod | min | max
    complex_id ::= attribute_id[.attribute_id]+ | complex_id [ term ]
    literals ::= literal [ , literal ]+
    literal ::= [ not ] atom
    atom ::= predicate_id(terms) | constraint_atom
    pred_clauses ::= pred_clause . [ pred_clause . ]+
    pred_clause ::= clause_head :- clause_body
    pred_clause ::= clause_head.
    clause_head ::= predicate_id( terms' )
    clause_body ::= goal [ , goal ]+
    goal ::= [ not ] predicate_id(terms')
    | complex_id.predicate_id(terms')
    terms' ::= term' [ , term' ]+
    term' ::= constant | var' | complex_id | func_id(terms')
    pref_clauses ::= pref_clause . | pref_clause . pref_clauses
    pref_clause ::= min arithmetic_expr .
    pref_clause ::= max arithmetic_expr .
    pref_clause ::= p(s1) ≤ p(s2) :- clause_body
    constructor_clauses ::= constructor_clause+
    constructor_clause ::= constructor_id(formal_pars) { constructor_body }
    constructor_body ::= constraints

```

Appendix B: Compiling and Running Cob Programs

Compilation Modes. The Cob compiler translates a Cob program to a CLP(R) program. Essentially, each class definition translates to one predicate clause. We use the underlying CLP(R) engine for constraint handling. Depending upon whether the programmer wants to work with the compiled CLP(R) file or directly run the executable, there are different modes of compilation described below.

1. To compile a Cob file named `foo.cob`, into a file `foo.clpr` use the command

```
$ /projects/tambay/temp/compiler/cob foo.cob foo.clpr
```

The clp program in `foo.clpr` should be run in the sicstus clpr module. To do this, start a prolog process and load in the file `foo.clpr`. Cob queries can now be evaluated at the prolog prompt. Note however, that the queries must be calls to the constructor of some constrained object appended with an extra argument.

```
$ prolog
```

```
|?- [ 'foo.clpr' ].
```

```
|?- samplecircuit(_,_).
```

2. An alternate method of compilation is to compile a Cob file named `foo.cob`, into a file named `foo.clpr` and place the compiled (in prolog) clpr file in `foo.run` using the command

```
$ /projects/tambay/temp/compiler/cob foo.cob foo.clpr  
-e foo.run
```

To run the executable, type in its filename and press return. A prolog prompt will be displayed. Queries should be given at this prompt. Note that the queries must be calls to the constructor of some constrained object appended with an extra argument.

```
$ foo.run
```

```
|?- samplecircuit(_,_).
```

3. A third way to compile is with the `-c` option. This will create the executable as in the previous command but with a cobinterface above the prolog interpreter. To compile in this way, run the command

```
$ /projects/tambay/temp/compiler/cob foo.cob foo.clpr  
-e foo.run -c
```

To run the executable, type in its filename and press return. A cob prompt will be displayed. Queries can be entered at this prompt in the form of calls to constructors of constrained objects.

```
$ foo.run
```



```
cob_query ?- samplecircuit(_).
```

During the evaluation of the query, if there is an exception, then control will return to the prolog prompt. To return to the `cob_query` mode, type the `cob_prompt` command.

Example Run. Suppose a file named `circuit.cob` contains the Cob program shown below.

```
class component {
  attributes
    real V, I, R;
  constraints
    V = (I * R);
  constructors component(V1, I1, R1) {
    V = V1; I = I1; R = R1;
  }
}
```

```
class series extends component {
  attributes
    component [] Cmp;
  constraints
    forall C in Cmp: C.I = I;
    sum C in Cmp: C.V = V;
    sum C in Cmp: (C.R) = R;
  constructors series(A) {
    Cmp = A;
  }
}
```

```
class parallel extends component {
  attributes
    component [] PC;
  constraints
    forall X in PC: X.V = V;
    sum X in PC: (X.I) = I;
    sum X in PC: (1/X.R) = 1/R;
  constructors parallel(B) {
    PC = B;
  }
}
```

```
class battery {
```

```

attributes
  Real V;
constructors battery(V1) {
  V = V1;
}
}

class connect {
  attributes
    battery B; component CC;
  constraints
    B.V = CC.V ;
  constructors connect(B1, C1) {
    B = B1; CC = C1 ;
  }
}

class samplecircuit {
  attributes
    battery B;
    connect C;
    component R1, R2, R3, R4, P1, P2, S;
    component[] R12, R34, P12;
  constructors samplecircuit(X) {
    R1 = new component(V1, I1, 10);
    R2 = new component(V2, I2, 20);
    R3 = new component(V3, I3, 20);
    R4 = new component(V4, I4, 20);
    R12[1] = R1;
    R12[2] = R2;
    R34[1] = R3;
    R34[2] = R4;
    P1 = new parallel(R12);
    P2 = new parallel(R34);
    P12[1] = P1;
    P12[2] = P2;
    S = new series(P12);
    B = new battery(30);
    C = new connect(B, S);
    dump([R1, R2, R3, R4]);
  }
}

```

Shown below is a script of the compilation and running of the above program.

```

tambay@kulta>cob circuit.cob circuit.clpr -e ckt
% restoring /projects/tambay/CobToSicstus/cobcompiler...
...
yes
...
% consulting /projects/tambay/temp/compiler/circuit.clpr...
....
% /projects/tambay/temp/compiler/ckt.sav created in 110 msec
yes

tambay@kulta>ckt
% restoring /projects/tambay/temp/compiler/ckt...
...
| ?- samplecircuit(_, _).
R1 = [12.0,1.2,1.0E+01]
R2 = [12.0,0.6,2.0E+01]
R3 = [18.0,0.8999999999999999,2.0E+01]
R4 = [18.0,0.8999999999999999,2.0E+01]
yes
| ?-

```

Error Messages and Warnings. The cob compiler will point out a syntax error by naming the class in which it occurs, the attribute/constraint/constructor definition in which it occurs and the index of the constraint/attribute declaration. If the error is in the i^{th} constraint, it means, it is in the i th semi-colon separated constraint. Line number of error is not given. Undeclared variables will not be caught unless the selection operation (.) is being performed on them. If the compilation hangs, please send e-mail with the uncompileable cob program to **tambay@cse.buffalo.edu**

If the executable is being formed along with compilation, then errors (if present) will be detected by the prolog interpreter. These should be removed by correcting the cob program and re-compiling it.

If the clp translation of a cob program is loaded within prolog manually, then there may be a series of warnings of singleton variables. In most cases, these can safely be ignored. However, errors shown during this compilation must not be ignored. They should be removed by correcting the cob program and re-compiling it.

Printing. Two functions are provided for printing the values of attributes to standard output.

1. **dump:** To print the value of a variable, use the built-in Cob predicate `dump/1`. If A,B,C are Cob program variables with values 1,2 and unknown respectively, and X is an undeclared variable, then `dump([A,B,C,X])` will print
A = 1
B = 2
C = `_some_internal_name`
X = `_some_internal_name`
2. **print:** To print a string or just the value of a variable, use `print/1`. `print('Sample String')` will print `Sample String` on standard output.

Type checking. Currently type checking is performed only on the variables on which the select/access “.” operation is performed. This type inference is done at run-time.

Tracing. The translated CLP program can be traced by using Prolog’s `trace` command. Once in trace mode, the normal debugging commands of Prolog can be used to trace the program.

Using underscore. The underscore character (`_`) can be given as argument to constructors or predicates. The programmer should get familiar with its use Prolog before using it in Cob.

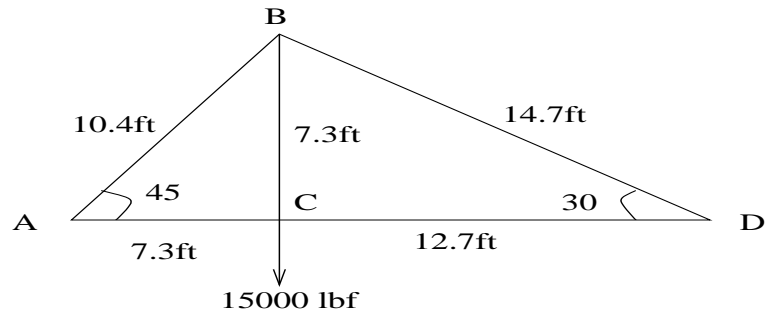


Figure 8.1: A Simple Truss

Appendix C: Sample Truss Code

The truss in Figure 8.1 can be modeled in Cob by the class `sampletruss` given below. The Cob model calculates the cross sectional area of the beams given that they have a square cross section, are made of steel and that the truss must support a load of 15000 lbf.

```
class sampletruss {
  attributes
    beam AB, BC, CD, BD, AC; load IAV, IAH, ICV, IDV;
    bar IAB, IAC, IBA, IBC, IBD, ICA, ICB, ICD, IDB, IDC;
    bar [] Ba, Bb, Bc, Bd; load [] La, Lc, Ld;
    real W1, W2, W3, W4, W5, H1, H2, H3, H4, H5, Fab, Fbc, Fcd,
    Fbd, Fac, Fab_bk, Fbc_bk, Fcd_bk, Fbd_bk, Fac_bk, Fab_bn, Fbc_bn,
    Fcd_bn, Fbd_bn, Fac_bn, Fab_t, Fbc_t, Fcd_t, Fbd_t, Fac_t, Fdv,
    Fcv, Fav, Fah, Sigab, Sigbc, Sigcd, Sigac, Sigbd,
    Iab, Ibc, Icd, Ibd, Iac, Es, Sy, Pi;
    joint JA, JB, JC, JD;
  constraints
    Es = 30000000; Sy = 30000; Pi = 3.141;
    W1 = H1; W2 = H2; W3 = H3; W4 = H4; W5 = H5;
  constructors sampletruss() {
    AB=new beam(Es,Sy,10.4,W1,H1,Fab_bn,Fab_bk,Fab_t,Sigab,Iab,Fab);
    BC=new beam(Es,Sy,7.3,W2,H2,Fbc_bn,Fbc_bk,Fbc_t,Sigbc,Ibc,Fbc);
```

```

CD=new beam(Es,Sy,12.7,W3,H3,Fcd_bn,Fcd_bk,Fcd_t,Sigcd,Icd,Fcd);
BD=new beam(Es,Sy,14.7,W4,H4,Fbd_bn,Fbd_bk,Fbd_t,Sigbd,Ibd,Fbd);
AC=new beam(Es,Sy,7.3,W5,H5,Fac_bn,Fac_bk,Fac_t,Sigac,Iac,Fac);
IAB = new bar(AB,Pi/4); IAC = new bar(AC,0);
IAV = new load(Fav,Pi/2); IAH = new load(Fah,0);
Ba = [IAB, IAC]; La = [IAV, IAH]; JA = new joint(Ba,La);
IBA = new bar(AB, 5*Pi/4); IBC = new bar(BC, 3*Pi/2);
IBD = new bar(BD, 11*Pi/6); Bb = [IBA, IBC, IBD]; Lb = [];
JB = new joint(Bb, Lb); ICA = new bar(AC, Pi);
ICB = new bar(BC, Pi/2); ICD = new bar(CD, 0);
ICV = new load(15000, 3*Pi/2); Bc = [ICA, ICB, ICD];
Lc = [ICV]; JC = new joint(Bc, Lc); IDB = new bar(BD, 5*Pi/6);
IDC = new bar(CD, Pi); IDV = new load(Fdv, Pi/2);
Bd = [IDB, IDC]; Ld = [IDV]; JD = new joint(Bd, Ld);
}
}

```

Appendix D: Cob Model of Heatplate using Inheritance

We give below a reformulation of the Cob model of a Heatplate that uses inheritance. We define a `cell` class that aggregates four cells (its neighbors) and has an attribute (`T`) representing its temperature. There are no constraints in this class and it is used to represent the cells along the border of a heatplate. We define a subclass of such a `cell` called `inner_cell` which inherits the attributes of `cell` and in addition has a constraint stating that its temperature is the average of the temperatures of its neighbors. An instance of `inner_cell` represents the interior points of a heatplate. We give below the Cob code for the above classes and the `heatplate` class which represents the creation and initialization of a heatplate. This formulation may be compared with the Cob model given in Section 5.2.2. The former uses inheritance to model the difference in behavior of a border cell and an interior cell, while the latter uses conditional constraints for this purpose.

```

class cell {
  attributes
    cell Left, Right, Up, Down;
    real T;
  constructor cell(T1) {
    T = T1;
  }
}
class inner_cell {
  constraints
    T = (Left.T+Right.T+Up.T+Down.T)/4;
  constructor inner_cell(T1) {
    T = T1; ;
  }
}
class heatplate {
  attributes
    int Size;
    cell [Size][Size] Plate;
  constraints
    forall I in 2..Size-1:
      (forall J in 2..Size-1:
        (Plate[I,J].Up = Plate[I-1,J];
         Plate[I,J].Right = Plate[I,J+1];
         Plate[I,J].Left = Plate[I,J-1];
         Plate[I,J].Down = Plate[I+1,J]);));
  constructors heatplate(S, A,B,C,D) {
    Size = S;
    forall I in 2..Size-1:
      forall J in 2..Size-1:
        (Plate[I,J] = new inner_cell(_));
    forall K in 1..Size:
      (Plate[1,K] = new cell(A);
       Plate[Size,K] = new cell(B));
    forall L in 2..Size-1:
      (Plate[L,1] = new cell(C);
       Plate[L,Size] = new cell(D));
  }
}

```