AN ABSTRACT OF THE DISSERTATION OF

Nabil M. Zamel for the degree of Doctor of Philosophy in Computer Science presented on December 6, 1994.

Title: ELECTRA: Integrating Constraints, Condition-Based Dispatching, and Feature Exclusion into the Multiparadigm Language Leda

Abstract approved: _____

Timothy A. Budd

Multiparadigm languages are languages that are designed to support more than one style of programming. Leda is a strongly-typed multiparadigm programming language that supports imperative, functional, object-oriented, and logic programming. The constraint programming paradigm is a declarative style of programming where the programmer is able to state relationships among some entities and expect the system to maintain the validity of these relationships throughout program execution. The system accomplishes this either by invoking user-defined fixes that impose rigid rules governing the evolution of the entities, or by finding suitable values to be assigned to the constrained entities without violating any active constraint. Constraints, due to their declarative semantics, are suitable for the direct mapping of the characteristics of a number of mechanisms including: consistency checks, constraint-directed search, and constraint-enforced reevaluation, among others. This makes constraint languages the most appropriate languages for the implementation of a large number of applications such as scheduling, planning, resource allocation, simulation, and graphical user interfaces.

The semantics of constraints cannot be easily emulated by other constructs in the paradigms that are offered by the language Leda. However, the constraint paradigm does not provide any general control constructs. The lack of general control constructs impedes this paradigm's ability to naturally express a large number of problems. This dissertation presents the language Electra, which integrates the

constraint paradigm into the language Leda by creating a unified construct that provides the ability to express the conventional semantics of constraints with some extensions. Due to the flexibility of this construct, the programmer is given the choice of either stating how a constraint is to be satisfied or delegating that task to the constraint-satisfier. The concept of providing the programmer with the ability to express system-maintained relations, which is the basic characteristic of constraints, provided a motivation for enhancing other paradigms with similar abilities. The functional paradigm is extended by adding to it the mechanism of condition-based dispatching which is similar to argument pattern-matching. The object-oriented paradigm is extended by allowing feature exclusion which is a form of inheritance exception. This dissertation claims that the integration provided by the language Electra will enable Leda programmers to reap the benefits of the paradigm of constraints while overcoming its limitations.

ELECTRA: Integrating Constraints, Condition-Based Dispatching, and Feature Exclusion into the Multiparadigm Language Leda

by

Nabil M. Zamel

A DISSERTATION

submitted to

Oregon State University

in partial fulfillment of the

requirements for the degree of

Doctor of Philosophy

Completed December 6, 1994

Commencement June 1995

Doctor of Philosophy dissertation of Nabil M. Zamel presented on December 6, 1994

APPROVED:

Redacted for Privacy

Major Professor, representing Computer Science

Redacted for Privacy

Chair of Computer Science Department

Redacted for Privacy

Dean of Graduate School

I understand that my dissertation will become part of the permanent collection of Oregon State University libraries. My signature below authorizes release of my dissertation to any reader upon request.

Redacted for Privacy

Nabil M. Zamel, Author

This work is dedicated to
my father Mohammed, my mother Hussah,
and my uncle Ahmed.

Their love, support, and encouragement are what have guided me all along,
without which I would not have gotten this far.

ACKNOWLEDGEMENT

I am especially grateful to my major professor Dr. Timothy Budd for his support, encouragement, and never ending patience. I deeply appreciate his great consideration and full understanding towards my visual disability. I would also like to express my appreciation to the other members of my Ph.D. committee, Dr. Bella Bose, Dr. Curtis Cook, Dr. Sheila Cordray, and Dr. Bruce D'Ambrosio for their helpful comments, encouragement, and constructive criticisms. I am also grateful to Dr. Alan Borning (University of Washington) and Dr. Bjorn Freeman-Benson (Carlton University, Canada).

I owe a tremendous debt to Timothy P. Justice and Rajeev K. Pandey, my colleagues and friends for their willingness to act as sounding boards and for their helpful comments on earlier drafts of this dissertation. Working with them has been more than fun. Timothy P. Justice provided numerous suggestions that helped me to clarify the ideas presented in this dissertation.

I would also like to thank Dr. Hussein Almuallim who taught me how to do research, Masami Takikawa who taught me a lot about constraints, and Dr. Leine Stuart who has been a teacher, an advisor, and a good friend throughout my higher education.

A very special thanks must be given to Naomi Hiramoto and Ebrahim Buzaboon who created a perfect, convenient, and comfortable environment, the kind a Ph.D. student can only dream of. Thank you two for your care and encouragement.

My graduate study was supported by a generous scholarship from ARAMCO Services Company. I am grateful for this support and would like to express my deep gratitude to all of my ARAMCO academic advisors.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ELECTRA: Integrating Constraints, Condition-Based Dispatching, and Feature Exclusion into the Multiparadigm Language Leda

Chapter 1
Introduction

The word "paradigm" was brought into modern usage by the historian of science Thomas Kuhn in his influential book *The Structure of Scientific Revolutions* [Kuhn, 1970]. He employed the term to represent a way of organizing knowledge, or a way of viewing the world. Robert Floyd introduced the term to the field of computer science in his 1978 ACM Turing Award lecture [Floyd, 1979]. He used the term to mean a model or an example, where a programming paradigm can be understood as a way of conceptualizing what it means to perform computation [Budd, 1995]. Since then the term has been used in many disciplines within the field of computer science. In problem solving and algorithms, for example, the term is used to refer to problem solving approaches such as divide-and-conquer, hill climbing, and dynamic programming [Harel, 1992]. The parallel computing researchers associate the term with models of parallelism such as data parallel, message passing, and shared variable [Hwang, 1993]. Many researchers in artificial intelligence view connectionism as a paradigm in AI research [Addanki, 1987]. The term "paradigm" has also been used in other disciplines such as database, operating systems, and software engineering to imply different meanings and connotations.

In the context of programming languages, the term "paradigm" has been defined by Budd [Budd, 1995] as a world view or a way of conceptualizing a computation. Appleby [Appleby, 1991] defined it as a simple collection of abstract features that categorize a group of languages. Bal and Grune [Bal and Grune, 1994] view a paradigm as a cohesive set of methods for solving a certain class of problems. Placer [Placer, 1988] states the following regarding the definition of the term "paradigm":

> A programming paradigm represents a way of thinking about a problem, a way of modeling a problem domain. The ontologies represented by different paradigms are different. For example, the logic paradigm views the world as composed of predicates and relations while the functional paradigm models a world of functions, function composition and function application.

Therefore, within the context of programming languages, a programming paradigm is a characteristic of a programming language that specifies how the language views its world and classifies the set of problems that can be naturally and easily expressed by the language.

1.1 Programming Language Paradigms

The paradigms of programming languages can be divided into two classes of paradigms: imperative paradigms and declarative paradigms [Ghezzi and Jazayeri, 1987, Wilson and Clark, 1988]. The imperative paradigms assume that the programmer will state how the the problem is to be solved as part of the specification of the problem, whereas the declarative paradigms only need to know what is to be done without any information regarding how the goal is to be achieved. The terms imperative and declarative designate the two extreme ends of a continuum where languages fall somewhere between the two extremes. These two classes can be subdivided into more specific subclasses. The following subsections present some of the most well-studied programming paradigms.

1.1.1 The Imperative-Procedural Programming Paradigm

The imperative programming paradigm views its world as variables and values. A variable is a named memory cell in which a value is stored. This paradigm facilitates computation by means of state changes, where the programmer states exactly what is to be done to every variable. It is sometimes referred to as a *statement-oriented* paradigm [Ghezzi and Jazayeri, 1987] due to the dominant role played by imperative

statements such as the assignment statement and other sequencing, repetition, and conditional statements. The computer acts as a data manager that alters variables following the directions dictated by the statements of the program. The desired result of a program is achieved by combining the changes done by the individual statements. This paradigm contains languages such as FORTRAN [Ellis, 1982], COBOL [Welland, 1983], and Algol 60 [Andersen, 1964].

1.1.2 The Object-Oriented Programming Paradigm

The world of this paradigm is composed of a set of independent objects. Each object is responsible for its behavior and structure. Objects can collaborate with each other by sending or receiving messages from one object to another. A message is a request to perform certain actions or to provide certain information. Each object has its own memory and can contain other objects. Classes are used to group objects. Each object is an instance of some class. A class acts as a repository for the behavior of the objects it represents. Inheritance is a mechanism via which one can organize classes in a hierarchical fashion such that a subclass inherits the behavior and structure of all its superclasses in addition to its own unique characteristics [Budd, 1991b, Shriver and Wegner, 1988, Kim and Lochovsky, 1989]. Languages that belong to this paradigm include: Simula [Birtwistle *et al.*, 1975], Smalltalk [Goldberg and Robson, 1983] and Beta [Madsen *et al.*, 1993].

1.1.3 The Functional Programming Paradigm

The functional programming paradigm views a drastically different world from the ones viewed by the previous two paradigms. In this paradigm the world is composed of a set of functions and computation is accomplished by function application and function composition. One major difference between this paradigm and the above two is that computation is not done by incremental modification to the global state, but rather by transformation. This means that any change to a value transforms

it into a new value which is independent of the original one. This paradigm treats functions as first class data items. This means that functions can be assigned to identifiers, and can be passed as arguments to other functions or returned as results of other functions [Henderson, 1980, Reade, 1989]. This approach towards programming inspired a number of language designers to design languages that are functional in nature such as: Standard ML [Milner *et al.*, 1990], Haskel [Hudak and Wadler, 1988], and Scheme [Abelson *et al.*, 1985, Steele Jr. and Sussman, 1975].

1.1.4 The Logic Programming Paradigm

The idea of using the logic programming approach grew out of research in some disciplines in artificial intelligence such as automatic theorem proving and natural language processing. This idea was a significant contribution because, until about 1970, logic was used only as a specification language. However, the work by Kowalski [Kowalski, 1974] showed that logic has a procedural interpretation as well, making it possible in principle to use logic as a programming language. The approach taken by logic programming is to decompose a program into two components: a logic component and a control component as explained by Kowalski [Kowalski, 1979]. The logic component is provided by the programmer and is composed of a set of facts, a collection of rules, and a query. The programmer does not need to specify how the query is to be answered. The process of figuring out how to answer the query using the given facts and rules is the job of the control component. The control component is part of the underlying run-time system and is built as some searching mechanism. This separation between logic and control makes the logic programming paradigm a declarative one since the programmer needs only to state what the problem is, and not how it is to be solved. The first and most popular logic programming language is Prolog [Clocksin and Mellish, 1987].

Other researchers such as Ambler, Burnett, and Zimmerman [Ambler *et al.*, 1992], Bal and Grune [Bal and Grune, 1994], Budd [Budd, 1995], Hailpern [Hailpern,

1986], and Placer [Placer, 1991a] identify additional paradigms including access-oriented, applicative, constraint, data-structure oriented, parallel, procedural, real-time, relational, rule-oriented, and the visual programming paradigm.

1.2 Multiparadigm Languages

Theoretically, the solving power of all general purpose programming languages is equal regardless of the paradigms these languages represent. This is due to the assumption that these languages are Turing equivalent. This means that if a problem can be solved using a language of a particular paradigm, then it can also be solved using another language that belongs to a different paradigm. However, even though languages are equal in terms of their power, they differ in their suitability for directly expressing the variety of problem domains as pointed out by Maclennan [MacLennan, 1987], "although it's possible to write any program in any programming language, it's not equally easy to do so." Budd [Budd, 1995] states the following regarding the influence of the characteristics of a particular programming language over the approaches used for solving problems in that language:

> What is true for natural human languages is even more true in the realm of artificial computer languages. That is, the language in which a programmer thinks a problem will be solved will color and alter, in a basic fundamental way, the fashion in which an algorithm is developed.

The recognition that there is a strong influence between the characteristics of the paradigms and their suitability for expressing differing problems led programming language researchers to propose the development of programming languages that comprise several programming paradigms. Their justification is that a single programming paradigm is inadequate to achieve a direct and natural expression of all aspects of complex problems [Budd, 1991a, Hailpern, 1986, Placer, 1991b]. The objectives of the multiparadigm language research is summarized in the following quote by Budd [Budd, 1995]:

> The basic tenet of multiparadigm programming is that the programmer should not be forced to solve all problems in a single style; instead, the

programmer should be free to select the programming tools for a problem that best match the nature of the task being performed.

1.3 The Leda Programming Language

Leda is a strongly-typed compiled programming language that supports the imperative, object-oriented, functional, and logic programming paradigms [Budd, 1995]. The objective of the design of Leda is "to provide the programmer with a system with which programs cannot only be developed in a number of different styles, but also one in which algorithms can be expressed in combination of styles" [Budd, 1989]. The evolution of Leda has progressed over a period of five years, resulting in improving both the syntax and semantics of the language. These improvements can be observed by comparing the examples presented in this dissertation and in [Budd, 1995] to those that appeared in previous publications such as [Budd, 1991a, Budd, 1992, Zamel and Budd, 1993]. This evolution also resulted in the production of three implementations of the language [Pesch and Shur, 1991, Pandey *et al.*, 1993, Budd, 1995]. Leda continues to develop, evolve, and grow. This growth is reflected by the current ongoing research related to Leda which includes Pandey's development of a Leda programming environment [Pandey, 1993], Budd, Justice, and Pandey's investigation of the suitability of multiparadigm languages for compiler construction [Justice *et al.*, 1993, Justice *et al.*, 1994], and Justice's study on the influence of the characteristics of multiparadigm languages on the design and implementation of complex applications such as compiler constructions tools [Justice, 1995].

1.4 The Constraint Paradigm

The constraint programming paradigm is a declarative style of programming where the programmer is able to state relationships among some entities and expect the system to maintain the validity of these relationships throughout program execution [Leler, 1988]. The system accomplishes that either by invoking user-defined

fixes that impose rigid rules governing the evolution of the entities, or by finding suitable values to be assigned to the constrained entities without violating any active constraint. Constraints, due to their declarative semantics, are suitable for the direct mapping of the characteristics of a number of mechanisms such as: consistency checks, constraint-directed search, and constraint-enforced reevaluation, among others [Gore, 1990]. This makes constraint languages the most appropriate languages for the implementation of a large number of applications such as scheduling [Fox, 1983, Dincbas *et al.*, 1988], planning [Boizumault *et al.*, 1993, Charman, 1993], resource allocation [Prosser *et al.*, 1992], simulation [Borning, 1981, Duisberg, 1986] , and graphical user interfaces [Myers, 1992, Maloney *et al.*, 1989]. The semantics of constraints cannot be easily emulated by other constructs in the paradigms that are offered by the language Leda. However, the constraint paradigm does not provide any general control constructs. The lack of general control constructs impedes this paradigm's ability to naturally express a large number of problems.

1.5 Thesis Objectives

The objectives of our thesis are to integrate the constraint paradigm into Leda and to enhance the characteristics of some of the paradigms that are provided by Leda. The objective of integrating constraints into Leda is summarized in the following points:

- To define a simple yet expressive syntactic construct called *constraint* that will aid programmers in stating system-maintained relations. This construct will be a vehicle for the integration of the constraint paradigm into the language Leda.

- To provide Leda programmers with the choice of either stating how a constraint is to be satisfied or delegating that task to an underlying constraint satisfier.

- To make it possible for Leda programmers to express required as well as preferential constraints and to provide them with the ability to define a hierarchy of preferences.

- To create a mechanism for separating the process of satisfying constraints from the Leda compiler. This means that the constraint satisfier will be a separate and independent module. This approach should make it possible for Leda programmers to take advantage of more powerful solvers once they become available by plugging them into the Leda run-time system.

The idea of providing the programmer with the ability to express system-maintained relations, which is the basic characteristic of constraints, motivated us to enhance other paradigms with similar abilities. The enhancement objective can be expressed as follows:

- To enhance the functional programming paradigm with the ability to express condition-based dispatching. Condition-based dispatching extends Leda's function dispatching process by allowing the programmer to add conditions to a function's signature in such a way that dispatching to that function will take place only if those conditions are met. This gives Leda programmers the ability to simulate the mechanism of argument pattern-matching which is provided by most functional programming languages.

- To enhance the object-oriented paradigm by giving the programmer the ability to exclude features of a superclass from appearing in a subclass. The main advantage of this capability is that it makes it possible for the programmer to express inheritance exceptions. This is helpful because it is not always possible to impose a rigid hierarchical structure on every real world situation.

These extensions and enhancements to the Leda language produce a programming language that we call *Electra*. We believe that these extensions and enhancements that are provided by the language Electra will enable programmers to reap the benefits of the paradigm of constraints while overcoming its limitations.

1.6 Previous Work

Research related to constraints spans a number of fields including, artificial intelligence, operations research, and engineering. In the field of computer science, the research includes the utilization of constraints in many applications such as: simulation, manipulation of geometric layouts, graphical user interfaces, and others. This section concentrates its coverage on research that is related to programming languages. More extensive presentations and further references can be found in [Freeman-Benson *et al.*, 1990a, Freeman-Benson, 1991], and [Leler, 1988].

1.6.1 Constraints and Imperative Paradigms

The concept of viewing a constraint as a software component that can be integrated into some system or programming language started with Ivan Sutherland's interactive graphical system Sketchpad [Sutherland, 1963a, Sutherland, 1963b]. This system is one of the first interactive graphical interfaces. It was designed to solve geometric constraints, where relations in the specifications of a problem are expressed as equations or tables. It utilizes constraint propagation and relaxation to solve its constraint networks. Lauriere [Lauriere, 1978] developed a system called ALICE which deals with constraints over finite-domains. Other pioneering efforts include Steels's Language CONSTRAINTS [Steele, 1980, Sussman and Steele, 1980] and ThingLab [Borning, 1979] which will be covered in the following paragraphs.

ThingLab is a constraint-based laboratory that allows a user to construct simulations of electrical circuits, mechanical linkages, demonstrations of geometric theorems, and graphical calculators using interactive direct manipulation techniques. It was built on top of Smalltalk-80 [Goldberg and Robson, 1983]. ThingLab solves its constraints by propagating degrees of freedom and relaxation. Borning enhanced ThingLab by adding constraint hierarchies [Borning *et al.*, 1987, Borning *et al.*, 1988, Borning *et al.*, 1989] Further improvements to ThingLab resulted in the creation of

its successor ThingLab II [Maloney *et al.*, 1989] which adds an incremental planning algorithm that helps in eliminating the need for recreating the solution plans every time a constraint is added or removed [Freeman-Benson and Wilson, 1990, Freeman-Benson *et al.*, 1990a, Freeman-Benson *et al.*, 1992]. It is oriented towards building user interfaces [Maloney *et al.*, 1989]. The Animus system [Duisberg, 1986] is an animation system that added temporal constraints to ThingLab.

The constraint imperative paradigm (CIP), which is investigated by Borning and his group at the University of Washington, aims at integrating constraints with other imperative paradigms. An instance of this paradigm is the language Kaleidoscope'90 [Freeman-Benson, 1990, Freeman-Benson and Borning, 1991] and its successors Kaleidoscope'91 [Freeman-Benson and Borning, 1992] and Kaleidoscope'93 [Lopez *et al.*, 1994b, Lopez *et al.*, 1994a]. Kaleidoscope allows the programmer to declare required as well as preferential constraints. Variables can be annotated as read-only or write-only in the statement of the constraint. This helps in blocking automatic changes to variables during the constraint satisfaction process.

Leler [Leler, 1986, Leler, 1988] designed a language called Bertrand which is a constraint language that is based on augmented term rewriting. The language Siri [Horn, 1992a, Horn, 1992b] combines constraints with object-oriented programming. It uses a graph rewriting model of execution, which is an extension to that of Bertrand.

1.6.2 Constraints and Declarative Paradigms

Logic Programming is considered the most natural programming paradigm for combining with constraints. In logic programming languages such as Prolog, rules are stated using the following syntax:

$$p(t) \text{ :- } q_1(t), \ldots, q_m(t).$$

where $p, q_1, \ldots, q_m$ are predicates, and t denotes a list of terms. Jaffar and Lassez introduced the idea of creating a general scheme for extending logic programming to in-

clude constraints [Jaffar and Lassez, 1987]. They called this scheme *Constraint Logic Programming* (CLP). This scheme represents a family of languages called CLP($\mathcal{D}$), where the parameter $\mathcal{D}$ is the domain of the constraints. In a CLP language, rules have the following syntax:

$$p(t) \text{ :- } q_1(t), \ldots, q_m(t), c_1(t), \ldots, c_n(t).$$

where $p, q_1, \ldots, q_m, t$ have the same meaning as above and $c_1, \ldots, c_n$ are constraints over the domain $\mathcal{D}$. In this scheme the predicates $p, q_1, \ldots, q_m$ are called control predicates and the constraints $c_1, \ldots, c_n$ are labeled constraint predicates. The difference between control predicates and constraint predicates is that constraint predicates are not allowed to have defining clauses (i.e. are not allowed to appear in the left hand side of a clause) because it is assumed that their meaning is known and cannot be altered.

Kowalski [Kowalski, 1979] represented logic programming by the following equation:

$$Program = Logic + Control.$$

Constraint Logic Programming can be viewed as:

$$Program = Logic + Control + Constraints.$$

The execution of a CLP program can be summarized as follows:

- The Prolog part of the program will execute in the usual fashion.
- As execution proceeds, constraints are accumulated on logic variables.
- Backtracking will occur if unification violates any constraint.
- If it is possible to instantiate all the unknown variables without violating any constraint, then the program terminates, outputting the values found for the unknown variables. If it is not possible to bind every unknown variable, then the output will include a listing of the remaining constraints on the unbound variables.

Several instances of the CLP($\mathcal{D}$) scheme have been implemented, including CLP($\mathcal{R}$) [Heintze *et al.*, 1992, Jaffar *et al.*, 1992], CLP(Σ^*) [Walinsky, 1989], CHIP [Van Hentenryck, 1991], Prolog III [Colmerauer, 1990], and CAL [Satoh and Aiba, 1991].

Wilson and Borning extended the scheme of Constraint Logic Programming by adding a strength level to each constraint [Wilson and Borning, 1993]. They call this scheme *Hierarchical Constraint Logic Programming* (HCLP).

Saraswat proposed a generalization of the CLP scheme by including concurrency, which resulted in the creation of the cc family of languages [Saraswat, 1993]. A cc program is decomposed into a set of elements called agents. These agents communicate via the use of a shared area called a constraint store. The basic actions of agents are telling and asking the constraints. A tell action adds a constraint to the store. If the store remains consistent then the tell operation returns success, otherwise it fails. The operation ask is used to check with the constraint store if a certain constraint is consistent with all other constraints. An example of a cc language is Janus [Saraswat *et al.*, 1990], which is a distributed programming language.

1.6.3 The Electra Difference

The approach taken in the design of Electra differs from the above approaches in the following aspects:

- Electra is an extension to the language Leda, which is designed as a multiparadigm language incorporating the four major paradigms: imperative, object-oriented, functional, and logic. This means that Electra's constraints can interact with more than a single paradigm.

- Constraints in Electra give the programmer the choice of either stating how a constraint is to be solved or delegating that task to an underlying constraint satisfier.

- The design of Electra is built on the concept of separating the constraint satisfaction process from the compiler. This makes it possible to take advantage of any new satisfier.

1.7 Outline of the Dissertation

- Chapter 2 covers the fundamental concepts of constraints including their definition, advantages, and approaches for satisfying them.
- Chapter 3 presents the syntax, semantics, and general characteristics of Electra's constraint constructs.
- Chapter 4 provides a description of condition-based dispatching and how it can be expressed using guarded functions.
- Chapter 5 discusses feature exclusions and shows why it is necessary to be able to express this phenomenon.
- Chapter 6 presents the approaches taken for the implementation of Electra's enhancements and its compiler.
- Chapter 7 lists some advantages of the added constructs and provides some illustrative examples.
- Chapter 8 summarizes the dissertation and presents some directions for further research.
- Appendix A gives a brief introduction to the DeltaBlue constraint solver.
- Appendix B lists the BNF grammar of the Electra language.

Chapter 2
Fundamental Concepts of Constraints

This chapter defines what constraints are and presents their role as programming tools. It also describes the concept of constraint-satisfaction problems and shows how some problems can easily be formulated as constraint-satisfaction problems by decomposing them into a set of constraints. Finally, it reviews the approaches that have been taken to build constraint-satisfaction systems.

2.1 What is a Constraint?

A constraint is an expression that states a relation among a set of variables. The validity of this relation is maintained by the *constraint-satisfaction system* throughout the evolution of the related variables. In addition, one can request that the system find appropriate values for the unknown variables in such a way that none of the relations are violated. One can view constraints as statements or expressions that have the objective of narrowing down the domains of the constrained entities. The history of constraint languages goes as far back as 1963 when Ivan Sutherland created his constraint graphical system Sketchpad [Sutherland, 1963a, Sutherland, 1963b]. After Sutherland paved the way, many other approaches were proposed [Leler, 1986, Leler, 1988]. The most popular example used to illustrate the behavior of constraints is the equation used in the conversion between Fahrenheit and Celsius temperatures. Two variables **C** and **F** maintain temperature such that the variable **C** measures temperature in Celsius and the variable **F** measures it in Fahrenheit. The two variables are linked so that they measure the same temperature. A change in either variable is automatically reflected by a change in the other. Figure 2.1 gives a graphical

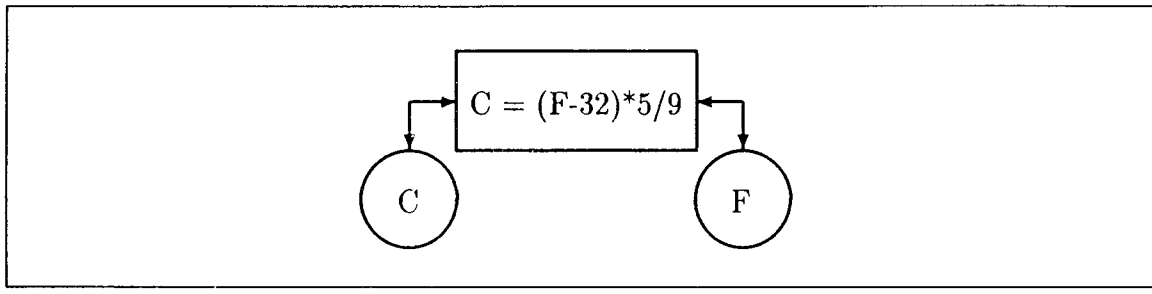

Figure 2.1. Fahrenheit and Celsius.

representation of the relation between **C** and **F**. Notice that each of the entities can behave as input as well as output. It is the job of the *constraint-satisfier* to figure out which entity is known, and find the value of the unknown.

In programming languages, constraints appear as constructs under different names and with a variety of functionalities. In terms of functionality, Gore [Gore, 1990] distinguishes the following three categories:

- Consistency Checks.
- Constraint-directed Search.
- Constraint-driven Reevaluation.

The following subsections present a description of each of the categories listed above.

2.1.1 The Role of Constraints as Consistency Checks

As consistency checks, constraints are used as guards rejecting any modification that violates any of the stated relations. An example of such consistency checks are the type checking rules of any strongly typed language. A language that provides explicit constructs for expressing consistency checks is Eiffel [Meyer, 1988, Meyer, 1993]. In Eiffel, consistency checks are done via class invariants. When a constraint that functions as a consistency check is broken, the underlying constraint checker

issues an error message and does not try to invoke any fixes. The primary usage of these kinds of constraints is as debugging tools that trap errors early on during the development phase.

2.1.2 The Role of Constraints in Guiding a Search

As tools for constraint-directed search, constraints are used to prune search trees, eliminating all unnecessary branches. Using constraints for such functionality is usually done as part of building Artificial Intelligence problem solvers. The problem solver takes as input a set of rules, a set of facts, and one or more goals. Its objective is to find a chain of rule applications that link the facts to the goals. The problem is that as the number of rules increases, the search tree becomes more bushy, assuming that a rule can be applied at any time and for one or more times. Thus, the search becomes impractical even for a small number of rules. To remedy the situation, the problem solver must avoid all useless branches of the search tree. Constraints have been used as guides to assist the problem solver in achieving this goal. ALICE [Lauriere, 1978] is an example of a system that is built around this idea.

2.1.3 Constraint-Driven Reevaluation

The evolution of a constrained entity might result in breaking some of the constraints imposed upon that entity. If a broken constraint is intended to function as a consistency check then, as stated earlier, the constraint satisfier will return an error message and terminate the execution. However, if the broken constraint is intended to motivate reevaluation then a constraint fix will be invoked. The job of the constraint fix is to bring the system back to a state of balance where all the constraints are satisfied. As will be detailed later, constraints in this category are normally implemented as attached procedures.

2.2 Constraint-Satisfaction Problems

Constraint-satisfaction is a problem solving methodology in which the objective is to find globally consistent assignments of values for variables subject to a set of constraints. A problem that can be naturally solved using this technique is called a *constraint-satisfaction problem* (CSP). This class of problems is also known as *Consistent Labeling Problems* (CLP).

A constraint-satisfaction problem can be described as a 3-tuple (V, D, C). Where V is a set of variables $V_1, V_2, \ldots, V_n$, D is a set of associated domains $D_1, D_2, \ldots, D_n$ and C is a set of boolean functions $C_1, C_2, \ldots, C_n$. These functions are called constraints. Each constraint C_i may be viewed as a predicate over a subset of V. A predicate that returns true is said to be satisfied. A solution to a constraint-satisfaction problem requires a search for a set of assignments $(V_1 = v_1, V_2 = v_2, \ldots, V_n = v_n)$, where $v_1 \in D_1, v_2 \in D_2, \ldots, v_n \in D_n$ in such a manner that all the constraints are satisfied. The statement of the problem might ask for a single solution, or it might require a search for the set of all possible solutions, or it might simply be an inquiry of whether the set of solutions is empty or not. If the set of solutions is empty then the constraint-satisfaction problem is said to be unsatisfiable.

In the literature, it is customary to represent a network of binary constraints by a graph called a *constraint graph*, where the variables are represented by nodes and the constraints by arcs. In this presentation, both variables and constraints will be viewed as nodes in the constraint graph, where the former will be displayed as a circles and the latter as a rectangles. One of the first comprehensive description of the formal aspects of constraint-satisfaction problems was presented by Montanari [Montanari, 1974]. Tsang [Tsang, 1993] gave a rigorous treatment of the topic with an extensive study of the algorithms that are used to solve constraint-satisfaction problems.

An example of a problem that can be stated as a constraint-satisfaction problem is the map-coloring problem. The objective of the problem is to color each region of a given map such that no two adjacent regions have the same color. Figure 2.2

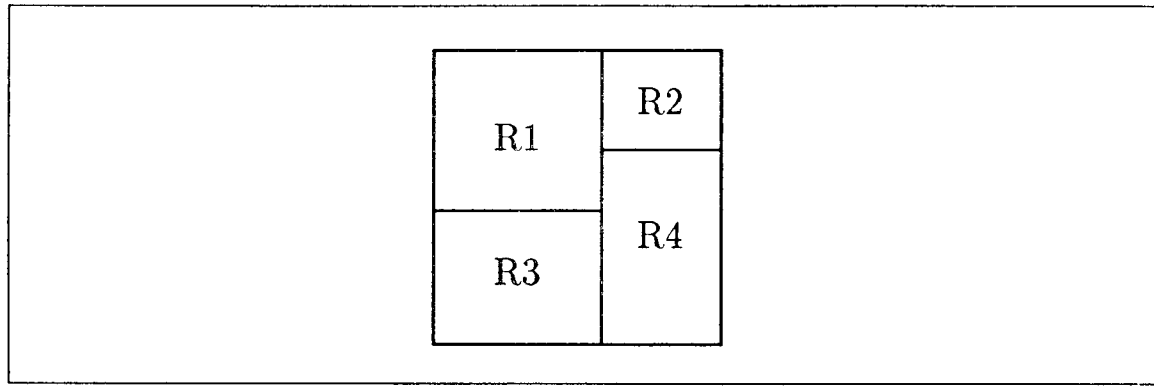

Figure 2.2. An Example Map-Coloring Problem.

shows an example of a map coloring problem. The four regions $R1$, $R2$, $R3$, and $R4$ are to be colored in such a way that the color of Ri is not the same as the color of Rj if Ri and Rj are adjacent regions. If we assume that the value of Ri for $1 \leq i \leq 4$ is taken from a set of colors then the constraint for this problem can be expressed as $Ri \neq Rj$. Figure 2.3 depicts the mapping of the map-coloring problem into a constraint-satisfaction problem formulation. It shows the decomposition of the map-coloring problem into the five constraints $C1$, $C2$, $C3$, $C4$, and $C5$. The approaches for solving constraint-satisfaction problems will be covered in the next section, but generally the five constraints can be viewed as a set of equations without any relation to the map-coloring domain and can be passed to any general constraint solver to be solved.

The versatility, generality, and power of this approach to problem solving encouraged many researchers to try to formulate many problems belonging to a variety of fields in this mold. A large number of seemingly different applications can be solved by modeling them as constraint-satisfaction problems. Applications have included such diverse areas as graph problems [Ullman, 1976, McGregor, 1979], scheduling [Prosser, 1988, Wallace, 1994, Chamard *et al.*, 1992, Van Hentenryck, 1993], machine vision [Waltz, 1975, Davis and Rosenfeld, 1981,

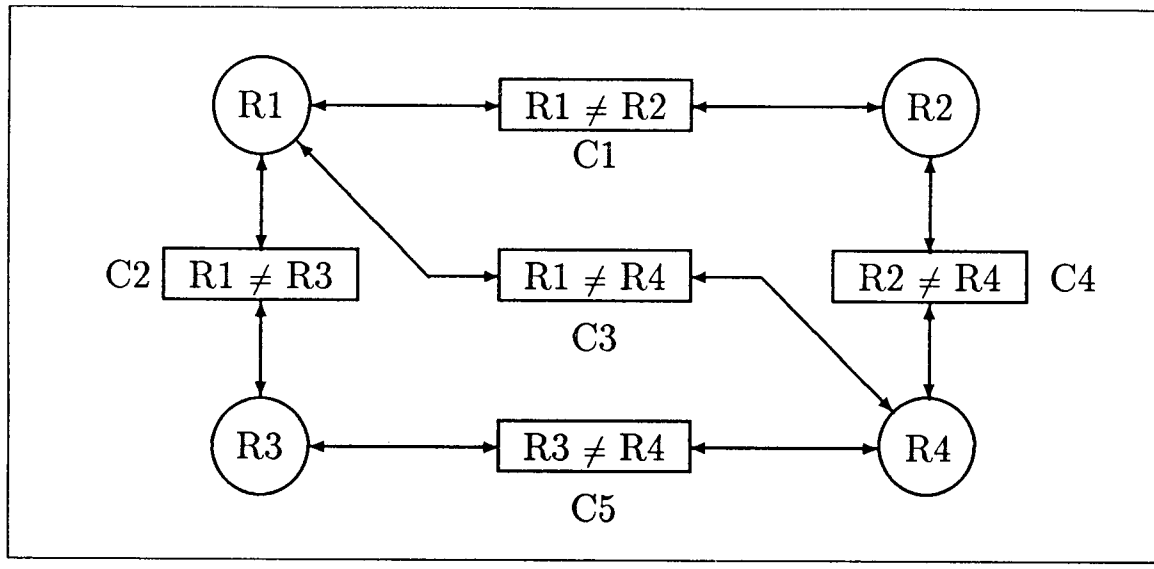

Figure 2.3. Representation of the Map-Coloring Problem as a CSP.

Mackworth, 1977b], floor-plan design [Eastman, 1972], machine design and manufacturing [Frayman and Mittal, 1987], and circuit design [de Kleer and Sussman, 1980].

2.3 Approaches to Satisfying Constraints

In the previous section, we saw how a map-coloring problem was transformed into a constraint-satisfaction problem by constructing it as a set of five constraints $C1, \ldots, C5$. If this problem were to be solved using a constraint language then the process of solving the problem can be viewed as feeding the five constraints in addition to some information about the domains of $R1$, $R2$, $R3$, and $R4$ to a constraint solver. It is the job of the constraint solver to find appropriate values for every Ri. This process is shown in Figure 2.4. The programmer can view the constraint solver as a black-box and not care about the structure of its design or how it solves the constraints. However, if the problem were to be solved using a conventional language with a

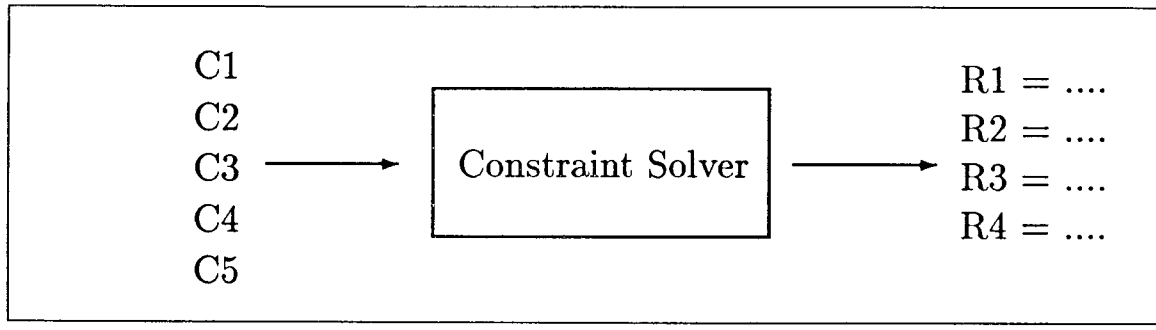

Figure 2.4. The Behavior of Constraint Solvers.

traditional paradigm such as Pascal [Jensen and Wirth, 1985], Smalltalk [Goldberg and Robson, 1983], or LISP [Steele Jr., 1990] then the programmer would have to figure out how to find values for the variables that will satisfy the constraints because he or she would have to write the code that emulates the behavior of the constraint solver. In the following paragraphs we will try to discuss the approaches that have been taken to build constraint solvers or in other words the algorithms used to solve constraints.

2.3.1 Generate-and-Test Strategy

In this approach every possible combination of value-variable assignments is generated and a test is done to see if the generated combination satisfies the given constraints. If we assume the respective domains $D_1, D_2, \ldots, D_n$ of the variables $V_1, V_2, \ldots, V_n$ to be all finite discrete domains, then the space of the problem D, which is defined as $D = D_1 \times D_2 \times \cdots \times D_n$, is also a finite discrete domain. One can evaluate the conjunction $C_1 \wedge C_2 \wedge \cdots \wedge C_n$ on each element of D. If the objective is to search for a single answer then what should be done is to stop the process once the evaluation returns true otherwise the evaluation process should continue collecting all the elements of D that return a true value upon evaluation. This algorithm is

correct but obviously very slow. Actually, its run-time complexity is proportional to the size of $D_1 \times D_2 \times \cdots \times D_n$.

2.3.2 Backtracking

The idea of backtracking is to instantiate the variables one at a time and after each instantiation a check is performed to see if the conjunction $C_1 \wedge C_2 \wedge \cdots \wedge C_n$ has been violated. If so, then the process backtracks and chooses another value for the most recently instantiated variable. If that variable has no more values in its corresponding domain, then the backtracking proceeds by applying the same process to the previously instantiated variable, and so forth until a solution is found. If it is found during the backtracking step that there are no more variables to backtrack to (i.e. the backtracking have searched the entire domain of each variable), then there are no more solutions to the problem at hand. This technique was given the name backtrack by D. H. Lehmer in the 1950's [Bitner and Reingold, 1975]. The major efficiency gain from backtracking is that every time a failure is encountered and a backtrack is performed, a portion of the search tree is pruned. This translates over time into a large reduction in the search space.

There are administrative strategies that can be utilized to improve both the forward and backward movements of the backtracking algorithm. The first is called *Variable-Ordering Strategy* and the second is *Value-Ordering Strategy.*

- Variable-Ordering Strategies: One strategy is to instantiate the variables with smaller domains earlier (i.e. the variables should be ordered in terms of increasing domain size). This will maximize the size of the eliminated subspace due to a single failure [Kumar, 1992]. Another strategy is to start by instantiating variables participating in the highest number of constraints. This has the effect of maximally constraining the rest of the search space [Freuder, 1982].

- Value-Ordering Strategies: When a variable is to be instantiated by assigning to it a value from its respective domain, a fairly good strategy is to choose a

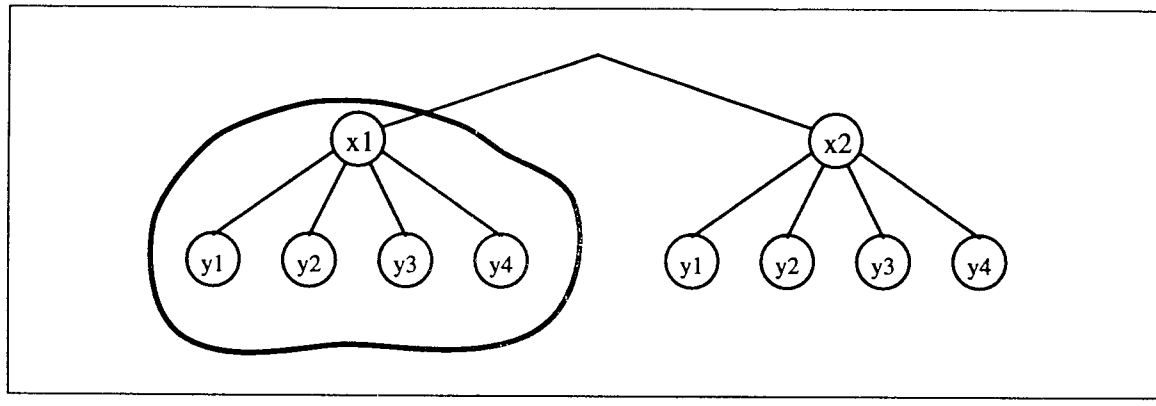

Figure 2.5. Appropriate Variable Ordering.

value that will maximize the number of options for future assignments [Dechter and Pearl, 1987, Haralick and Elliot, 1980].

To illustrate the workings of the variable-ordering strategy, suppose we have two variables X and Y such that the domain of X is composed of the two values x_1 and x_2 and the domain of Y contains the values y_1, y_2, y_3, and y_4. The appropriate ordering of variable instantiation should start with the variable X because a failure on x_1 will eliminate at most four instantiation attempts of the variable Y. Figure 2.5 shows part of the search tree where the first level represents the instantiation of the variable X with the values x_1 and x_2. It also shows how a failure on x_1 will result in pruning that part of the tree.

If the variable instantiation ordering were to start with Y then every failure on a value y_i will only eliminate two future attempts to instantiate the variable X. Figure 2.6 shows this choice of variable ordering. The inappropriateness of the this choice of value ordering is indicated by the fact that the size of the pruned search space is smaller than that of the previous choice.

Once a variable is chosen for instantiation then the question is what value within that variable's domain should be selected first. One value-ordering strategy suggests selecting a value that will maximize the possible selections for other variables. To

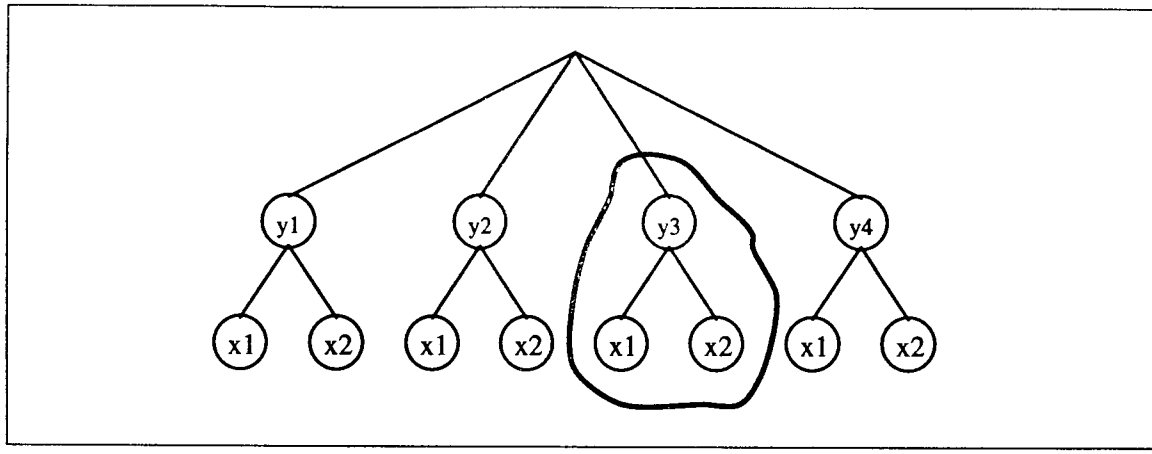

Figure 2.6. Inappropriate Variable Ordering.

clarify that let us look back into the map-coloring problem. Suppose the domain of region $R1$ is the set $\{RED, GREEN, BLUE\}$ and the domain of $R2$ is the set $\{RED, WHITE\}$. The variable-ordering strategy suggests that we start by instantiating $R2$ since it has a smaller domain. If we assign the color RED to $R2$ then $R1$ will have only two possible assignments either $GREEN$ or $BLUE$. This is due to the required satisfaction of constraint $C1$ which states that the color of region $R1$ should not be the same as that of region $R2$. The recommendation of the value-ordering strategy is to start by assigning to $R2$ the color $WHITE$ because that will maximize the possible values for $R1$.

In spite of the fact that the backtracking approach is strictly better than the generate-and-test method, yet its run-time complexity for most practical problems is still exponential. The major reason for this is that backtracking is very often accompanied by a *thrashing* behavior. Thrashing is the behavior of a search that seems to keep on failing in different parts of the search space for the same reasons. Mackworth [Mackworth, 1992] claimed that there is a strong coupling between backtrack search and thrashing by stating that regardless of the instantiation ordering, a thrashing behavior is almost always observed in backtrack search.

A class of algorithms called *dependency directed backtracking algorithms* or also known as *intelligent backtracking algorithms* [Tsang, 1993, Haralick and Elliot, 1980, Gaschnig, 1977] are designed to reduce or eliminate thrashing. The basic objective of most intelligent backtracking algorithms can be simply stated as: If condition C causes the search to fail then instead of backtracking to the last decision point, backtrack to the first possible point where C no longer holds; otherwise condition C will be encountered again. More will be said about thrashing in the next subsection. It is interesting to note that almost all the approaches that were used in the implementation of the logic programming language Prolog [Clocksin and Mellish, 1987] relied on some form of backtracking.

2.3.3 Consistency Algorithms

Many researchers have studied the various causes of thrashing behavior in backtracking and they have designed algorithms that eliminate those causes [Mackworth, 1977a, Waltz, 1975, Montanari, 1974, Freuder, 1978]. These algorithms are grouped in a class of algorithms called *consistency algorithms* [Mackworth, 1977a]. Consistency algorithms prevent against thrashing by eliminating, from the respective domains, all the values that can cause it. This is done by analyzing the participating variables, their domains, and the relations between them and imposing certain consistency rules before the search is conducted. These rules include *node consistency*, *arc consistency*, and *path consistency*. In the following paragraphs we will explain these rules by reviewing some simple thrashing cases.

Suppose that a variable V_i has a unary constraint C_i and it draws its values from the domain D_i. Suppose also that the domain D_i has a value v_j such that the assignment $V_i = v_j$ always violates the constraint C_i. This is a case of thrashing called *node inconsistency* [Mackworth, 1977a] because the instantiation of V_i with v_j will always result in a failure. This situation can be fixed by removing the value v_j from the domain D_i. In general, node inconsistency can be eliminated by inspecting every

variable and removing from its domain any value that violates any unary constraint that is imposed over that variable.

Another cause of thrashing is the situation where instantiating a variable with a certain value prevents another variable from being instantiated to any value in its domain. Revisiting the map-coloring problem, suppose that $R2$ has the domain $\{GREEN, RED, BLUE\}$ while $R4$ has the domain $\{BLUE\}$. Instantiating $R2$ with the value $BLUE$ will prevent $R4$ from being instantiated to any value, because of the constraint $C4$ which states that $R2 \neq R4$. Therefore, every time the above instantiation occurs, an immediate failure will take place and a backtrack must be performed. The cause of this kind of thrashing is described as a lack of *arc consistency* [Mackworth, 1977a]. Arc consistency in a constraint graph is achieved if for every two variables V_i and V_j that participate in the constraint $C(V_i, V_j)$, the following holds: for every value a in the domain D_i of V_i there exists a value b in the domain D_j of V_j such that the assignments $V_i = a, V_j = b$ will not violate the constraint $C(V_i, V_j)$. The above can be formally stated as follows,

$$(\forall a)[a \in D_i] \supset (\exists b)(b \in D_j) \wedge C(V_i = a, V_j = b)$$

Obviously, arc consistency between two variables V_i and V_j can be obtained by removing from the domain of V_i all the values for which the above condition is not true.

Path consistency is a generalization of the arc consistency rule. There are algorithms for achieving arc and path consistencies. The first such algorithms, AC-1, AC-2, AC-3, and PC-1, PC-2 were proposed in [Mackworth, 1977a]. It is important to know that the removal of values in order to attain consistency does not eliminate any possible solution for the constraint satisfaction problem [Kumar, 1992].

2.3.4 Other Approaches and Domains

In the previous paragraphs, we reviewed a number of approaches to solving constraints over variables drawn from finite discrete domains. A number of techniques

have been developed to solve problems over other types of domains such as the domain of real numbers or Booleans. In the case of rational numbers or real arithmetic where we typically have linear equations and inequalities, the Gaussian elimination method can be used to solve a system of linear equations and linear programming techniques can be applied to solve a system of inequalities [Nemhasuer *et al.*, 1989]. It is important to mention that the solutions for problems whose variables are drawn from infinite domains generally require domain-specific algorithms. Constraint-satisfaction problems belonging to the domain of Boolean values can be solved using a variety of Boolean unification algorithms

Term rewriting is an approach that has been used in building constraint-satisfaction systems and languages such as the Purdue Equational Interpreter [O'Donnel, 1985] and Wim Leler's constraint language Bertrand [Leler, 1988]. A term rewriting system is composed of a set of rules that defines rewriting or reduction relations between terms. A rewrite is done when a match is found between the left-hand side of a rule (i.e. rule head) and the contents of an expression or a subexpression. In general, the process can be defined as the application of a set of rewrite rules to an expression to transform it into another expression.

Chapter 3
Electra Constraint Constructs

3.1 Introduction

This chapter describes the constraint constructs that have been integrated into the Leda language to define the language Electra. It begins by presenting a motivating example to show the expressiveness and flexibility of constraints. It follows by giving a short description of the categories of constraints as viewed by Electra. Finally, it closes by presenting the characteristics and behavior of fixable and satisfiable constraints. The examples used in this chapter are simple because they are presented not for the sake of emphasizing their usefulness but rather to serve as means to explain the underlying concepts. More useful and meaningful examples will be presented in the following chapters.

3.2 A Motivating Example

The declarative nature of constraints makes them very useful constructs that can aid programmers in expressing solutions to complex problems. As a motivation to lead to the following discussion of constraints, this section shows two solutions to the map-coloring problem that was presented in the previous chapter.

As described earlier, the goal of the map-coloring problem is to color each region of a given map such that no two adjacent regions have the same color. Figure 2.2 illustrates an example of a map with the four regions $R1$, $R2$, $R3$, and $R4$. Figure 3.1 displays one way of solving the map-coloring problem in C [Kernighan and Ritchie, 1988], and Figure 3.2 shows a listing of an Electra solution to the same problem using the constraint construct. Comparing the two solutions, it is easy

```
enum Region {red, green, blue};

enum Region  R1, R2, R3, R4;

int map_coloring()
{
  /*
   * Search for possible values for R1, R2, R3, and R4
   * such that the desired relation is valid
   */
  for (R1 = red ; R1 <= blue ; R1++ )
      for (R2 = red ; R2 <= blue ; R2++ )
          for (R3 = red ; R3 <= blue ; R3++ )
              for (R4 = red ; R4 <= blue ; R4++ )
                  if ( (R1 != R2) && (R1 != R3) && (R1 != R4) &&
                       (R2 != R4) && (R3 != R4) )
                      /* A possible solution is found */
                      return 1 ;

  /* No solution is found */
  return 0;

}
```

Figure 3.1. A C Solution to the Map-Coloring Problem.

```
var
  R1, R2, R3, R4 : Region;

 constraint Map_Coloring ;
  assert
       (R1 <> R2) & (R1 <> R3) & (R1 <> R4)
       &  (R2 <> R4) & (R3 <> R4)
 end;
```

Figure 3.2. An Electra Solution to the Map-Coloring Problem.

to see that the approach used to construct the Electra solution produces a shorter program. The reason for the differences in code size between the two approaches is that the Electra solution is only listing the desired relations, while the C solution is listing the relations in addition to the mechanism used to conduct the search (i.e. the generate-and-test mechanism). This comparison is not intended to emphasize the differences in code size because shorter programs are not necessarily clearer. For example, APL [Iverson, 1962] is a language that is notorious for its characteristic of having compact programs that are not easy to read which lead some researchers to label it as a *write-only* language [Budd, 1995]. It is important to note that the differences between the two solutions are not limited to program size and clarity but rather include a major semantic difference. In the C version, whenever the function `map_coloring` is called, it goes through the four for-loops searching for the first possible values for the variables `R1`, `R2`, `R3`, and `R4` such that the composite relation `R1 != R2 && R1 != R3 && R1 != R4 && R2 != R4 && R3 != R4` is valid. It returns 1 once the desired values are found, otherwise the function returns 0. Outside this function, there is no enforced restriction that dictates what values can be assumed by the four variables. This means that the above relation is not guaranteed to be valid at all times. Compare that with the Electra solution, where upon encountering the constraint `Map_Coloring` a solution is found by the satisfier. If one of the four variables is to be updated anywhere in the program then the solver will try to keep the relation valid by altering the values of one or more of the other three variables. The semantics of Electra constraints seem to reflect the statement of the map-coloring problem more faithfully.

3.3 General Characteristics of Constraints

The versatility, expressiveness, and generality of the concept of constraints as building blocks in formulating constraint-satisfaction problems motivated us to create a

syntactic construct that captures the behavior of a constraint as a relation linking entities together. We call this construct a *constraint construct.*

The Electra language divides constraints into separate categories depending on two criteria. The first is the type of influence they have and the second is the method by which they are solved. In terms of influence or types of participants, constraints are classified into two separate designations. The first are those that declare relations between objects regardless of what class these objects belong to. For example, in the context of user interfaces, if we want to have an arrow always pointing to a specific corner of a particular box, then we can declare a constraint that states that the coordinates of the arrow's head are always equal to the coordinates of that corner. This constraint is relating an arrow to a box which belong to two different classes. The second designation are constraints that declare assertions relevant to all instances of a particular class. Let us call the former type *object constraints*, the latter *class constraints.* Class constraints can be further divided into two types, *intra-instance class constraints* and *inter-instance class constraints.* Intra-Instance class constraints state assertions relevant to every individual instance of a particular class. They do not express any relationships between the instances. An example of this type is the constraint that states that no human being can have an age of less than or equal to zero. It expresses a restriction over the value that can be assumed by the age of every individual human being, which does not state any relationship between individual humans. An inter-instance class constraint is concerned with the relationships between all the instances of the class where it is declared. For example, the constraint that states that every employee must have a unique identification number is considered an inter-instance class constraint since it expresses a relationship among all the employees. Figure 3.3 presents the influence based categorization of constraints.

In terms of solvability, Electra distinguishes two classes of constraints. We call the first *fixable constraints* and the second *satisfiable constraints.* Fixable constraints are constructs that declare as part of their structure a component called *the fix.*

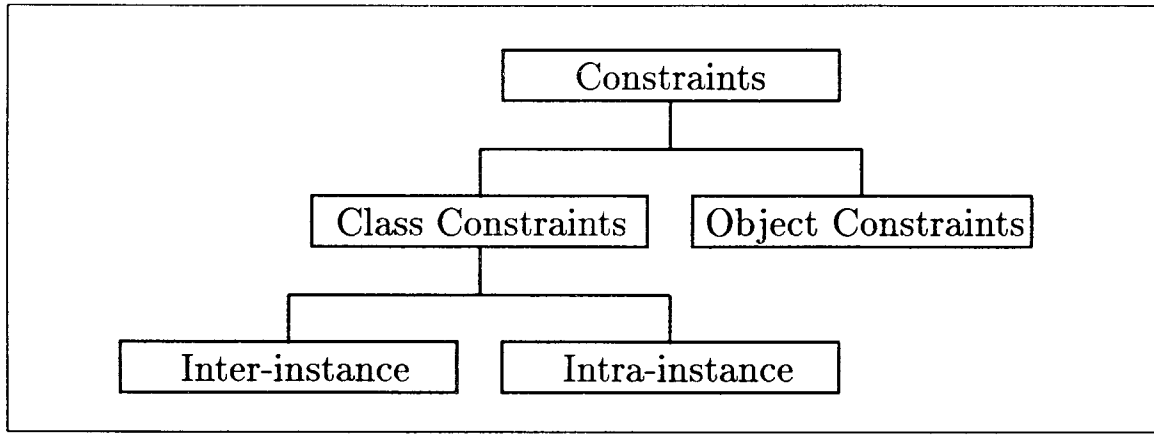

Figure 3.3. Constraint Categories Based on Influence.

The functionality of this component is to bring its enclosing constraint back to a state of consistency whenever that constraint is broken. The process of solving satisfiable constraints is totally independent of the Electra compiler and is achieved by a separate external component called the constraint-satisfaction system. Once a satisfiable constraint is violated, then the constraint-satisfier is informed and it is up to the satisfier as to how the situation is handled. The solvability categorization is shown in Figure 3.4.

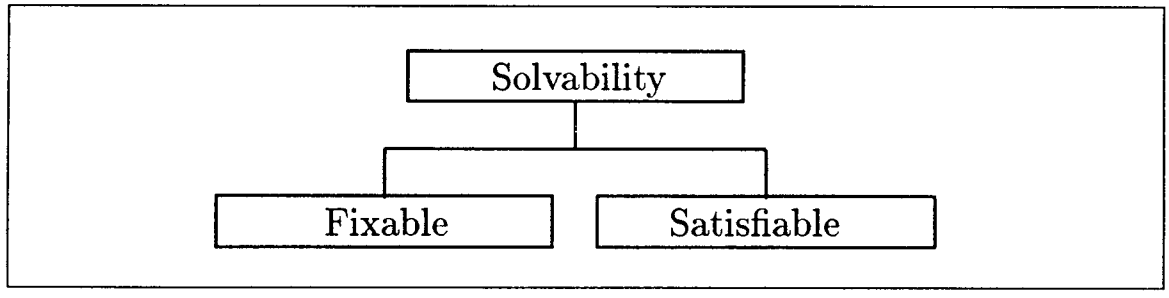

Figure 3.4. Constraint Categories Based on Solvability.

3.4 Fixable Constraints

A fixable constraint is a syntactic structure that declares a relation between a set of variables. This type of relation is maintained by the Electra compiler once all the participating variables are defined. In Electra when a variable is declared, it is classified as being undefined. An undefined variable is a variable that has as a value the generic object NIL. Any undefined variable can be made defined by assigning it a legal value from its respective domain. Variables can shift status from being defined to becoming undefined by assuming the value NIL.

If an assignment to one of the constrained variables results in breaking the stated assertion then the fix part is invoked. The function of the fix part is to alter the participants in a manner that brings the assertion expression back to a state of being valid. Figure 3.5 shows the syntax of fixable constraints.

```
⟨fixableConstraint⟩ ⟶ ⟨fixableConsHeader⟩
                       ⟨fixableConsAssertion⟩
                       ⟨fixPart⟩
                       end;
⟨fixableConsHeader⟩ ⟶ constraint ⟨id⟩ ;
⟨fixableConsAssertion⟩ ⟶ assert ⟨expression⟩ ;
⟨fixPart⟩ ⟶ fix ⟨nonReturnStatements⟩
```

Figure 3.5. Syntax of Fixable Constraints.

Once a fixable constraint is declared, it becomes active within its scope unit. Upon activation, a fixable constraint validates its assertion to ensure that all the variables participating in the assertion expression are satisfying it. If one of the

participating variables is not defined yet, then the constraint is considered satisfied. When a variable that is constrained by a fixable constraint is to be updated then the new value is checked with the constraining assertion. If the update does not bring about a violation to the assertion then the update is processed normally; otherwise the old value of the updated variable is saved before processing the update. The old value can be accessed using the `old()` operator which takes, as argument, the name of the updated participating variable and returns its old value. After the update is performed, the fix part of the constraint is executed. Examples and further description of the `old()` operator and the behavior of the fix part will be presented in the following sections.

3.4.1 The Assertion Part of a Fixable Constraint

Any valid Electra Boolean expression can serve as an assert expression in a fixable constraint. The only restriction is that a variable can only participate in a single assert expression (i.e. a participant of a fixable constraint is not permitted to participate in any other constraint). The reason for this restriction is that permitting a variable to participate in more than one constraint will make it difficult to decide which fix to invoke if an update to that variable were to result in violating more than a single constraint. This restriction does not impose any limitations because a number of assertion expressions can be conjugated forming a single assertion that is a conjunction of the basic assertions. Such conjunctions of basic assertions will appear in some of the examples that will be presented.

3.4.2 The Fix Part of a Fixable Constraint

The fix statement is a statement that is supposed to bring the constraint back to a state where its assertion is satisfied. Therefore, it is not permitted to terminate without bringing its constraint's assertion to a valid state. During run-time, a check is performed before exiting the fix part to be sure that the cumulative actions of

the fix do not violate the constraint's assertion expression. If a violation is detected then a run-time error is generated. Inside the fix part, one can take advantage of the availability of the predicate `BrokenBy()`. This predicate takes as a parameter a participating variable and returns true if that variable is the one that violated the assertion and caused the fix to be invoked, otherwise it returns false. Another function that is provided for usage inside the fix part is the function `old()`, which when passed an updated participant, returns its previous value.

3.4.3 The Behavior of Fixable Constraints

The behavior of fixable constraints is explained by analyzing the following simple yet illustrative example. Suppose that we have the two variables x and y such that the relation $x = y - 1$ between them is to be maintained at all times. If an update to one of the variables were to result in breaking the above relation, then the other variable should be altered in such a manner that the validity of the relation is restored. Figure 3.6 shows an instance of a fixable constraint construct that expresses the stated relation. The constraint `x_y_relation` which has the assertion

```
constraint x_y_relation ;
  assert
        x = y − 1;
  fix
     if brokenBy( x ) then
        y := x + 1;
     else
        if brokenBy( y ) then
           x := y − 1;
end;
```

Figure 3.6. Simple Fixable Constraint.

$x = y - 1$, declaratively reflecting the relation to be maintained. The variables x and y are referred to as participants of the constraint `x_y_relation`. Suppose that the participants x and y are assuming the values 5 and 6 respectively. The current values of the variables are adhering to the assertion of the constraint linking them. However, if variable x is to be assigned a new value such as 8 which will result in violating the assertion then the fix will be invoked. Inside the fix, a check is made to find out which variable caused the violation. This is done using the predicate `brokenBy()`. This check guides the actions taken by the fix to resolve the situation. In the above case, the action that will be taken is to set the variable y to the value 9 which will satisfy the assertion. The old value of x is stored before the fix is invoked and it can be retrieved by calling the built-in function `old()` with the parameter x.

3.5 Satisfiable Constraints

A satisfiable constraint is a constraint that is maintained and solved by an underlying constraint satisfaction system. Finger 3.7 shows the syntax of satisfiable constraints in Backus-Naur Form. The syntax illustrates that the body of a satisfiable constraint is composed of three components: a name, a strength, and an assertion. This type of construct is suitable for expressing a multiple of constraints that form a network, where the only way to have all the constraints satisfied is to find values for the constrained variables such that all the relations are simultaneously satisfied. This process requires a search and cannot be achieved using a simple fix because, as justified earlier, invoking the fix of one constraint might result in breaking another constraint in the network. To avoid such cases, the process of satisfying the whole network of constraints is assigned to an external solver that is designed primarily for such purpose. The external solver is responsible not only for solving the network of constraints, but also for maintaining the values of their participants. The Electra compiler does not keep track of the values of the participants of a satisfiable constraint but rather delegates that task to the solver. This avoids having to

```
⟨satisfiableConstraint⟩ ⟶ ⟨satisfiableConsHeader⟩
                            ⟨satisfiableConsAssertion⟩
                            end;
⟨satisfiableConsHeader⟩ ⟶ constraint ⟨id⟩ : ⟨strength⟩ ;
⟨satisfiableConsAssertion⟩ ⟶ assert ⟨andSatAssertion⟩ ;
⟨andSatAssertion⟩ ⟶ ⟨satAssertion⟩
                    | ⟨andSatAssertion⟩ & ⟨satAssertion⟩
⟨strength⟩ ⟶ required
             | strong
             | medium
             | weak
```

Figure 3.7. Syntax of Satisfiable Constraints.

have duplicate copies of each participant, which might result in jeopardizing their integrity. Whenever a participant of a fixable constraint is updated, then the satisfier is informed about the update to keep its symbol table up-to-date, and whenever a participant is accessed then the satisfier is queried about that participant's value.

3.5.1 The Assertion Part of a Satisfiable Constraint

Unlike fixable constraints, where any valid Electra boolean expression can be used as an assertion expression, a satisfiable constraint's assertion must be an expression that can be understood by the constraint solver. This restriction is not limited to the constraint's assertion expression but also every participant must be drawn from a domain that can be handled by the solver. Therefore the degree of complexity of the assertion expressions and the generality of the domains are implementation dependent. This is due to the fact that the Electra language does not state any

specifications or restrictions over the characteristics of the satisfier. This approach is similar to the approach taken by the designers of the language Gödel [Hill and Lloyd, 1994].

3.5.2 The Behavior of Satisfiable Constraints

Once a satisfiable constraint is encountered, information about its name, strength, assert expression, and participants is passed to the satisfier. Since the satisfier keeps track of the participants then their initial values cannot be determined.

Figure 3.8 shows a network of two satisfiable constraints. The constraint `CandF` expresses the relation between Fahrenheit and Celsius temperatures and the constraint `CandK` expresses the relation between Celsius and Kelvin temperatures. Once the compiler parses the aforementioned constraints, it passes them to the solver. The solver will start by assigning arbitrary values to the participants such that the two constraints `CandF` and `CandK` are valid. The solver is kept informed regarding any new update to the participants `C`, `F`, and `K`. The two constraints form a network as shown in Figure 3.9. The satisfier used for the current implementation is the DeltaBlue satisfier designed by Borning and his group at the University of Washington [Maloney, 1991]. Appendix A gives a more detailed description covering the design of the DeltaBlue constraint solver.

3.5.3 Required Versus Preferential Constraints

In some applications it is necessary to state both required and preferential constraints. The required constraints must always be satisfied, while the preferential ones should be satisfied, if possible. If the system is unable to satisfy a preferential constraint then no error will be generated.

To express this hierarchical structure of constraints in Electra, one can use the strength level field, where there are four levels of strength: *Required*, *Strong*, *Medium*,

```
var
   c :integer;
   f :integer;
   k :integer;

constraint CandF : required;
  assert
      (f − 32) ∗ 5 = c ∗ 9  ;
end;

constraint CandK : required;
  assert
      k = c − 273  ;
end;
```

Figure 3.8. Relations Between Celsius, Fahrenheit, and Kelvin Temperatures.

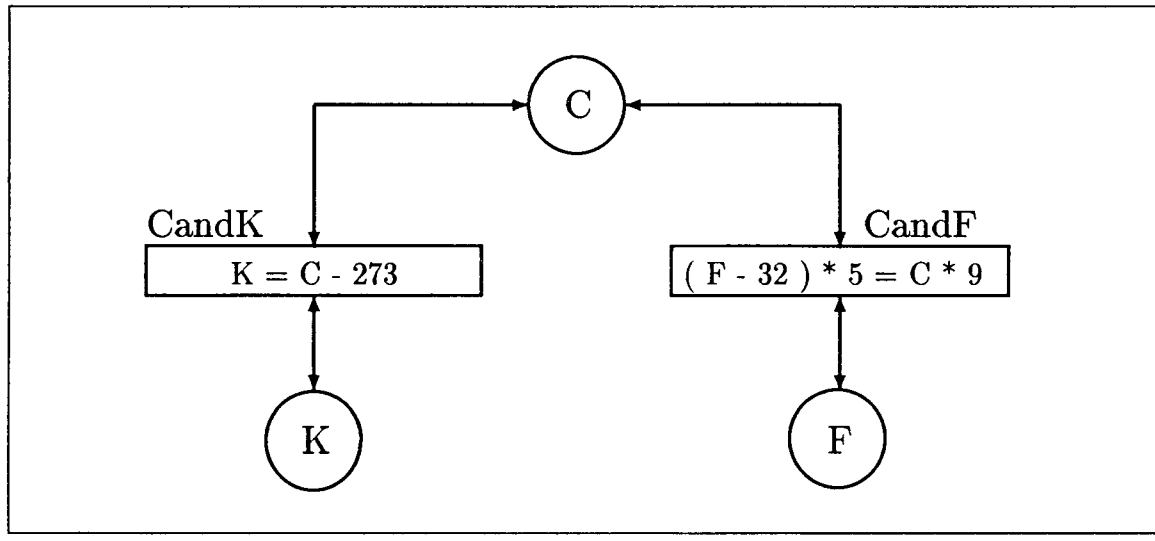

Figure 3.9. A Network of Satisfiable Constraints.

and *Weak*. Required constraints are satisfied first then preferential constraints are prioritized according to their strength level.

3.6 Class Constraints and Object Constraints

Object constraints, as defined earlier, are constraints that state relations between objects regardless of what class these objects belong to. The constraints are normally defined independently and not as part of any class. The objects that participate in an object constraint do not have to be distinct. Thus, an object constraint could be used to relate an object to itself. This is useful in stating relations that are relevant to individual objects.

Figure 3.10 shows the declaration of the classes `Point` and `Line` and an example of an object constraint. The constraint `line1_Vertical` is an object constraint that constrains the particular instance `line1` of class `Line` to be always vertical. If one of its points is to be updated then the second will follow and the line will be redisplayed.

A class constraint is considered part of a particular class. It functions as a guard that imposes a particular restriction on all the instances of its class, or a particular relation among them. For example, one can create the class `VerticalLine` which is a subclass of class `Line` with the constraint that the x-coordinates of the points are to be always equal as shown in Figure 3.11

The constraint `Vertical` is part of the class `VerticalLine`, and thus, it is referred to by the name `VerticalLine.Vertical`. This constraint is automatically attached to every instance of the class `VerticalLine`. It is classified as an intra-instance class constraint since it imposes an assertion over every instance of the class `VerticalLine`, yet it does not express any relation between the individual instances.

Sometimes, it is necessary to relate instances of one class to each other or to speak of all instances of a certain class. To achieve that, one can use the built-in function `instances_of()` which takes a class name and returns a list of all the instances of that class and its subclasses. For example, suppose that we are defining

```
class Point;
  x : integer;
  y : integer;
end;

class Line;
  p1 : Point;
  p2 : Point;
end;

var
  line1 : Line;

  constraint line1_Vertical;
   assert
       line1.p1.x = line1.p2.x;
   fix
     begin
      if BrokenBy( line1.p1.x ) then
          line1.p2.x := line1.p1.x;
      else
          line1.p1.x := line1.p2.x;

      line1.display;
     end;  // end of fix part

   end; // end of constraint body
```

Figure 3.10. Declaration of an Object Constraint.

a class that describes the concept employee, and we want to express the notion that every employee must have a unique identification number. Figure 3.12 shows a possible definition of such a class. Class `Employee` declares the constraint `UniqueID` which is an inter-instance class constraint because it enforces a relation over all the instances. Every time an instance of class Employee is created, a constraint linking it to all other instances is attached to the newly created instance. Figure 3.13 shows how inter-instance class constraints are added as instances are created. In part (a) there is only one instance of class `Employee` namely `e1` therefore there is no need

```
class VerticalLine of Line ;
    constraint Vertical;
      assert
           p1.x = p2.x;
      fix
        begin
           if BrokenBy( p1.x ) then
                p2.x := p1.x;
           else
                p1.x := p2.x;
           display();
        end;  //end of fix
      end;  // end of constraint
end;
```

Figure 3.11. Declaration of an Intra-Instance Class Constraint.

```
class Employee;
    name : names;
    age  : ages;
    ID   : IDNumbers;
     ... Other instance vars and methods ...
    constraint UniqueID;
      assert
          forall e1 in instances_of (Employee)
             if e1 <> self then
                 e1.ID <> ID;
      fix
          ...
    end;
end;
```

Figure 3.12. Declaration of an Inter-Instance Class Constraint.

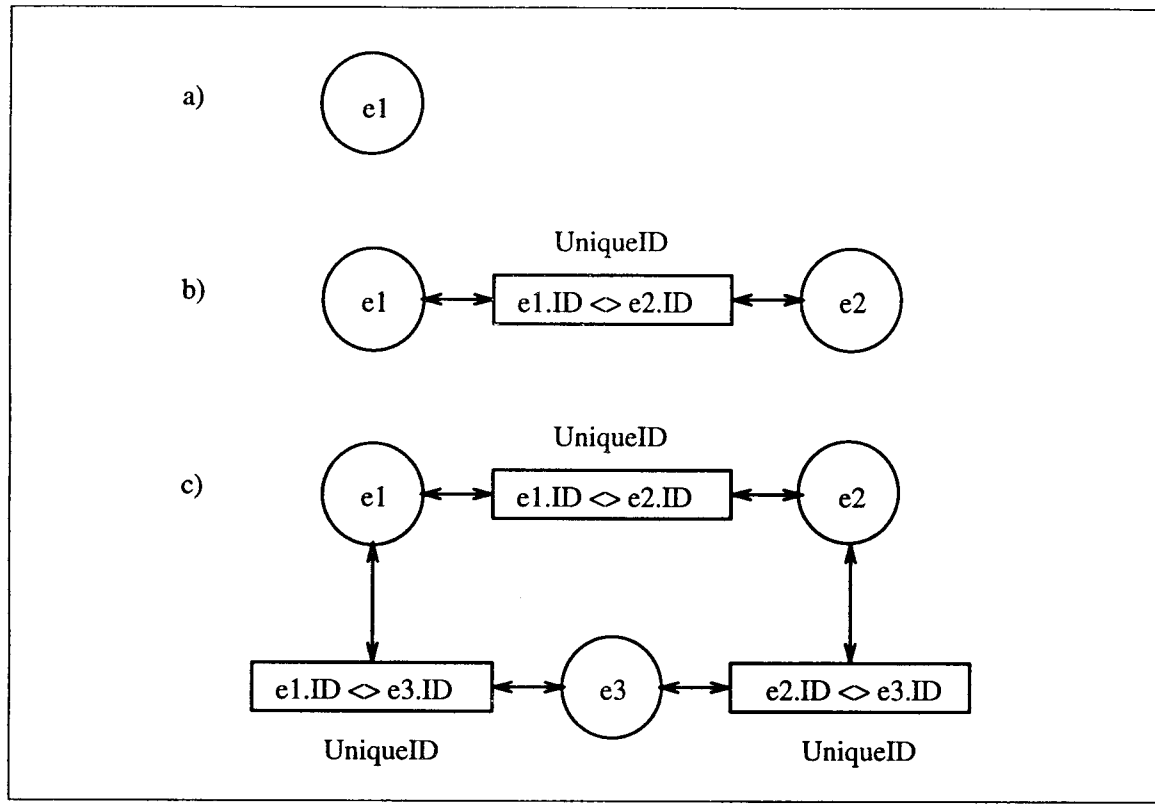

Figure 3.13. The Behavior of Inter-Instance Class Constraints.

to enforce the constraint `UniqueID`. However, once `e2` is created as a new instance of class `Employee` a copy of `UniqueID` is created linking instances `e1`, and `e2` and stating the ID fields of the two objects should be different as shown in part (b). Part (c) shows the network of three constraints linking three variables upon the creation of the third instance `e3`.

Although this approach seems to be too costly, it should be noted that the programmer would have to do the same kind of checking if ID uniqueness is to be assured.

Chapter 4
Condition-Based Dispatching

This chapter explains the concept of *condition-based dispatching*, which is a concept that is similar to argument pattern-matching in functional languages. The chapter begins by reviewing the techniques of functional abstraction and parametric overloading. After that, it introduces condition-based dispatching and presents its characteristics with some examples to illustrate the advantages and appropriate usage of this technique. The guarded function construct is presented as a tool for expressing condition-based dispatching in the language Electra. The chapter ends by presenting a comparison between guarded functions and argument pattern-matching in functional programming, illustrating how guarded functions provide Electra programmers with the ability to simulate the behavior of argument pattern-matching.

4.1 Functions, Dispatching, and Parametric Overloading

The term *function*, as understood in the context of programming languages, is an abstraction technique via which statements and expressions are grouped and can only be invoked explicitly. An invocation of a function returns flow of control to the point following the invocation point. A function is invoked by issuing a function call, which names the particular function and supplies its actual parameters, if any. The function call starts by searching for the specified routine. Once it is found, execution control is transferred to the beginning of that routine. This process is called *function dispatching* [Bal and Grune, 1994].

Overloading is the ability to bind one name to more than a single object. However, since names are used to identify objects, it is natural to conclude that overload-

ing will cause an ambiguity. Therefore, in order for overloading to be useful, enough information must be provided to resolve any ambiguities. Budd [Budd, 1991b] defines parametric overloading as:

> Overloading of function names in which two or more procedure bodies are known by the same name in a given context, and are disambiguated by the type and number of parameters supplied with the procedure call.

Thus in the traditional notion of method invocation, when a generic function is invoked, the dispatcher determines which function body to dispatch to simply by checking the number and type of the actual arguments. Electra extends the disambiguation procedure performed by the dispatcher by adding conditions to the formal arguments list. With this extension, the function dispatched to is selected by checking if the values of the actual arguments satisfy the conditions imposed over the formal arguments. We call this type of parametric overloading *condition-based dispatching.* This technique is similar to the process of argument pattern-matching that is provided by a number of functional programming languages [MacLennan, 1990, Field and Harrison, 1988, Peyton Jones, 1987]

4.2 Guarded Functions

The technique of condition-based dispatching can be expressed in Electra using a construct called the *guarded function.* A guarded function differs from a regular function in that its name can be overloaded by defining multiple guarded functions with the same name. and that its signature has another component called *the function guard.* A function guard can be viewed as a necessary precondition for executing the particular function body which it guards.

The syntax of guarded functions is shown in Figure 4.1. The signature of a guarded function is similar to that of a regular function. The difference between the two is that the signature of a guarded function starts with the keyword `guarded` and ends with a guard expression. Any Electra boolean expression can serve as a guard expression. The name of a guarded function can refer to more than a single function

body. Each body is referred to as an instance of the guarded function. When a guarded function is called, the guard for the first declared instance is evaluated. If the evaluation results in a true value then the dispatcher dispatches to that instance; otherwise the guard of the next instance is evaluated. The process of evaluating guards continues until an evaluation returns true or all the instance functions are exhausted.

⟨functiondeclaration⟩ ⟶ ⟨guardedFuncHeadAndGuard⟩
⟨declarations⟩
⟨body⟩ ;
⟨guardedFuncHeadAndGuard⟩ ⟶ ⟨guardedFuncHead⟩ ⟨guard⟩ ;
⟨guardedFuncHead⟩ ⟶ ⟨guardedFuncName⟩ ⟨valueArguments⟩
⟨optReturnType⟩ ;
⟨guardedFuncName⟩ ⟶ `guarded function` ⟨id⟩ ⟨typeArguments⟩
⟨guard⟩ ⟶ [⟨expression⟩]

Figure 4.1. Syntax of Guarded Functions.

The mechanism of condition-based dispatching provides a number of advantages:

- It offers an alternative to using some complicated or deeply nested conditional statements.
- It serves as a design aid in helping the programmer to consider all possible inputs to a function.
- It provides a mechanism for direct mapping of specifications to actual code.

- It can be viewed as an approach to make code more readable and thus easier to maintain and enhance.

The benefits of condition-based dispatching become more apparent when more than two alternatives are to be considered, especially in cases where each alternative constitutes a large number of actions which result in having the condition statement body spreads over a number of pages. This approach to decomposing functions is generally concise and readable. Contrast this approach with the approach of using conditional statements which often requires nested if–then–else statements and is consequently less easy to read, comprehend, and modify.

4.3 Tree Insertion: An Example

Figure 4.2 shows a tree insertion example exactly as presented by Kernighan and Ritchie [Kernighan and Ritchie, 1978]. The function `tree()` is part of a word counting program and is called to insert a new word into a binary tree, or increment the count of an existing word. It is invoked with the root node to find where to insert the word that is passed to it. The process continues by comparing that word to the one residing in the current node. The new word is percolated down to either the left or the right subtree. If the new word is found to be already in the tree then a counter reflecting the number of occurrences of the word is incremented, otherwise a new node is created to host the newly inserted word. The function `talloc()` is a user-defined function that is designed to create storage for a new node. Similarly, the function `strsave()` is used to copy the new word.

The behavior of the function `tree()` can be expressed by the following equation:

$$tree(p, w) = \begin{cases} \text{create node(w)} & \text{if p = NULL} \\ \text{increment } p\text{->}count & \text{if } p\text{->}word = \text{w} \\ \text{tree}(p\text{->}left, \text{w}) & \text{if } p\text{->}word < \text{w} \\ \text{tree}(p\text{->}right, \text{w}) & \text{otherwise} \end{cases}$$

The above equation shows that the expression on the right hand side is dependent

```
struct tnode *tree(p, w) /*install w at or below p */
struct tnode *p;
char *w;
{
  struct tnode *talloc();
  char *strsave();
  int cond;

  if (p == NULL) {  /* a new word has arrived */
      p = talloc(); /* make a new node */
      p→word = strsave(w);
      p→count = 1;
      p→left = p→right = NULL;
  } else if ((cond = strcmp(w, p→word)) == 0)
      p→count++;    /* repeated word */
  else if (cond < 0)  /* lower goes into left subtree */
      p→left = tree(p→left, w);
  else                /* greater into right subree */
      p→right = tree(p→right, w);
  return(p);
}
```

Figure 4.2. Tree Insertion Using C.

on the four listed conditions. This motivates another approach to solving the tree insertion problem that is based on condition-based dispatching. Figure 4.3 lists an Electra solution to the tree insertion problem. The problem is decomposed into four cases. Each case is represented by an instance of the guarded function `tree()`.

When a call is issued to the guarded function `tree()`, the dispatcher will test the guards one at a time and dispatch to the first instance whose guard expression evaluates to true. Note that the last instance has a catch-all guard expression namely the expression `[ true ]` which always evaluates to true.

Substituting guarded functions for conditional statements in the representation of a problem is sometimes a better alternative, especially in cases where the description of the problem is composed of a large number of complex or nested conditions.

```
guarded function tree ( p: TreeNode; w:String ) → TreeNode;
[ p = NIL ];
begin
    p := TreeNode();
    p.word := w;
    p.count := 1;
    p.left := NIL;
    p.right := NIL;
    return p;
end;

guarded function tree ( p: TreeNode; w:String ) → TreeNode;
[ p.word = w ];
begin
    p.count := p.count + 1;
    return p;
end;


guarded function tree ( p: TreeNode; w:String ) → TreeNode;
[ p.word < w ];
begin
    p.left := tree(p.left, w);
    return p;
end;

guarded function tree ( p: TreeNode; w:String ) → TreeNode;
[ true ];
begin
    p.right := tree(p.right, w);
    return p;
end;
```

Figure 4.3. Tree Insertion Using Electra's Guarded Functions.

For example, an important part of any windowing system is the code that describes the behavior of the system when a mouse button is clicked. The action to be taken when an event such as this occurs depends on many factors. Before a decision is made as to what action to take, a number of questions must be answered such as:

- Which mouse button is clicked?
- Which window is the cursor on?
- Was the Shift or Control keys simultaneously pressed?
- What is the status of the system?

There could also be other questions related to the characteristics of that particular system. Figure 4.4 shows a fragment of code that uses conditional statements to implement the behavior of a windowing system when a mouse button is clicked. Figure 4.5 utilizes guarded functions to implement the same behavior. Decomposing behavior into smaller components aids the programmer not only in easily comprehending the code, but also in effortlessly enhancing it, since adding a new functionality to the system only requires creating a small segment that implements the specification of the new enhancement and usually does not involve altering existing code.

4.4 Order of Declaration of Guarded Functions

When a call is issued to a function that is defined as a guarded function, the dispatcher checks the alternative definitions sequentially starting with the earliest defined one. Once a particular guard evaluates to true, that instance of the guarded function is chosen and invoked. Since guard conditions are checked sequentially by the dispatcher, it is reasonable to wonder if there is any significance to the ordering of declaration of guarded function instances. If the guard conditions are mutually exclusive (i.e. values that satisfy one guard's condition will not satisfy any other condition), then it does not matter in which order the guarded functions are declared.

```
function mouse_click( location : Locations; button : Buttons)
begin
    if (button = 1 ) then
    begin
      if (location = root ) then
          ...
      else if (location = window1 ) then
          ...
      else if  (location = window2 ) then
          begin
            if (shift = true) then
              ....
            else
              ....
          end
      else if (location = FontMenu ) then
          ...
    end
    else if (button = 2 ) then
      ...
      ...
end;
```

Figure 4.4. Description of Behavior Using Conditional Statements.

However, if the conditions overlap, then the order of declarations is crucial. The best approach for figuring out the proper ordering of declarations of instances of a guarded function is to arrange the instances such that the ones with more specific guard conditions appear first. The example shown in Figure 4.6 has an inappropriate ordering of instances because any value that is less than ten is not guaranteed to be less than 3. Therefore, the second instance will never be dispatched to. The guarded function instance with the condition `x < 3` should be declared first.

```
guarded function mouse_click( location : Locations; button : Buttons)
[(button = 1 ) & (location = root ) ];
begin
    ...
end;

guarded function mouse_click( location : Locations; button : Buttons)
[(button = 1 ) & (location = window1 )];
begin
    ...
end;

guarded function mouse_click( location : Locations; button : Buttons)
[(button = 1 ) & (location = window2 ) & (shift = true)];
begin
    ...
end;

guarded function mouse_click( location : Locations; button : Buttons)
[ ... ];
begin
    ...
end;
```

Figure 4.5. Description of Behavior Using Guarded Functions.

```
guarded function foo(x integer);
[ x < 10 ];
begin
    ....
end;


guarded function foo(x integer);
[ x < 3];
begin
    ....
end;
```

Figure 4.6. Improper Ordering of Declarations of Guarded Functions.

4.5 Functional Pattern-Matching

Most functional programming languages, such as Hope [Bailey, 1990], Haskell [Hudak and Wadler, 1988, Davie, 1992], Miranda [Turner, 1985, Turner, 1986], and Standard ML [Milner *et al.*, 1990], provide a mechanism called *argument pattern-matching.* Argument pattern-matching is a technique that provides the programmer with the ability to define a function based on the patterns of its parameters. Therefore, a function can be constructed by displaying all the patterns that its arguments may have and describing what value to return in each case. For example, let us consider the following ML function:

```
fun not true = false
|   not false = true;
```

The above function defines the characteristics of the boolean operator `not()` which takes a single argument. The definition states that if the argument matches the value true, then the action to take is to return false. Similarly if the argument matches the value false then the function will return true.

Patterns are not limited to constants, as in the case above. They can be expressions with variables. When a pattern expression matches an argument, the variables of the pattern are assigned the values in the actual arguments. These variables can then be used in the definition of the function [Wikström, 1987, Ullman, 1994]. The following shows an ML function that has expression patterns:

```
fun reverse nil = nil
|   reverse [item] = [item]
|   reverse [front :: rest] = (reverse rest) @ [front]
```

The function `reverse()` defines the procedure needed to reverse a list. If the list is empty, then the first pattern is matched and a nil is returned. If the list has only

one item then the second pattern is matched and the same list is returned, otherwise the third pattern is matched and the reversal is done according to its expression. Note that the operator @ is a built-in ML operator that perform the appending of two lists. The general template for stating function patterns is shown in Figure 4.7

The integration of guarded functions into the Electra language provides the programmer with a mechanism that helps in simulating the characteristics of pattern-matching. Figure 4.8 shows the ML function `merge()` that merges two lists. The function `merge()` employs pattern-matching to achieve its task. Figure 4.9 shows an Electra function that achieves the same task. The Electra version of the function `merge()` uses guarded functions to simulate the behavior of pattern-matching in the ML version of the `merge()` function.

```
fun  <function-name> <pattern-1> = <function-body-1>
|    <function-name> <pattern-2> = <function-body-2>
                .          .          .
                .          .          .
                .          .          .
|    <function-name> <pattern-N> = <function-body-N>;
```

Figure 4.7. Patterns in ML Functions.

Note that the ML patterns do not permit the appearance of relational or logical expressions. Thus the following is disallowed and would produce an error message.

```
fun foo ( x = y) = "Arguments are the same. "
```

```
fun merge(L1,nil) = L1
|   merge(nil,L2) = L2
|   merge(L1 as x::xs, L2 as y::ys) =
            if (x:int) < y then x::merge(xs,L2)
            else y::merge(L1,ys);
```

Figure 4.8. An ML Function That Merges Two Lists.

```
guarded function merge(L1, L2 : List)→List;
[ L2 = NIL];
begin
   return L1;
end;

guarded function merge(L1, L2 : List)→List;
[ L1 = NIL];
begin
   return L2;
end;

guarded function merge(L1, L2 : List)→List;
[ L1.head < L2.head ];
begin
   return List( L1.head, merge(L1.rest, L2));
end;

guarded function merge(L1, L2 : List)→List;
[ true ];
begin
   return List( L2.head, merge(L1, L2.rest ));
end;
```

Figure 4.9. Merging Lists in Electra Using Guarded Functions.

However, guard expressions in Electra would permit any boolean expression. Therefore, the following is valid in Electra:

```
guarded function foo(x,y :integer);
[x=y];
begin
   print("Arguments are the same.\n ");
end;
```

Disallowing relational and logical expressions to be included in a pattern expression forced the ML version of the function `merge()` to be limited to three patterns. On the other hand, the flexibility of the guard expressions permits the Electra version to achieve even finer decomposition of the problem by splitting the third pattern into two possible cases. Therefore, the Electra solution displays four possible cases instead of three.

Chapter 5
Feature Exclusion

This chapter begins by presenting the concepts of inheritance and inheritance hierarchies prior to introducing inheritance exceptions. It follows by listing the reasons that induce inheritance exceptions and shows approaches for eliminating them. Subsequently, it shows how Electra uses the construct of feature exclusion to represent inheritance exceptions. The chapter closes by describing the interaction between feature exclusion, inheritance, and constraints.

5.1 Inheritance, Inheritance Hierarchies, and Inheritance Exceptions

Inheritance, as defined in most object-oriented languages, typically extends the interface of a superclass by adding new features in the subclass definition. A superclass might have inherited its structure from another class therefore creating a hierarchy of superclasses and subclasses. Figure 5.1 shows a part of an inheritance hierarchy that was presented by Budd [Budd, 1991b]. This hierarchy of inheritances states relations between classes and superclasses. For example, it states that an artist is a human, a human is a mammal, a mammal is an animal, and an animal is a material object. This abstraction structure is known as an *inheritance hierarchy* or a taxonomic hierarchy. Inheritance, in this fashion, represents the *Is-a* relationship [Brachman, 1983, Budd, 1991b]. This means that it imposes an implicit assumption that every property attributed to a superclass is automatically carried by all of its subclasses. Therefore, since a shopkeeper is a human being then all the characteristics of a human are automatically assumed by all shopkeepers.

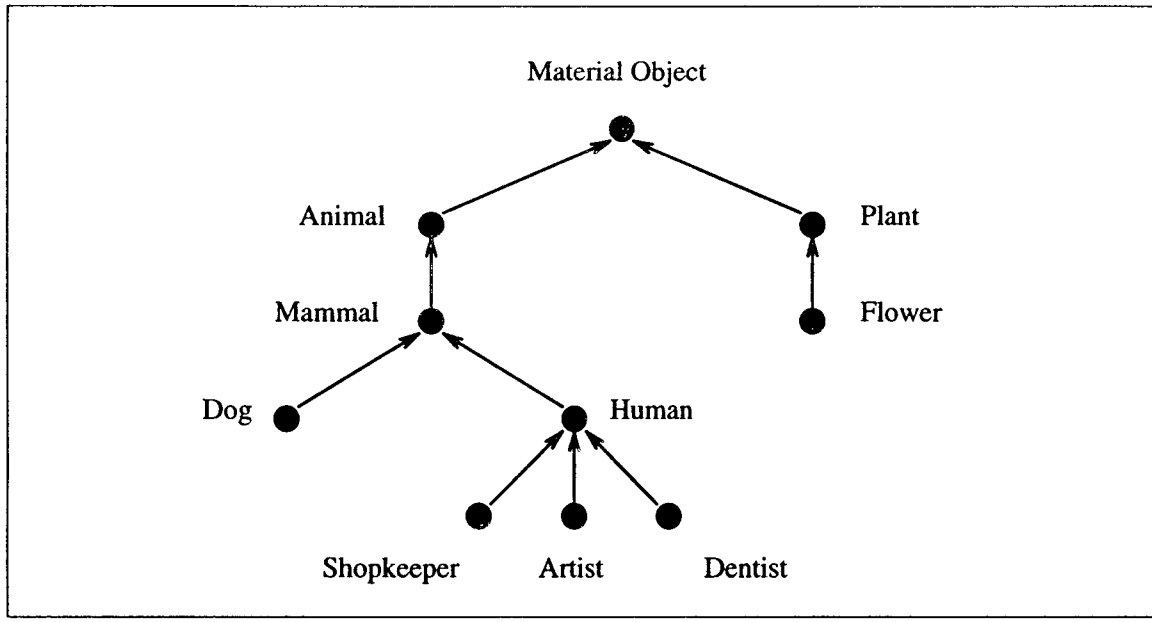

Figure 5.1. A Class Hierarchy for Various Material Objects.

Unfortunately, it is not always possible to impose such a rigid hierarchical structure on every real world situation. For instance, chickens, ostriches, and penguins are all birds that do not fly. A non-profit organization is an enterprise for which profit is a meaningless concept, and a parking structure is a building with no rooms. Other examples include countries with two capital cities or countries with no capital city. Therefore, it is difficult, or even impossible, to use this type of hierarchical inheritance in the construction of an abstraction that describes properties valid for all specializations. Some specializations represent objects with exceptional characteristics, for which it is not enough to extend the descriptions of the superclass but rather it is necessary to cancel some of its attributes [Brachman, 1985]

This limitation of the rigid structure of hierarchical inheritance is one of the most difficult problems in knowledge representation. Aspects of its theoretical characteristics have been a topic of great interest to many researchers [Etherington, 1987, Lenzerini *et al.*, 1991, Touretzky, 1986, Etherington and Reiter, 1983, Fox, 1979].

This characteristic of hierarchical inheritance is known as *partial inheritance* [Kim and Lochovsky, 1989], and in the field of knowledge representation it is called *inheritance exceptions.*

5.2 Reasons for the Rise of Inheritance Exceptions

When an exception appears in an inheritance hierarchy, in most cases the exception can be removed by restructuring the original inheritance hierarchy to produce a new hierarchy that does not contain any exceptions. However, this restructuring might not be possible due to limited access to source code such as the case in reusable software libraries. Even if the source code is fully accessible, this restructuring might not be desirable because the result might be an unnatural or excessively larger hierarchy. This section presents some causes that often induce the appearance of exceptions in an inheritance hierarchy. The need to exclude superclass features in the construction of a subclass may be a result of one of the following:

- Erroneous design of the inheritance hierarchy.
- The natural characteristics of the taxonomy at hand dictates the need for some cancellation of features anywhere in the hierarchy.
- The need to reuse existing classes that should not or cannot be modified.

5.2.1 Erroneous Design of Inheritance

Inheritance exceptions often appear in an inheritance hierarchy due to bad decisions during the design or enhancement of that hierarchy. Figure 5.2 shows the placement of a houseboat class in an inheritance hierarchy. Part (a) shows the Houseboat class as a subclass of class House. This hierarchical structure causes the appearance of inheritance exceptions because a house has certain characteristics that are not applicable to a houseboat. These characteristics include the nature of having permanent

location such as an address which is not valid in the case of a houseboat and therefore must be removed. Part (b) shows a more appropriate inheritance hierarchy that does not induce any exceptions. The creation of some additional classes in part (b) is justifiable because it resulted in a better reflection of the relations between a house and a houseboat.

5.2.2 The Natural Characteristics of Domain Knowledge

Every domain has its own agreed upon characteristics that cannot be altered to fit some logical taxonomy. For example when the word "bird" is mentioned, it conjures an image of a flying feathery creature. However, there are birds such as ostriches that do not fly. Part (a) of Figure 5.3 shows an inheritance hierarchy in which the class Ostrich inherits the properties of the class Bird. This seems reasonable since an ostrich is a bird. However, if the class Bird has the instance variable `Average_Flight_Speed` which indicates the average speed of a bird when flying, then this instance variable does not make any sense for instances of class Ostrich and therefore should not appear in it. Part (b) of the same figure shows a restructuring of the hierarchy by separating birds that fly from birds that do not fly. The problem with this restructuring is that it adds another class that makes the hierarchy slightly unnatural. The relation between domain knowledge and inheritance hierarchies is depicted in the following quote by David Touretzky[Touretzky, 1986]:

> Mandatory inheritance of properties is too inflexible for representing real-world knowledge. The real world contains exceptions to almost every generalization. Although most people's ideal elephant is a gray, four-legged, peanut-eating jungle dweller, there are non-gray elephants, three-legged elephants, elephants who do not eat peanuts, and elephants who do not live in jungles. If we require an abstraction to hold true for all members of a class, very few properties could be placed there.

This process of enforcing hierarchical inheritance over domain knowledge by creating additional classes is not always appealing especially when it results in the creation

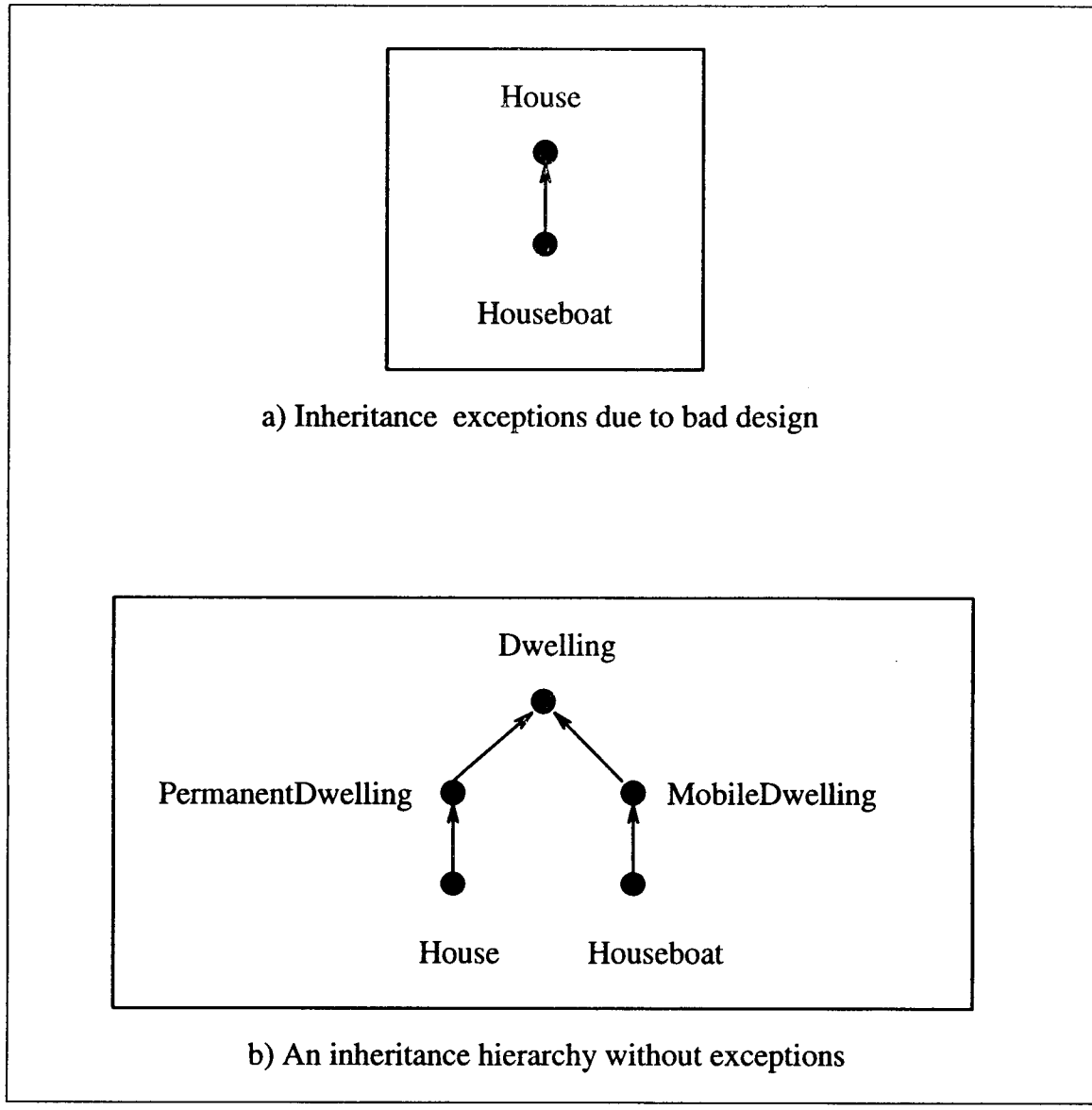

Figure 5.2. Redesigning a Class Hierarchy to Remove Exceptions.

of an unnatural hierarchical structure. Therefore, sometimes it is necessary to explicitly exclude certain features of a superclass from appearing in the structure of a subclass.

5.2.3 Reuse of Nonmodifiable Classes

Object-oriented techniques provide mechanisms for the separation between the interface and the implementation of software modules, as well as mechanisms for inheriting and expanding the characteristics of existing modules. Thus, using object-oriented approaches helps in the construction of easily reusable software components [Meyer, 1987]. The suitability of object-orientation for the creation of reusable components led Brad Cox to compare these components to integrated circuits and call them *software IC's* [Cox, 1986, Cox and Hunt, 1986, Ledbetter and Cox, 1985]. However, it is not feasible to create a library of reusable objects that cover every possible future requirement [Johnson and Rees, 1992]. Therefore, users are forced to alter or extend the behavior of the reusable classes to suit their needs.

For example, suppose the class BAG is given in a standard library. This class defines the structure and behavior of a bag – an unordered collection of objects with the possibility of having the same object occurring more than once. The method `Occurrences()` is provided in the protocol of that class and is used to evaluate the number of occurrences of a particular object in a bag. The concept of a set is similar to that of a bag except that every element in a set can occur at most once. Thus, a set is a special kind of bag. In order to create a set, one can reuse the provided class BAG and subclass it to create the subclass SET, overriding a number of the provided methods. The problem now is that the method `Occurrences()` is totally meaningless to a set. Therefore, conceptually it should not be part of the protocol of a set since it is not part of its behavior. Instead, class SET should have the predicate `Member()` that returns true if the specified element is in the set. This indicates that

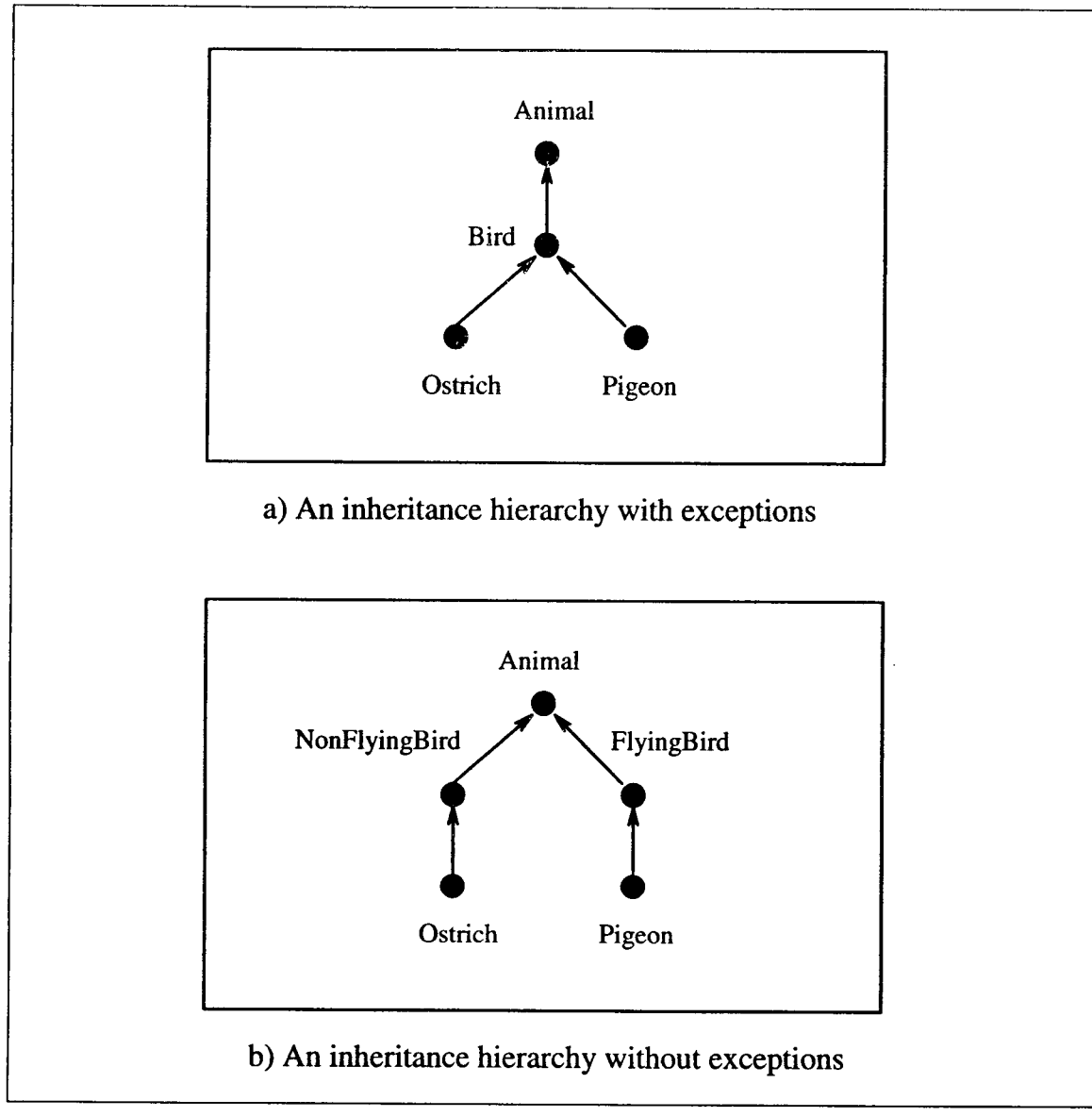

Figure 5.3. Restructuring a Class Hierarchy to Remove Exceptions.

sometimes it is necessary to exclude features when subclassing an existing class for reuse. Alan Snyder [Snyder, 1986b] has the following view regarding the exclusion of features for the sake of reuse:

> Most object-oriented languages promote inheritance as a technique for specialization and do not permit a class to "exclude" an inherited operation from its own external interface. However, if inheritance is viewed as an implementation technique, then excluding operations is both reasonable and useful.

5.3 Inheritance Exceptions in Electra

It is possible to express inheritance exceptions in Electra using a construct that we call *feature exclusion.* It is defined as a mechanism via which one can exclude features[1] of a superclass from appearing in the protocol of one of its subclasses. Similar mechanisms for removing a feature of a superclass when subclassing exist in other languages such as CommonObjects [Snyder, 1986a], and Trellis/Owl [Schaffert *et al.*, 1986].

In Electra, feature exclusion is expressed using the *exclude statement* inside the excluding class. Figure 5.4 shows the syntax of the exclude statement. This statement must appear before any declaration statement in the excluding class, and it lists all the features that are to be excluded. Once a feature is excluded, it is no longer visible as a feature in the excluding class or its subclasses. An excluded feature is not removed but merely masked. When a class that excludes some features is instantiated then every excluded instance variable is assigned the value NIL, and every excluded method is assigned a standard excluded method body.

Figure 5.5 shows an example of how features are excluded using the exclude statement. Class `Bird` declares the instance variable `Average_Flight_Speed` which is excluded in the subclass `Ostrich`. Due to this exclusion the instance

[1]The term *feature* is borrowed from Eiffel [Meyer, 1993] where it is used to refer to both methods and instance variables.

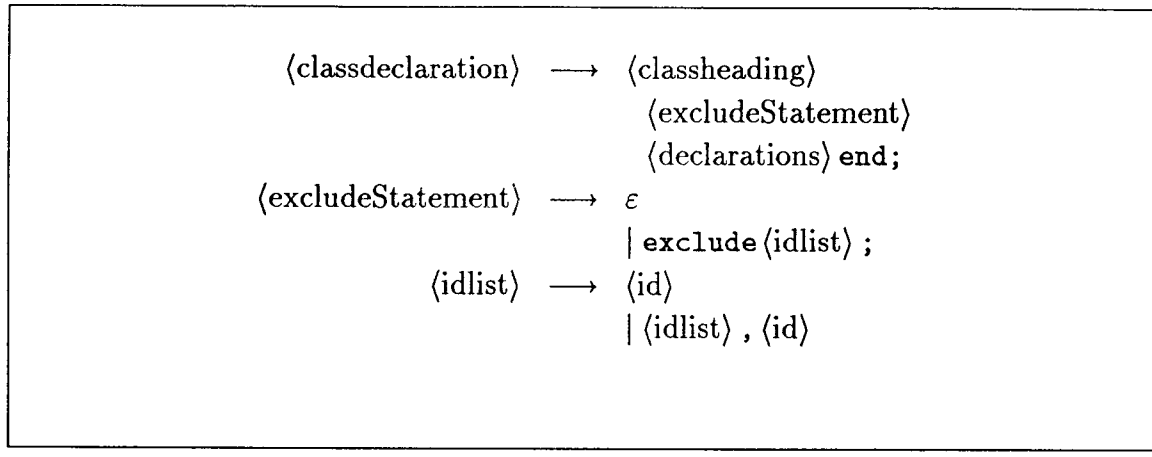

Figure 5.4. Syntax of Feature Exclusion.

variable `Average_Flight_Speed` is no longer considered part of the protocol of the class `Ostrich`. When class `Ostrich` is instantiated, the instance variable `Average_Flight_Speed` is assigned the value NIL by the compiler.

```
class Bird;
     Average_Flight_Speed : real;
     ...
end;

class Ostrich of Bird;
  exclude Average_Flight_Speed;
     ...
end;
```

Figure 5.5. An Example of Feature Exclusion.

5.4 Exclusion Versus Overriding

One way of excluding undesirable inherited methods is by overriding them with a segment of code that returns a diagnostic message when the excluded method is called, indicating that this method is not part of the receiver's protocol. One advantage of overriding undesirable methods is that overriding adheres to a typing system principle called *the principle of substitutability.* The principle of substitutability states that an instance of a subtype can always be used in any context in which an instance of a supertype is expected [Wegner and Zdonik, 1988]. Another advantage of overriding is that it does not generate any conflicts with dynamic binding. However, a disadvantage is the creation of meaningless parts in the protocol of the subclass. In addition, the purpose of this overriding is to exclude a feature, not to alter it. Thus, it is preferable to equip the language with a construct that conveys the user's intent of excluding undesirable inherited features and let the compiler automatically generate the needed overrides. For example, the definition of class SET should look something like:

```
class SET of BAG;
  exclude Occurrences;
  ...
end;
```

If the message `Occurrences()` is sent to an instance of type SET then an error message will be generated indicating that `Occurrences()` is not part of SET's interface. The statement:

```
exclude Occurrences;
```

can be viewed as expressing the relation:

$$\boxed{\text{SET} = \text{BAG} - \text{Occurrences}() + \ldots}$$

This relation states that a set is a bag with the feature Occurrences removed, and that there could be some more features. The system maintains this relation by providing the appropriate overrides. This approach of providing constructs for explicit

specification of exclusion of features is both cleaner and clearer. It makes it easier to map the specification of the simulated model. Another disadvantage of overriding is that it cannot be used to exclude instance variables.

5.5 The Interaction Between Inheritance and Feature Exclusion

Most object-oriented languages permit a subclass to extend the definition of its superclass and prohibit the removal of any features of the superclass when subclassing. The reason for this restriction is that these languages do not distinguish between inheritance and subtyping. In a typing system, T_1 is a subtype of T_2 if every instance of T_1 is also an instance of T_2 [Cardelli and Wegner, 1985]. This implies that an instance of T_1 can be used whenever an instance of T_2 is expected. This is what is called the principle of substitutability.

An object-oriented language that views subclassing as subtyping cannot permit the removal of superclass features in a subclass because that would violate the principle of substitutability. To remedy the situation, some languages such as Sather [Szypersky *et al.*, 1993] separate the type/subtype hierarchy from the class/subclass hierarchy. In Electra excluded features are not removed but rather masked and assigned the generic value NIL. The following refers to the classes `Bird` and `Ostrich` that were presented earlier. Let us suppose that we have the following declarations:

```
b : Bird;
o : Ostrich;
...
o := Ostrich();    { Instantiate class Ostrich }
...
b := o;
```

The definition of `Ostrich` excludes the feature `Average_Flight_Speed`. Therefore, accessing the variable `o.Average_Flight_Speed` should produce an error message during compile time. The feature `Average_Flight_Speed` is not removed from the

definition of class `Ostrich`, it is merely inaccessible as a feature of that class. After assigning the variable `o` of type `Ostrich` to the variable `b` of type `Bird`, the feature can be accessed as `b.Average_Flight_Speed` and it has an undefined value (i.e. the value NIL).

5.6 The Interaction Between Constraints and Feature Exclusion

A constraint that is declared by a class is inherited by all of its subclasses. A subclass is permitted to exclude any of the features that it inherits from its superclass. A reasonable question to ask is "What happens to a constraint that is inherited by a subclass that excludes one or more of that constraint's participants?" Since an excluded feature is not removed but merely masked and assigned the the value NIL and since a constraint does not get activated until all of its participants are defined then an inherited constraint that has at least one excluded participant will never be activated.

It is important to note that the current definition of the language Electra prohibits the exclusion of any variable that participate in a satisfiable constraint. The reason for this restriction is that every participant of a satisfiable constraint is maintained by the underlying satisfier.

Chapter 6
Implementation Approaches and the Electra Compiler

This chapter presents the approaches taken to implement Electra's added constructs. It begins by covering how fixable and satisfiable constraints are implemented. After that it presents the approach taken to implement guarded functions which are used to express condition-based dispatching. A discussion of the implementation of feature exclusion is also presented. The chapter closes by illustrating the differences between Leda and Electra compilers.

6.1 Implementation of Constraints

Chapter three presents the syntax and semantics of both fixable and satisfiable constraints. This section describes the implementation of fixable and satisfiable constraints in the Electra compiler. It begins by presenting data structures that are common to both types of constraints then it covers the implementation aspects of each type separately.

Two new lists were added to the symbol table of every scope level in order to keep track of the declared constraints and their participating variables. The first is called *constraint list* and the second is *constrained variables list.* The constraint list is composed of constraint records where each record represents a constraint, and the constrained variables list contains constrained variable records that keep information relevant to every constrained variable. Figure 6.1 illustrates the structure of constraint records and constrained variable records. Whenever a constraint is encountered a constraint record is created for it containing its name, type of solvability (i.e. fixable or satisfiable), and appropriate pointers to its participants. A

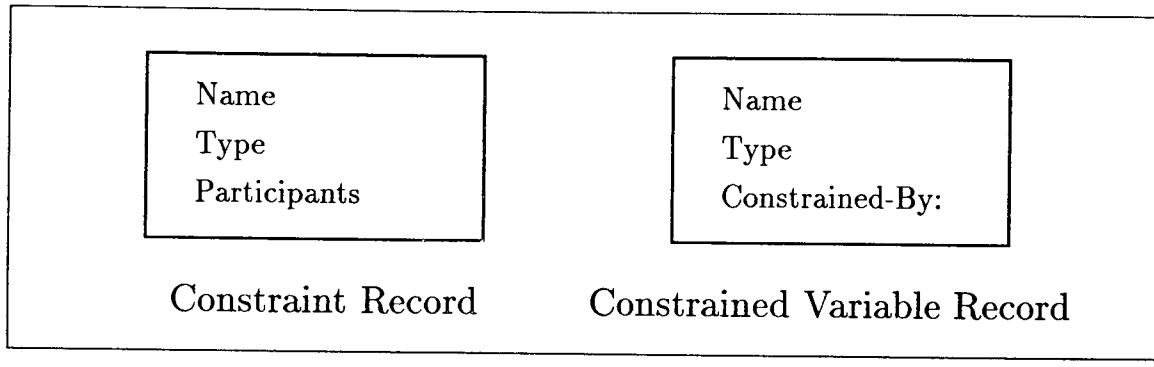

Figure 6.1. Constraint Records and Constrained Variable Records.

constrained variable record is also created for every participant. That record contains the participant's name, type of solvability, and pointers to the constraints it is participating in. The Electra language supports two types of constraints: fixable constraints and satisfiable constraints. In the following paragraphs, we will describe the approaches taken to implement each type.

6.1.1 Implementation of Fixable Constraints

A fixable constraint contains its own fix. The fix section of a constraint is composed of a set of nonreturn statements. This section of code is invoked whenever the constraint is broken. A check is automatically added at the bottom of the fix to verify that at the end of the fix the assertion is satisfied. A fixable constraint gets activated when all its participants are defined. A variable that participates in a fixable constraint is not permitted to participate in any other constraint.

In terms of implementation, fixable constraints can be viewed as *procedural attachments*. The concept of procedural attachments [Brachman, 1979] has its roots in frame languages like KRL [Bobrow and Winograd, 1977], FRL [Roberts and Goldstein, 1977], and KL-One [Brachman, 1979, Brachman, 1977]. This concept is also known as *access oriented programming* [Stafik *et al.*, 1986]. The idea is that a generic

function containing both the assertion and the fix of the constraint is created and attached to the constrained object in such a way that it is invoked whenever the constrained object is accessed for an update. The constraint condition is then checked. If the condition is not met then the constraint is not being maintained and, as a reaction, the fix part is executed.

Every fixable constraint is implemented as a subclass of a predefined class called `FixableConstraints` which has a structure that is depicted by Figure 6.2. The class `FixableConstraints` is composed of two instance variables and five member functions. The instance variables `breaker` and `oldVal` serve as repositories for the participant that violated the constraint and its old value respectively. The member function `brokenBy()` takes a participant as an argument and returns a boolean value indicating whether the given participant is the one that violated the constraint. The function `old()` takes no arguments and returns the old value of the participant that violated the constraint if it has an old value otherwise it returns NIL. The third function `checkAssertion()` is overridden during the parsing of the actual constraint. It is constructed in such a way that it will return true if all the participants of its constraint are defined and the assertion of that constraint is violated otherwise it returns false. The function `fixPart()` is another function that is overridden during the processing of its constraint. It contains the body of the fix part of its constraint, as indicated by its name. The most important part of the satisfaction process is performed by the function `checkValidity()`. It is invoked whenever a participant is updated. Its job is to check if the assertion is broken. If so, then it invokes the fix and checks the assertion again after the invocation of the fix to be sure that the constraint has been satisfied. If the assertion is still invalid then the program is terminated with an error message stating that the fix did not satisfy the assertion.

A simple example of a fixable constraint is shown in Figure 6.3 that will help in explaining the process of activation, maintenance, and satisfaction of a fixable constraint. Once the constraint `foo` is parsed, a constraint record is created for it and is inserted into the constraint list of its scope. Also, constrained variable records

```
class FixableConstraints;
var
   breaker, oldVal : integer;

   function brokenBy( x : integer )→boolean;
   begin
      if (x == breaker ) then
         return true
      else
         return false;
   end;

   function old()→integer;
   begin
      if defined(oldVal) then
          return oldVal
      else
          return NIL;
   end;

   function checkAssertion ()→ boolean;
   begin  end;

   function fixPart();
   begin  end;

   function checkValidity();
   begin
       if (checkAssertion() ) then
          begin
            fixPart();
            if (checkAssertion() ) then
             begin
                print("The fix does not satisfy the assertion. \n");
                cfunction Leda_prog_exit();
             end;
          end;
   end;

end; { of class FixableConstraints declaration }
```

Figure 6.2. Fixable Constraints.

```
var
   d, e : integer;

constraint foo;
  assert
       d = e + 1;
  fix
     if brokenBy( d ) then
        e := d -1
     else
        d := e + 1;
end;
```

Figure 6.3. Example of a Fixable Constraint.

are created for the variables `d` and `e`. A subclass of the class `FixableConstraints` is created with the name `constraint_foo`. The new subclass overrides the functions `checkAssertion()` and `fixPart()`. The new bodies for the overridden functions are shown in Figure 6.4. The final step of initializing and activating this constraint is to instantiate the class `constraint_foo` with an instance named `foo`. The process of initializing and activating any fixable constraint can be summarized in the following steps:

1. Create a constraint record for the new constraint and insert it into the constraint list.

2. Create a constrained variable record for every participant and store the created records in the constrained variable list.

3. Create a subclass of the class `FixableConstraints` to represent the new constraint.

4. Override the functions `checkAssertion()` and `fixPart()` with the appropriate bodies.

```
function checkAssertion ()→ boolean;
begin
   if defined ( e ) & defined( d ) & ( ~( d = e + 1) ) then
      return true
   else
      return false;
end;

function fixPart();
begin
  if brokenBy( d ) then
     e := d −1
  else
     d := e + 1;
end;
```

Figure 6.4. Overriding of Some Functions of the Base Constraint Class.

5. Instantiate the newly created subclass.

If a variable that is constrained by a fixable constraint is to be updated then a search is performed to locate the constraint that is constraining it. Once that constraint is found then its instance variables are set to the proper values and its member function `checkValidity()` is invoked. For instance, in the previous example, if the variable `d` were to be updated then the following operations are performed:

1. Search for the constraint `foo`.

2. Set `foo.oldVal` to the old value of `d`.

3. Set `foo.breaker` to the new value of `d`.

4. Invoke the function `foo.checkValidity()`.

The function `foo.checkValidity()` will invoke the fix if necessary.

6.1.2 Implementation of Satisfiable Constraints

A satisfiable constraint is a constraint that is totally maintained by the constraint solver. Every variable that participates in a satisfiable constraint is represented by a single copy that is kept by the solver. The Electra compiler is separate from the solver and has no knowledge of its characteristics. All the communication between the compiler and the constraint satisfier is done through an interface called the *compiler-solver interface.* This approach of separating the compiler from the constraint satisfier makes it possible to use different solvers depending on the nature of the declared constraints and the capabilities of the available solvers. Figure 6.5 illustrates the interface between the compiler and the satisfier.

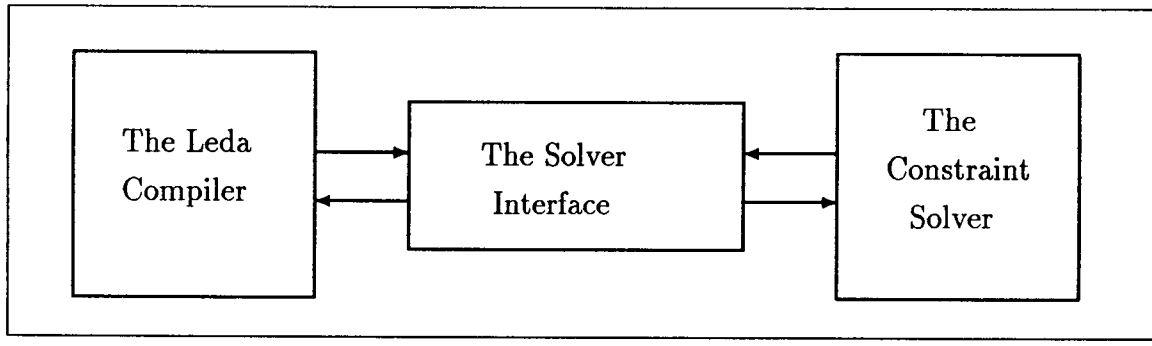

Figure 6.5. The Compiler-Solver Interface.

When a satisfiable constraint is encountered then it is the job of the interface to check that the constraint assertion is understandable by the satisfier and that all the participating variables belong to domains that can be handled by the current satisfier. Once the assertion and the respective domains are deemed to be valid, then the interface passes the constraint's name, its strength, its assertion, and its participants to the solver. Every access or update to any of the participating variables

is converted to an Electra cfunction call which is part of the interface to the solver. The interface provides the following services to the Leda compiler:

1. Insert variable into the solver's symbol table.
2. Remove variable from the solver's symbol table.
3. Insert constraint into the solver's symbol table.
4. Remove constraint from the solver's symbol table.
5. Enforce constraint.
6. Relax constraint.
7. Get value of a variable.
8. Set value of a variable.

6.2 Implementation of Guarded Functions

The syntax and semantics of guarded functions are presented in chapter four. The approach taken for the implementation of guarded functions in Electra is to create an enclosing function and view all instance functions as subfunctions of the enclosing function. The body of the enclosing function is composed of a nested if-then-else statement where the conditions are the guard expressions and the statements are calls to the corresponding instance functions.

Figure 6.6 shows an Electra program that has the two functions `foo()` and `bar()`. The function `bar()` is a guarded function and there are three instances of it. The instance functions have the guard expressions `x > y`, `x = y`, and `x < y`, respectively. Figure 6.7 shows the same example with the guarded functions transformed into a single enclosing function with all the instance functions appearing as subfunctions of that enclosing function. The subfunctions are given new names according to their order of appearance. Note that the function `foo()` remains intact

since it is not a guarded function. The declarations of the instance functions do not have to be grouped together but rather can be interleaved by other declarations such as the case of having the declaration of the function `foo()` appears between the first and second instances of the guarded function `bar()`.

```
guarded function bar(x, y : integer);
[ x > y ]
begin
    .... body of first instance of the guarded function bar ....
end;

function foo();
begin
      .... body of function foo ....
end;

guarded function bar(x, y : integer);
[ x = y ]
begin
    .... body of second instance of the guarded function bar ....
end;

guarded function bar(x, y : integer);
[ x < y ]
begin
    .... body of third instance of the guarded function bar ....
end;

begin
    .... body of the main program ...
end;
```

Figure 6.6. Guarded Functions Implementations: An Example.

```
function foo();
begin
     .... body of function foo ....
end;
function bar(x, y : integer);

     function bar_1_();
     begin
        .... body of first instance of the guarded function bar ....
     end;

     function bar_2_();
     begin
        .... body of second instance of the guarded function bar ....
     end;

     function bar_3_();
     begin
        .... body of third instance of the guarded function bar ....
     end;

begin
   if ( x > y ) then
       bar_1_();
   else if ( x = y ) then
            bar_2_();
       else if ( x < y ) then
                bar_3_();
end;

begin
   .... body of the main program ...
end;
```

Figure 6.7. Converting Guarded Functions to Nested Regular Functions.

6.2.1 Managing Guarded Functions

Every instance function is represented by a record called the *instance guarded function record.* This record contains the new name given to the instance and the instance's guard expression. Whenever an enclosing function is created a corresponding *enclosing function record* is instantiated for that function. This type of record keeps information related to the name of the represented guarded function, the number of instance functions it has, a list of the names of these instance functions and their guard expressions, and some other bookkeeping information. Figure 6.8 illustrates the structure of instance guarded function records and enclosing function records.

A new entry is added to the symbol table of every scoping level. This entry is called *guardsInfo.* It contains a pointer to a list containing information about the guarded functions within that scope. Each guarded function is represented by an enclosing function record that is inserted into the guardsInfo list in its scope level.

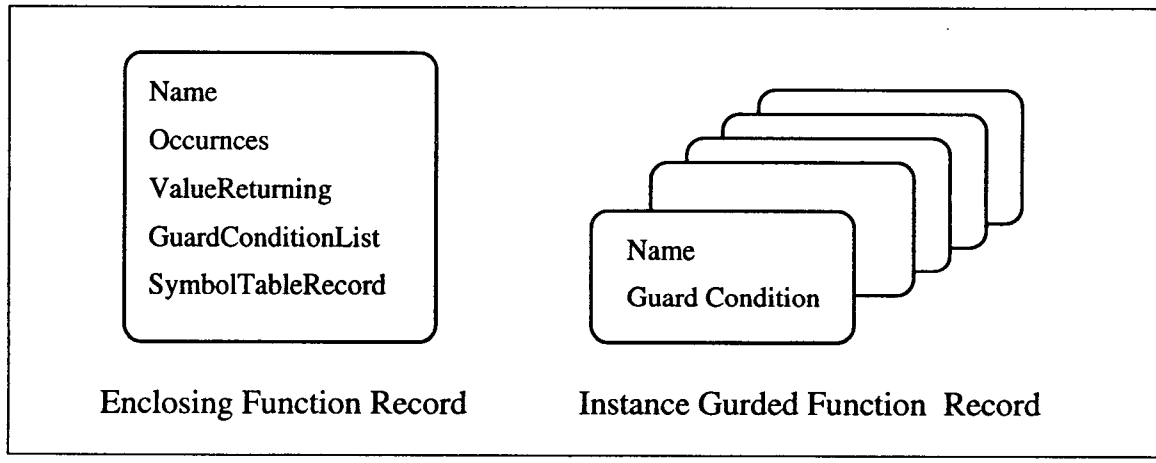

Figure 6.8. Managing Guarded Functions.

6.2.2 The Transformation Process

This section describes the process of transforming guarded functions into nested regular functions. When an instance of the guarded function `bar()` is encountered a check is made to see if it is the first instance of that guarded function. If it is, then an enclosing function is created with the name `bar()`. An enclosing function record is created for it and inserted into the guardsInfo list of the containing scope. The encountered instance function is given the new name `bar__1__()` and made as a subfunction of the enclosing function. The name of this function (i.e. `bar__1__()`) and its guard expression are added to the enclosing function record of `bar()`. If the encountered instance is not the first instance, then the existing enclosing function record for `bar()` is fetched and the encountered instance function is given a new name and inserted as a subfunction of `bar()`. The enclosing function `bar()` still has an empty body. Its body will be created just before the enclosing scope is exited. All the information needed to create its body can be found in its record that resides in the guardsInfo list. Figure 6.9 shows the symbol tables for the three scoping levels: global, function `foo()`, and function `bar()`. The symbol table for the function `foo()` shows that this function has no subfunctions and its guardsInfo list is empty since it does not contain any guarded functions. The symbol table for the function `bar()` shows that it has three subfunctions: `bar__1__()`, `bar__2__()`, and `bar__3__()`. Figure 6.10 takes a closer look at the symbol table of the global scope level. It shows that the global scope has two functions. The first is `foo()` and it is not guarded, while the second is `bar()` which is guarded, The guardsInfo list of the global scope has one entry related to the only guarded function in that scope. This entry shows that this guarded function has three instances and it gives their names and their guard conditions. This information is used to create the nested if-then-else statement that constitutes the body of the function `bar()`.

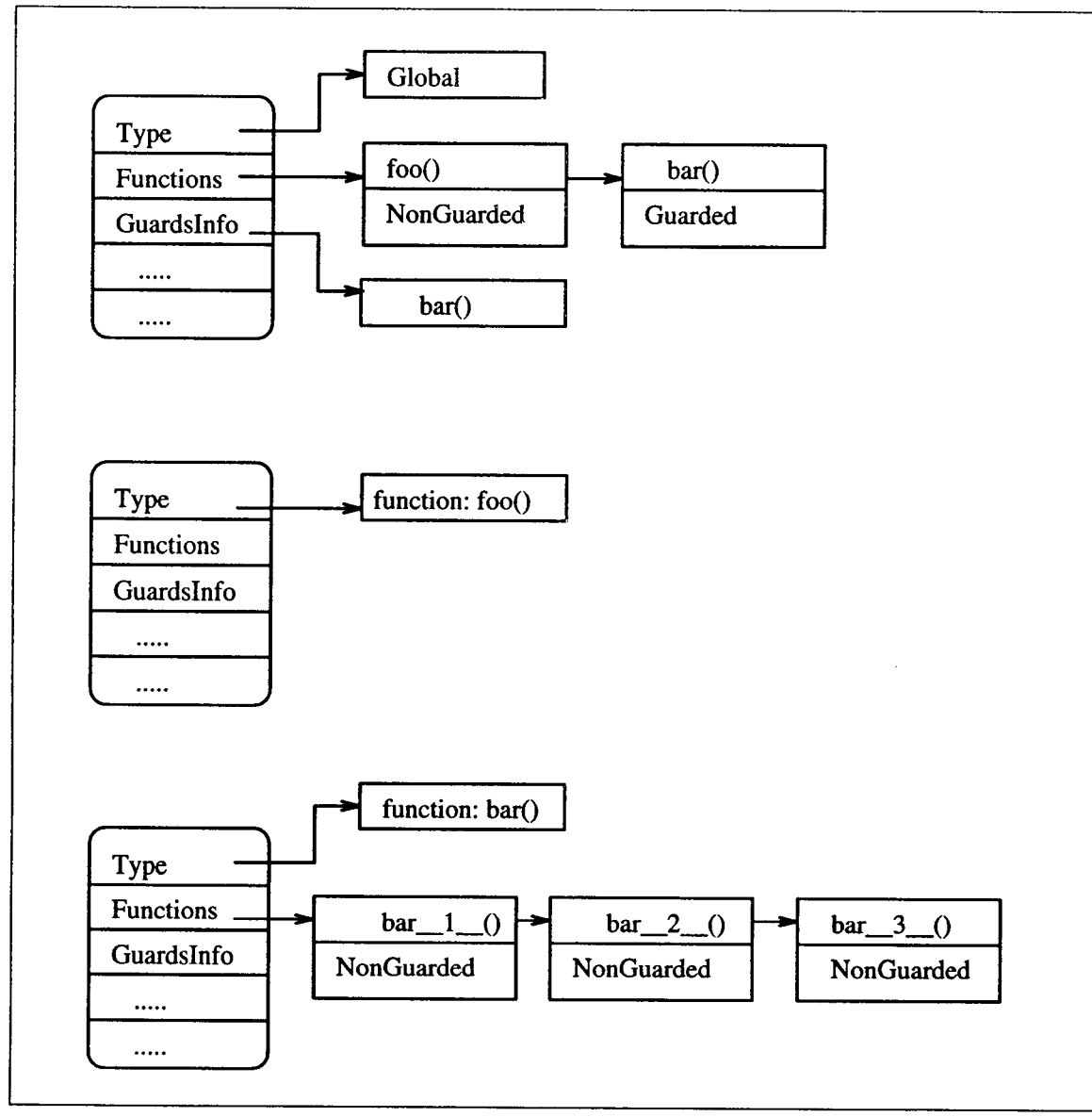

Figure 6.9. Symbol Table Records.

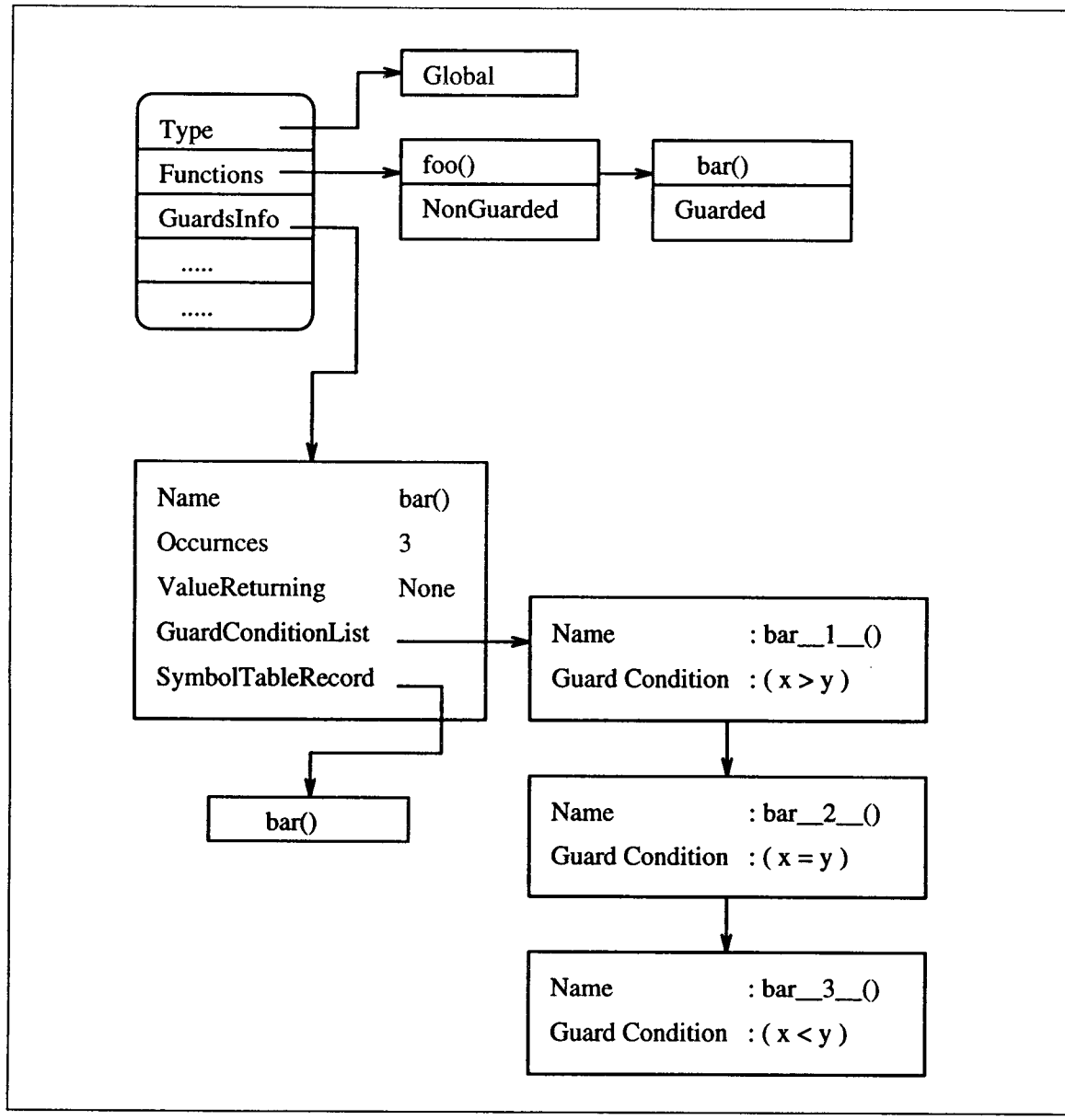

Figure 6.10. Symbol Table Record of the Global Scope.

6.3 Implementation of Feature Exclusion

The behavior, syntax, and semantics of the exclude statement is presented in chapter five. Every feature that appears in an exclude statement is marked as excluded. An excluded instance variable cannot be accessed in the excluding class or its subclasses. Since a default value of NIL is assigned to every variable upon creation by the compiler, and since an excluded instance variable cannot be updated then every excluded instance variable has the value NIL. The body of an excluded method is replaced by a call to the standard method `excludedMethodBody()` passing it the name of the excluded method. Figure 6.11 lists the method `excludedMethod Body()`. The job of this method is to indicate that an excluded method has been called and terminate the program. The actions of this method can be altered to reflect the programmer's wishes. If a value-returning method is excluded then a NIL is returned.

```
function excludedMethodBody(s:string);
var
   temp : boolean;
begin
   print("An attempt to call an excluded method: " );
   print(s);
   print("\n");
   temp :=  cfunction
            Leda_prog_exit()→boolean;
end;
```

Figure 6.11. A Standard Method that Replaces Every Excluded Method.

6.4 The Electra Language Compiler

A compiler for the language Electra has been built by integrating the above described implementation approaches into the latest Leda compiler. Electra's compiler supports all the added features as described in this dissertation. The only limitation is that it does not support class constraints. The reason for this limitation is that the current implementation of Leda does not support the declaration of a class inside another class.

Table 6.1 shows what the Electra's enhancements added to the Leda compiler in term of size, production rules, keywords, and abstraction constructs. The table

Table 6.1. Differences Between Leda and Electra Compilers.

Language or Compiler Aspect	Leda	Electra	Difference
Run-time system size	219795	278528	58733
Number of non-terminal symbols	48	67	19
Number of keywords	22	31	9
Number of abstraction constructs	3	4	1

indicates that the size of the compiler has increased by about one fourth which is reasonable since Electra adds a new paradigm to the four paradigms that are supported by Leda. The larger proportional increase in the number of non-terminal symbols and keywords is due to the fact that Electra supports two types of constraints in addition to its enhancements of the functional and object-oriented paradigms. In terms of keywords, Leda has seventeen additional keywords that represent the textual names for operator symbols. The abstraction constructs that are supported by

Leda are: the function abstraction, the relation abstraction, and the class abstraction. Electra supports one additional abstraction construct which is the constraint abstraction.

The Leda compiler groups all predefined types, classes, functions, and variables into an include file called `std.led` that can be customized by programmers. The Electra compiler takes the same approach by grouping all predefined entities that are relevant to constraints, condition-based dispatching, and feature exclusion into a single include file called `constraints.led`.

Chapter 7
Advantages and Examples

This chapter examines some of the advantages and capabilities that can be attributed to constraints, condition-based dispatching, and feature exclusion. It begins by showing how constraints can be used in implementing the technique of enforced reevaluation and how their declarative style can help in hiding the intricate details of search. After that, it illustrates how condition-based dispatching can be a great aid in the direct mapping of problem specification into Electra code. It then shows how feature exclusion and constraints can help in increasing the degree of software reuse that is normally provided by inheritance. An illustration is provided as to how the expressiveness of data abstraction and the versatility of type extensions can benefit from the availability of the constraint construct. The chapter closes by showing how constraints can help in insuring data integrity and validating code correctness.

7.1 Constraints and the Enforced Reevaluation

It is often necessary to create a linkage between two entities in such a way that if one is changed then the change is reflected on the other. This means that when one entity changes a reevaluation is enforced upon the other one. This section presents two examples that shows how enforced reevaluation is used in implementing scrollable windows and screen savers utilizing the constraint construct.

7.1.1 Implementing a Scrollable Window

A scrollable window is a window that is normally used to view a text file by scrolling it up or down. Each scrollable window has one or more scroll bars associated with it.

Figure 7.1 shows a scrollable window and a scroll bar. The scroll bar is composed of two buttons and a thumb. The scrollable window maintains two constraints between the displayed text and the orientation of the scroll bar. These constraints are as follows:

1. The first constraint states that the relation between the scrolled portion of the viewed file and the total length of the file must be the same as the relation between the length of the part of the scroll bar above the thumb and the whole length of the scroll bar. This constraint can be expressed using the following equation:

$$\frac{sp}{fl} = \frac{spb}{bl}$$

2. The second constraint states that the relation between the length of the viewed part of the text file and the total length of the file must be the same as the relation between the length of the thumb and the whole length of the scroll bar. The second constraint can be expressed using the equation:

$$\frac{wl}{fl} = \frac{tl}{bl}$$

Table 7.1 lists the variables used in the two equations that represent the constraints maintained by the scrollable window along with a description of their purpose, and Table 7.2 displays the specifications of a scrollable window by listing the possible actions and their associated reactions.

A reasonable way to correctly implement a scrollable window is to create an accessor function for each variable involved in the two equations. The need for the accessor functions is justified because an update to any variable requires a check to validate the integrity of the two equations. A better approach is to use the technique of enforced reevaluation by implementing the scrollable window using constraints.

Figure 7.2 shows the specification for an Electra constraint that expresses the behavior of a scrollable window. The assertion expression of the constraint `ScrollableWindow` lists the two equations that express the relations that must be

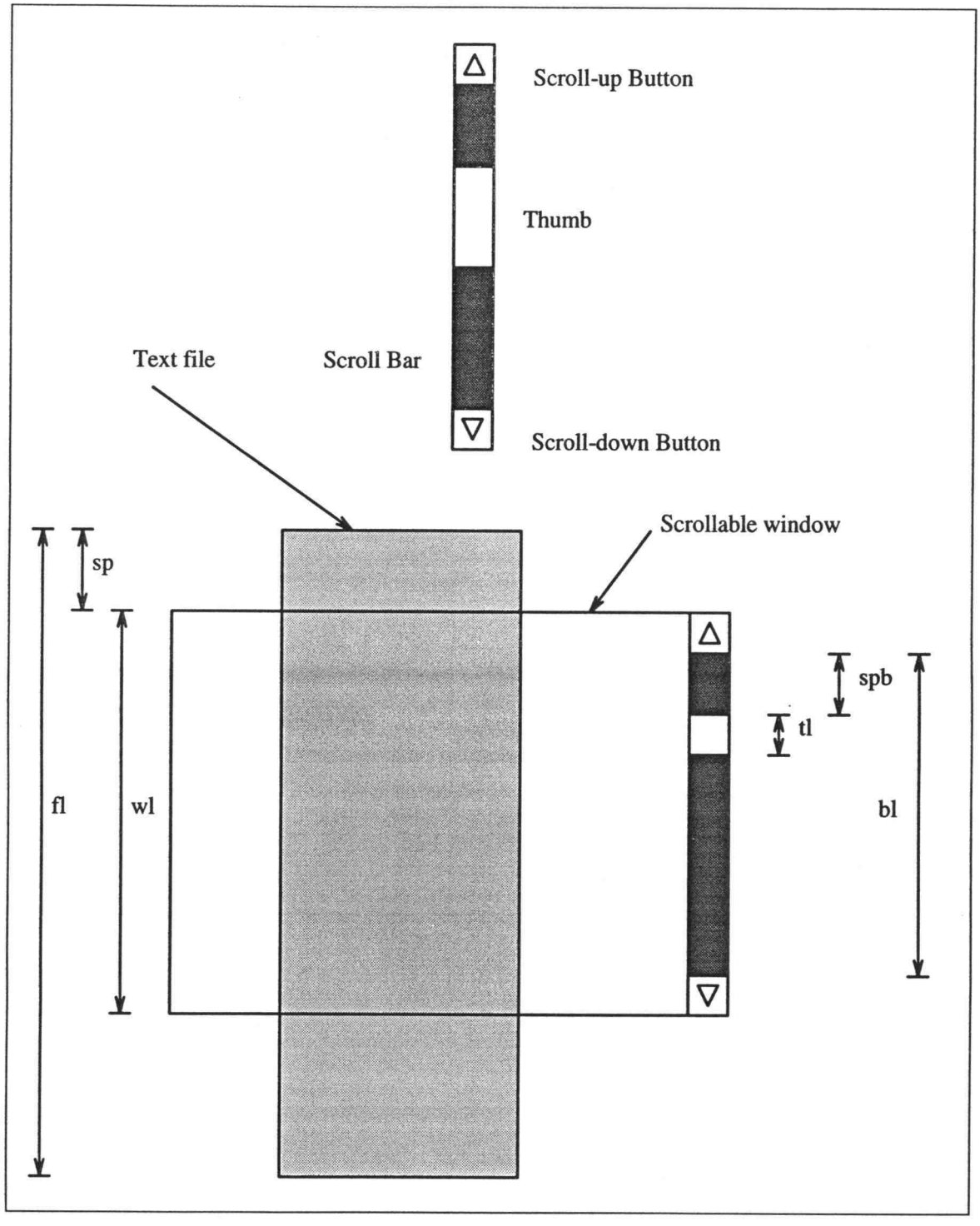

Figure 7.1. Characteristics of Scrollable Windows.

Table 7.1. Variables and Their Meanings: This table lists the variables used in the scrollable window example along with their meaning.

Variable Name	Description
sp	Length of the scrolled part of the viewed file.
fl	Total length of the viewed file.
wl	Length of the scrollable window.
bl	Length of the scroll bar.
tl	Length of the thumb.
spb	Length of the part of the scroll bar above the thumb.

Table 7.2. The Specification of a Scrollable Window: This table shows the specifications of a scrollable window by listing the possible actions and their associated reactions.

External Action	Reaction of Scrollable Window
File length increased	Move thumb up and reduce its length
File length decreased	Move thumb down and increase its length
Thumb moved down	Increase scroll part of file Update window
Thumb moved up	Decrease scroll part of file Update window
Scroll part of file increased	Move thumb down
Scroll part of file decreased	Move thumb up
Window length increased	Increase length of thumb
Window length decreased	Decrease length of thumb

```
constraint ScrollableWindow;
   assert
     ( (sp/fl = spb/bl) &
       (wl/fl = tl/bl)  ) ;
   fix
    begin
       if brokenBy( sp ) then
              spb := (sp * bl)/fl;
       else if brokenBy( spb ) then
              sp := ( spb * fl )/bl;
       else if brokenBy( fl ) then
              begin
                  spb := ( sp * bl)/fl;
                  tl := ( wl * bl )/fl;
              end;
       else if brokenBy( wl ) then
              tl := ( wl * bl )/fl;

       RefreshWindow( CurrentWindow );
    end;
end;
```

Figure 7.2. Implementing a Scrollable Window Using Constraints.

maintained by the scrollable window. The fix part states the actions that are to be taken when the involved variables are changed. There are four possible actions that can result in violating the constraint `ScrollableWindow`. The following list summarizes the behavior of the fix part by listing its reaction to each of these four possible actions.

1. *Scrolled part of file has changed*

   Update the location of the thumb.

2. *The location of the thumb has changed*

   Update the length of the scrolled part of the file.

3. *The length of the file has changed*

   Update the location of the thumb and its length.

4. *The length of the window has changed*

   Update the length of the thumb.

Once the appropriate action is taken, the current window is redisplayed.

7.1.2 Implementing a Screen Saver

A screen saver is a software utility that has the advantage of prolonging the life-span of a monitor screen by blanking it whenever it is not being used. Upon activation, a screen saver continuously monitors certain system events such as keyboard strokes, mouse operations, or screen updates. Once a certain interval of time is elapsed without encountering any of the above events, the screen saver blanks the screen. If one of the aforementioned events occurs while the screen is blanked by the screen saver, the screen is refreshed and the screen saver continues its monitoring of events. Figure 7.3 shows an Electra implementation of a screen saver using the constraint `ScreenSaver`.

The assert expression of the constraint `ScreenSaver` states that the constraint is satisfied as long as no new event has taken place and the time limit has not been exceeded since the last event. The variable **event** is a boolean that is set to true by the system whenever a keystroke, mouse operation, or screen update is to take place. If an event were to occur or if the time limit where to expire then the fix would be invoked. The fix part checks for the reason the constraint was violated and performs the appropriate reactions. If the violation was due to an event then the action to take is to refresh the screen if it is blanked and to update the starting time. If the violation occurred due to time limit expiration or a change in the length of the time limit then the screen is blanked and the starting time is set to a large value.

```
constraint ScreenSaver;
      assert
          ( ~event ) &
          ( clock < ( startTime + interval ));
      fix
        if brokenBy( event ) then
            begin
               if ( activeSS ) then
                  begin
                     activeSS := false;
                     refreshScreen();
                  end;
                startTime := clock;
            end;
         else if (brokenBy(clock) | brokenBy(interval)) then
            begin
               activeSS := true;
               blankScreen();
                  {Set startTine to the largest possible integer.}
               startTime := MaxInt - interval;
            end;
      end;
```

Figure 7.3. Implementing a Screen Saver Using Constraints.

A constraint is a natural approach for expressing the technique of enforced reevaluation. Enforced reevaluation can be used for stating the specification of many problems in diverse domains such as simulation of physical systems and user interfaces.

7.2 Constraints and Search

Chapter two showed how a problem such as the map-coloring problem can be converted into a constraint-satisfaction problem by decomposing its specification into a set of constraints. The map-coloring problem can easily be solved by feeding the obtained constraints to a constraint-satisfier. This shows how the declarative nature

```
class Queen
  x, y : integer;
  constraint Location;
    assert
        forall q in instances_of( Queen )
            if q <> self then
              q.x <> x and q.y <> y
              and  q.x + x <> q.y +y
              and  q.x − x <> q.y −y;
  end;
end;
```

Figure 7.4. The Definition of the Queen Class With its Location Constraint.

of constraints can help in relieving the programmer from worrying about the intricate details of the search process since that task is delegated to the constraint-satisfier.

This section shows another example of how constraints can help in hiding the search implementation. A good example of a search problem is the 8-Queens problem. The object of the 8-Queens problem is to place eight queens on a chess board in such a way that none of the queens can attack any of the others. In other words, in order to satisfy the requirements of the problem, no two queens can share a row, a column, or a diagonal. Cull and Pandey present further discussion and very comprehensive bibliography related to the N-Queen problem [Cull and Pandey, 1994]. Figure 7.4 shows how the specifications of the problem (i.e. the restrictions imposed on every queen) can be declared as a constraint inside the class `Queen`. It lists the declaration of the class `Queen` with the x and y coordinates of its location. The constraint `Location` is imposed on every instance of the declared class. Every time an instance is created, the satisfier tries to find appropriate values for its x and y coordinates. This might result in changing the coordinates of one or more of the previously created instances.

The advantage of this approach is that the restrictions on every queen are clearly stated inside its class and are not buried between a large number of lines of code. Additionally, the programmer does not have to worry about how search is done.

A similar approach can be taken in solving many search problems. Another example of a search problem is the cryptoarithmetic problem. In cryptoarithmetic problems, the objective is to find appropriate assignments of digits to letters in order to meet a certain mathematical statement [Newell and Simon, 1972, Simon, 1981]. For example:

$$\begin{array}{cccccc} & S & E & N & D \\ + & M & O & R & E \\ \hline M & O & N & E & Y \end{array}$$

The above cryptoarithmetic problem can be solved by taking each column and convert it into an equation, producing the following set of equations:

$$D + E = Y \ mod \ 10$$

$$(Y \ div \ 10) + N + R = (E \ mod \ 10)$$

$$(E \ div \ 10) + E + O = (N \ mod \ 10)$$

$$(N \ div \ 10) + S + M = (O \ mod \ 10)$$

$$(O \ div \ 10) = M$$

The above equations are then converted into constraints and passed to a constraint-satisfier with the additional constraint that every variable must assume a unique value.

7.3 Support for Direct Mapping of Problem Specifications

Hoare [Hoare, 1987] introduced and presented a selection of formal methods for the process of software specifications. He used, as an example, the process of computing

the greatest common divisor of two positive numbers. He presented the following as the functional specification for the process of finding the greatest common divisor:

$$gcd(x,y) = x \qquad \text{if } x = y$$
$$gcd(x,y) = gcd(x-y,y) \qquad \text{if } x > y$$
$$gcd(x,y) = gcd(y,x) \qquad \text{if } x < y$$

He also said the following regarding the choice of an implementation language:

> It is obviously sensible to use for final coding a language which is as close as possible to that of the original specification, so that the number of steps in the design process is kept small.

Figure 7.5 shows how the above specifications of finding a greatest common divisor can be mapped directly into Electra code by representing each line as a separate instance of the guarded function `gcd()`. It is obvious that the same behavior can

```
guarded function gcd( x,y :integer )→integer;
[ x = y ];
  begin
     return x;
  end;


guarded function gcd( x,y :integer )→integer;
[ x > y ];
  begin
     return gcd( x − y, y);
  end;


guarded function gcd( x,y :integer )→integer;
[ x < y ];
  begin
     return gcd( y , x);
  end;
```

Figure 7.5. The Mapping of the Specification of GCD into Electra Code.

be achieved by combining the bodies of the instances of the `gcd()` guarded function into a single regular function and use some conditional statements to get to the appropriate actions. However, this approach of defining behavior by a group of independent functions provides the programmer with the ability to incrementally define the specifications and characteristics of that behavior. This way, the evolution of the function is more localized instead of being lost within the branches of a case or if-then-else statements.

7.4 Support for Increasing Software Reusability

Biggerstaff and Perlis [Biggerstaff and Perlis, 1989] defined software reuse in the following quote:

> Software reuse is the reapplication of a variety of kinds of knowledge about one system to another similar system in order to reduce the effort of development and maintenance of that other system. This reused knowledge includes artifacts such as domain knowledge, development experience, design decisions, architectural structures, requirements, designs, code, documentation, and so forth.

This section concentrate on code reuse and shows how the constructs of feature exclusion and constraints help in increasing the degree of software reuse that is normally obtained by using inheritance.

7.4.1 Support of Feature Exclusion for Software Reuse

Through inheritance we can build new software modules on top of an existing hierarchy of modules. Adding feature exclusion to inheritance will enable programmers to start with more comprehensive and general classes, and be able to focus behavior and structure via constraining the basic types. This will increase the level of code sharing (and hence, reusability). Snyder [Snyder, 1991] defines two meanings for the term inheritance. The first refers to a classification hierarchy of classes. This meaning defines a type of inheritance sometimes called *specification inheritance* which

often relates to the type model used in type checking. The other meaning is as a mechanism that allows new classes to be defined as incremental modifications to old ones. This meaning defines a type of inheritance called *implementation inheritance* and is intended to support modularity and code reuse. Following that, he indicated that it is necessary to separate the two notions. Regarding the relation between implementation inheritance and inheritance exceptions, Snyder [Snyder, 1986b] states: "... if inheritance is viewed as an implementation technique, then excluding operations is both reasonable and useful." In the same article, he gave an example of the abstraction of a stack and a deque, where he defined a stack as a queue that permits elements to be added or removed from one end, and a deque as a queue that permits elements to be added or removed from either end. Therefore the external interface of a deque is a superset of the external interface of a stack because a deque has two additional operations for adding and removing elements from the other end.

Regarding the implementation of these two data structures, Snyder proposed the following:

> The simplest way to implement these two abstractions (at least for prototyping purposes) is to define the class stack to inherit from the class deque, and exclude any extra operations. Stack inherits the implementation of deque, but is not a specialization of deque, as it does not provide all the deque operations.

Inheritance provides an approach of implementing Is-a relationships. However, when combined with the construct of feature exclusion then one is able to express a new kind of relationships which we call *Is-a-kind-of* relationships [2]. For instance, a set is a kind of bag with the constraint that each element occurs only once, a three-legged elephant is a kind of elephant, and a circle is a kind of ellipse. Being able to express this kind of relationship (i.e., is-a-kind-of relationship) increases the

---

[2]It is important to note that some researchers such as Wirfs-Brock, Wilkerson and Wiener [Wirfs-Brock *et al.*, 1990] use the term "Is-kind-of" to refer to what we call "Is-a" relationship.

chances of being able to reuse a class, because the set of entities conforming to an is-a relationship is a subset of the set of entities conforming to the new relationship.

7.4.2 Support of Constraints for Software Reuse

Constraints can be a useful tool in increasing the degree of software reuse that is normally provided by inheritance, especially in cases where the subclass adds certain structural or behavioral restrictions over those features appearing in its superclass. Examples of such characteristics are often faced when constructing classes that describe geometrical shapes. Such examples include:

- A square is a rectangle where all sides having equal length.
- A trapezoid is a quadrilateral with two parallel sides.
- A right triangle is a triangle with one 90° angle.
- A circle is an ellipse with its two foci residing at the same point.

Figure 7.6 shows the declaration of a `Rectangle` class and subclassing it to create the `Square` class with the addition of the constraint `EqualSides`. This constraint states that the length and height of a square must always be equal. Every operation that is applicable on a rectangle is also applicable on a square as long as the constraint `EqualSides` is maintained valid.

7.5 Support for Expressing Data Abstractions

The availability of constraints provides support for the task of abstraction in data typing because they enhance the mapping between implementation and specification which makes the former a better reflection of the latter. Thus, they help the programmer express the semantics of an abstract data type as directly as possible, and hence, they help in constructing more complete and correct data types. For instance, when one is creating an employee class with the feature EMPLOYEE-NUMBER, it

```
class Rectangle of Shape;
        length : integer;
        hieght : integer;
        ....
        some methods such as Area()
        ....
    end;

class Square of Rectangle;
        constraint EqualSides: required;
            assert
                length = height;
        end;
   end;
```

Figure 7.6. Constraint and Software Reuse.

is difficult to express the fact that every employee has a unique number if one does not have a construct that can express this kind of constraint.

7.6 Support for Expressing Type Extensions

A number of languages such as Ada [US Department of Defense, 1983, Barnes, 1994], Modula-2 [Wirth, 1985], and Pascal [Jensen and Wirth, 1985] support a mechanism to define a type as a subtype of another type. In particular, they provide the programmer with the ability to define subranges of the integer type. For example, in Pascal it is possible to define a subrange of integer as follows:

```
Type
    Digits : 0 .. 9;
```

Any variable that is declared of type `Digits` can assume any value between 0 and 9 inclusively, and cannot be assigned any value outside that range. Any attempt to violate this restriction would generate a compile-time error (a run-time error in the

case of computable values). With the exception of this restriction, variables of this type can be treated exactly like integer variables. Electra does not support creating subranges of the integer type for reasons of simplicity. However, it is possible to create a class with an intra-instance class constraint that simulate the behavior of the subrange type. Figure 7.7 lists the class `Digits` which provides a generalized approach for simulating the behavior of the Pascal subrange that is declared above. If the instance variable `val` is to be assigned a value out of the range 0..9 then the fix of the constraint `Boundary` is invoked. The fix will send an error message and set the instance variable `val` to NIL. The reason we claim that this is a generalized approach is that the programmer has total control as to how the situation is resolved when an attempt is made to assign a value out of the permitted range. Compare this to the Pascal version where an assignment of an out of range value will generate an error message and terminate the program.

```
class Digits;
      val : integer;

      constraint Boundary;
        assert
            ( val >= 0) & (val <= 9 );
        fix
          begin
             print("Value out of range, set to NIL. \n");
             val := NIL;
          end;
      end;
   end;
```

Figure 7.7. Creating a Subrange Class Via a Constraint.

7.7 Integrity Constraints

Integrity constraints of any system, especially that of a database system, are concerned with the maintenance of the correctness and consistency of the data. Implementation of such integrity rules can be an error-prone and complicated process. Thus, there is a need for constructs that assist in directly expressing relationships among objects. For instance, *domain integrity rules* which maintain the correctness of attribute values in a relational database can be directly mapped to simple constraints. An example of a domain integrity rule is the rule needed to state that the attribute COST must have a positive value. Another example is the rule needed to express the fact that the value stored in the field HOURS-PER-WEEK must not be negative and must not exceed the maximum number of hours possible per week. Another type of integrity rule is known as an *intrarelation integrity rule*, which is used to express the functional dependencies between attributes. For example, in a relation such as FATHER(X,Y), a rule is needed to express the fact that a person can never be a father to himself (i.e., X and Y should be constrained not to have the same value). Therefore, constraint constructs can be a very useful mechanism to implement integrity constraints that are associated with objects or their instance variables.

7.8 Support for Validation and Debugging

There are several types of validation:

- The validation that the design of a system reflects the stated specifications.
- The validation that the written code behaves according to the designer's intention.

The process of software validation is a desirable, yet complicated process. Some languages provide constructs to support the process of code correctness validation such as the "assert()" construct in ANCI C [Plauger, 1992], and Eiffel's "require" and

"ensure" constructs [Meyer, 1993]. Since constraints provide the programmer with more control over what is provided by simple assertions, we feel that they can provide more assistance and be more effective in influencing the second type of validation.

The advantages of having constraints, condition-based dispatching, and feature exclusion are not limited to the above list but rather include expressiveness, ease of use, and reduction of checks that otherwise would have to be performed at various points throughout the program. In particular, the availability of constraints with their declarative style assists the programmer in expressing relations without worrying about their maintenance or satisfaction.

Chapter 8
Summary, Future Work, and Conclusion

This chapter begins by presenting a short summary of this dissertation. After that, it lists some of the anticipated future directions of this research. Finally, the chapter closes by providing some concluding remarks.

8.1 Summary

This dissertation presents the design and implementation aspects of the language Electra. Electra extends the language Leda by integrating the constraint paradigm into it. It also enhances the functional and object-oriented paradigms of Leda by adding condition-based dispatching to the former and feature exclusions to the latter. The dissertation begins by providing a general meaning of the term "paradigm" and its connotation in the context of programming languages. After that, it describes the four paradigms that are provided by Leda and presents Electra's added constructs. The second chapter covers the fundamental concepts of constraints including their definition, advantages, and approaches for satisfying them. It also illustrates how constraints can provide a natural approach for expressing the solutions to a class of problems called constraint-satisfaction problems. The following chapter presents Electra's constraint constructs. The presentation includes the syntax, semantics, and general characteristics. In chapter four a description is provided regarding the technique of condition-based dispatching and how it can be expressed using guarded functions. Additionally, it shows how condition-based dispatching can be used to simulate the behavior of argument pattern-matching which is a technique that is provided by most functional languages. Chapter five discusses feature exclusions

and shows why it is necessary to be able to express this phenomenon. The Electra compiler and the approaches it took in implementing the added constructs is covered in chapter six. Finally, the dissertation lists some advantages of the added constructs and provides some illustrative examples.

8.2 Future Work

The following lists some of the planned and anticipated work and directions for the future of our research:

- A need for a closer look into the implementation of Electra in order to improve the performance of the compiler and the quality of the generated code.
- A further investigation of the syntax and semantics is needed to insure a smoother blending with other constructs and paradigms in Electra.
- In order to increase the generality of Electra's constraint constructs, the design of the compiler-solver interface must be enhanced to be able to accommodate communication with multiple solvers that have varying capabilities. Additionally, the interface should have the ability to select the appropriate satisfier for each constraint network, depending on the domain and characteristics of that constraint network.
- There is a need to study the interactions between constraints and other paradigms and see how the existence of constraints can help in increasing the utility of other paradigms. Therefore it is necessary to design and implement some large applications that utilize the added constructs along with some of the constructs of the existing paradigms. These applications will provide a vehicle that assists in the process of evaluating how constraints can take advantage of the characteristics of the other paradigms and how the other paradigms can benefit from the existence of constraints.

- The incorporation of Electra's added constructs into the Leda programming environment that is being developed by Pandey [Pandey, 1993]. It will be necessary to extend the programming environment by adding some constraint debugging tools such as a hierarchical constraint graph viewer.

- In the dissertation, we stated that constraint constructs are better suited for expressing constraint-satisfaction problems than any other construct in the portfolio of the Leda language. We feel that using constraint constructs in programs that solve such problems should make these programs easier to understand and therefore maintain or enhance.

  A comprehensibility experiment is in the design stage. The objective of this experiment is to provide an empirical study that will aid in determining the influence of the usage of constraint constructs on program comprehension. The crux of the experiment is to select a number of constraint-satisfaction problems and construct two solution for each problem. The first solution should use constraint constructs while the second should utilize other types of constructs. The experiment is to test how easy it is to comprehend each solution. We hope that the results of this experiment will help in validating our hypothesis that the inclusion of the constraint constructs will make it easier for novice or intermediate programmers to comprehend programs written in Electra.

8.3 Conclusions

The creation of a single programming paradigm that is adequately equipped to naturally express all aspects of complex problems seems to be improbable, at least in the near future. This indicates that multiparadigm languages will be given great attention by language researchers.

The constraint paradigm gives the programmer the ability to declare system-maintained relationships. The semantics of constraints provide a natural and convenient way for expressing a class of problems called constraint-satisfaction problems.

A problem in this class can be solved by decomposing it into a set of constraints, then feeding these constraints to a constraint-satisfier. Solutions to problems in this class cannot be easily expressed using the paradigms that are offered by Leda. The relation between the representation of a problem and its solvability is depicted in the following quote by Freeman-Benson [Freeman-Benson, 1991]:

> It has been argued that *solving a problem is simply a matter of representing it so that the solution is transparent* [Polya, 1945, Simon, 1981]. Although there are many problems for which the solution will never be "transparent," the way a problem is represented has a major influence on our understanding and ability to solve it. Or, from the other side of the problem, the language in which one writes shapes the way that one views the world.

Therefore, we feel that integrating the constraint paradigm into the language Leda will increase its generality since it will be able to naturally express a wider range of problems. Additionally, constraints are suitable for the direct mapping of the characteristics of a number of mechanisms such as: consistency checks, constraint-directed search, and constraint-enforced reevaluation, among others.

The idea of providing the programmer with the ability to state system-maintained relationships is an appealing one. This motivated us to extend other paradigms with similar capabilities. The functional programming paradigm, as presented in Leda, is extended with the ability to express condition-based dispatching. This gives Leda programmers the ability to simulate the mechanism of argument pattern-matching which is provided by most functional programming languages. The advantages of the mechanism of condition-based dispatching include the following:

- It offers an alternative to using some complicated or deeply nested conditional statements.
- It serves as a design aid in helping the programmer to consider all possible inputs to a function.
- It provides a mechanism for direct mapping of specifications to actual code.

- It can be viewed as an approach to make code more readable and thus easier to maintain and enhance.

The object-oriented paradigm is extended by giving the programmer the ability to exclude features of a superclass from appearing in a subclass. This technique represents a form of inheritance exceptions. The main usage of feature exclusion is that it provides the programmer with the ability to express inheritance exceptions. This is helpful because it is not always possible to impose a rigid hierarchical structure on every real world situation. Feature exclusion is also useful in increasing the degree of software reuse that can normally be obtained via the use of inheritance.

The expressiveness and power of the paradigm of constraints as presented in this dissertation is difficult to simulate by any other paradigm. Thus, we are convinced that integrating these concepts into a multiparadigm language should provide the programmer with a new and powerful tool to express his or her objectives.

BIBLIOGRAPHY

[Abelson *et al.*, 1985] Harold Abelson, Gerald Jay Sussman, and Julie Sussman. *Structure and Interpretation of Computer Programs.* MIT Press, Cambridge, Massachusetts, 1985.

[Addanki, 1987] S. Addanki. Connectionism. In Stuart C. Shapiro, editor, *Encyclopedia of Artificial Intelligence*, pages 200–205. John Wiley & Sons, New York, 1987.

[Ambler *et al.*, 1992] Allen L. Ambler, Margaret M. Burnett, and Betsy A. Zimmerman. Operational Versus Definitional: A Perspective on Programming Paradigms. *IEEE Computer*, 25(9):28–43, September 1992.

[Andersen, 1964] Christian Andersen. *An Introduction to Algol 60.* Addison-Wesley Publishing Company, Reading, Massachusetts, 1964.

[Appleby, 1991] Doris Appleby. *Programming Languages Paradigms and Practice.* McGraw-Hill, Inc., New York, 1991.

[Bailey, 1990] R. Bailey. *Functional Programming With HOPE.* Ellis Horwood Limited, Chichester, England, 1990.

[Bal and Grune, 1994] Henri E. Bal and Dick Grune. *Programming Language Essentials.* Addison-Wesley Publishing Company Inc., Wokingham, England, 1994.

[Barnes, 1994] J. G. P. Barnes. *Programming in Ada, Plus an Overview of Ada 9X, 4th Edition.* Addison-Wesley Publishing Company, Reading, Massachusetts, 1994.

[Biggerstaff and Perlis, 1989] Ted J. Biggerstaff and Alan J. Perlis, editors. *Software Reusability.* Addison Wesley, Reading, Massachusetts, 1989.

[Birtwistle *et al.*, 1975] G. M. Birtwistle, O.-J. Dahl, B. Myhrhaug, and K. Nygaard. *Simula Begin.* Petrocelli Charter, New York, 1975.

[Bitner and Reingold, 1975] James R. Bitner and Edward M. Reingold. Backtrack Programming Techniques. *Communications of the ACM*, 18(11):651–656, November 1975.

[Bobrow and Winograd, 1977] D. G. Bobrow and T. Winograd. An Overview of KRL, a Knowledge Representation Language. *Cognitive Science*, 1(1), 1977.

[Boizumault *et al.*, 1993] Patrice Boizumault, Yan Delon, and Laurent Péridy. Solving a Real-Life Planning Exams Problem Using Constraint Logic Programming. In Manfred Meyer, editor, *Constraint Processing: Proceedings of the International Workshop at CSAM'93, St. Petersburg, July 1993*, Research Report RR-93-39, pages 107–112, DFKI Kaiserslautern, Germany, August 1993.

[Borning, 1979] Alan Borning. *ThingLab–A Constraint-Oriented Simulation Laboratory.* PhD thesis, Department of Computer Science, Stanford University, March 1979. A revised version is published as Xerox Palo Alto Research Center Technical Report SSL-79-3 (July 1979).

[Borning, 1981] Alan Borning. The Programming Language Aspects of ThingLab, A Constraint-Oriented Simulation Laboratory. *ACM Transactions on Programming Languages and Systems*, 3(4):353–387, October 1981.

[Borning *et al.*, 1987] Alan Borning, Robert Duisberg, Bjorn Freeman-Benson, Axel Kramer, and Michael Woolf. Constraint Hierarchies. In *OOPSLA'87, Proceedings of the 1987 Conference on Object-Oriented Progamming Systems, Languages, and Applications*, pages 48–60, Orlando, Florida, October 1987.

[Borning *et al.*, 1988] Alan Borning, Michael Maher, Amy Martindale, and Molly Wilson. Constraint Hierarchies and Logic Programming. Technical Report 88-11-10, University of Washington, Seattle, November 1988.

[Borning *et al.*, 1989] Alan Borning, Michael Maher, Amy Martindale, and Molly Wilson. Constraint Hierarchies and Logic Programming. In Giorgio Levi and Maurizio Martelli, editors, *ICLP'89: Proceedings of the 6th International Conference on Logic Programming*, pages 149–164, Lisbon, Portugal, June 1989. MIT Press.

[Brachman, 1977] Ronald J. Brachman. What's in a Concept: Structural Foundations for Semantic Networks. *International Journal of Man-Machine Studies*, 9:127–152, 1977.

[Brachman, 1979] Ronald J. Brachman. On The Epistemological Status of Semantic Networks. In N. Findler, editor, *Associative Networks: The Representation and Use of Knowledge By Computers.* Academic Press, 1979.

[Brachman, 1983] Ronald J. Brachman. What IS-A Is and Isn't: An Analysis of Taxonomic Links in Semantic Networks. *IEEE Cimputer*, 16(10):30–37, 1983.

[Brachman, 1985] Ronald J. Brachman. "I Lied about the Trees" or, Defaults and Definitions in Knowledge Representation. *The AI Magazine*, 6(3):80–93, 1985.

[Budd, 1989] Timothy A. Budd. Data Structures in LEDA. Technical Report 89–60–17, Department of Computer Science, Oregon State University, Corvallis, Oregon, 1989.

[Budd, 1991a] Timothy A. Budd. Blending Imperative and Relational Programming. *IEEE Software*, 8(1):58–65, 1991.

[Budd, 1991b] Timothy A. Budd. *An Introduction to Object Oriented Programming.* Addison-Wesley Publishing Company Inc., Reading, Massachusetts, 1991.

[Budd, 1992] Timothy A. Budd. Multiparadigm Data Structures in Leda. In *Proceedings of the 1992 International Conference on Computer Languages*, pages 165–173, Oakland, CA, April 1992.

[Budd, 1995] Timothy A. Budd. *Multiparadigm Programming In Leda.* Addison-Wesley Publishing Company Inc., Reading, Massachusetts, 1995.

[Cardelli and Wegner, 1985] Luca Cardelli and Peter Wegner. On Understanding Types, Data Abstractions, and Polymorphism. *Computing Surveys*, 17(4):471–522, 1985.

[Chamard *et al.*, 1992] André Chamard, Frédéric Decès, and Annie Fischler. Applying CHIP to a Complex Scheduling Problem. In Krzysztof Apt, editor, *JICSLP'92: Proceedings Joint International Conference and Symposium on Logic Programming*, Washington, DC, November 1992. MIT Press.

[Charman, 1993] Philippe Charman. Solving Space Planning using Constraint Technology. In Manfred Meyer, editor, *Constraint Processing: Proceedings of the International Workshop at CSAM'93, St. Petersburg, July 1993*, Research Report RR-93-39, pages 159–172, DFKI Kaiserslautern, Germany, August 1993.

[Clocksin and Mellish, 1987] W. F. Clocksin and C. S. Mellish. *Programming in Prolog.* Springer-Verlag, Berlin, third, revised and extended edition, 1987.

[Colmerauer, 1990] Alain Colmerauer. An Introduction to Prolog III. *Communications of the ACM*, 33(7):69–90, July 1990.

[Cox, 1986] Brad J. Cox. *Object Oriented Programming: An Evolutionary Approach.* Addison-Wesley Publishing Company Inc., Reading, Massachusetts, 1986.

[Cox and Hunt, 1986] Brad J. Cox and B. Hunt. Objects, Icons, and Software-IC's. *Byte Magazine*, 11(8):161–176, 1986.

[Cull and Pandey, 1994] Paul Cull and Rajeev K. Pandey. Isomorphism and the N-Queens Problem. *SIGCSE Bulletin*, 26(3):29–36, 44, September 1994. Also published as Technical Report 94–20–02, Department of Computer Science, Oregon State University, Corvallis OR.

[Davie, 1992] Antony J. T. Davie. *An Introduction to Functional Programming Systems Using Haskell.* Cambridge University Press, Cambridge, England, 1992.

[Davis and Rosenfeld, 1981] A. L. Davis and A. Rosenfeld. Cooperating Processes for Low-Level Vision: A Survey. *Artificial Intelligence*, 17:245–263, 1981.

[de Kleer and Sussman, 1980] J. de Kleer and G. J. Sussman. Propagation of Constraints Applied to Circuit Synthesis. *Circuit Theory and Applications*, 8:127–144, 1980.

[Dechter and Pearl, 1987] R. Dechter and J. Pearl. Network-based Heuristics for Constraint-satisfaction Problems. *Artificial Intelligence*, 34:1–38, 1987.

[Dincbas *et al.*, 1988] M. Dincbas, H. Simonis, and P. Van Hentenryck. Solving Large Scheduling Problems in Logic Programming. In *EURO-TIMS Joint International Conference on Operations Research and Management Science*, Paris, July 1988.

[Duisberg, 1986] R. Duisberg. *Constraint-Based Animation: The Implementation of Temporal Constraints in the Animus System.* PhD thesis, Department of Computer Science and Engineering, University of Washington, Seattle, Washington, 1986. Also published as University of Washington Computer Science Technical Report 86-09-01 (September 1986).

[Eastman, 1972] C. Eastman. Preliminary Report on a System for General Space Planning. *Communications of the ACM*, 15:76–87, 1972.

[Ellis, 1982] T. M. R. Ellis. *A Structured Approach to Fortran 77 Programming.* Addison-Wesley Publishing Company Inc., Wokingham, England, 1982.

[Etherington, 1987] David Etherington. More on Inheritance Hierarchies With Exceptions Default Theories and Inferential Distance. In *AAAI-87: Proceedings of the 7th National Conference on Artificial Intelligence*, pages 352–357, Seattle, Wash., August 1987. American Association for Artificial Intelligence.

[Etherington and Reiter, 1983] D. Etherington and R. Reiter. An Inheritance Hierarchies With Exceptions. In *AAAI-83: Proceedings of the National Conference on Artificial Intelligence*, Washington, DC, August 1983. American Association for Artificial Intelligence.

[Field and Harrison, 1988] Anthony J. Field and Peter G. Harrison. *Functional Programming.* Addison-Wesley Publishing Company, Reading, Massachusetts, 1988.

[Floyd, 1979] Robert W. Floyd. The Paradigms of Programming. *Communications of the ACM*, 22(8):455–460, August 1979.

[Fox, 1979] Mark S. Fox. On Inheritance in Knowledge Representation. In *IJCAI'79: Proceedings of the International Joint Conference on Artificial Intelligence*, volume 1, Tokyo, Japan, August 1979.

[Fox, 1983] Mark S. Fox. *Constraint-Directed Search: A Case Study of Job-Shop Schedualing.* PhD thesis, Carnegie-Mellon University, 1983.

[Frayman and Mittal, 1987] F. Frayman and S. Mittal. COSSACK: A Constraint-Based Expert System for Configuration Tasks. In D. Sriram and R. A. Adey, editors, *Knowledge-Based Expert Systems in Engineering: Planning and Design.* Computational Mechanics, Billerica, Massachusetts, 1987.

[Freeman-Benson, 1990] Bjorn N. Freeman-Benson. Kaleidoscope: Mixing Objects, Constraints, and Imperative Programming. In Norman Meyrowitz, editor, *OOPSLA/ECOOP'90*, pages 77–88, Ottawa, Canada, October 1990. ACM Press.

[Freeman-Benson, 1991] Bjorn Freeman-Benson. *Constraint Imperative Programming.* PhD thesis, University of Washington, Department of Computer Science

and Engineering, Seattle, Washington, July 1991. Published as Department of Computer Science and Engineering Technical Report 91-07-02.

[Freeman-Benson and Borning, 1991] Bjorn N. Freeman-Benson and Alan Borning. The Design and Implementation of Kaleidoscope'90: A Constraint Imperative Programming Language. In *Proceedings IEEE Computer Society International Conference on Computer Languages*, pages 174–180, San Fransisco, April 1991.

[Freeman-Benson and Borning, 1992] Bjorn N. Freeman-Benson and Alan Borning. Integrating Constraints with an Object-Oriented Language. In *ECOOP'92: Proceedings of the European Conference on Object-Oriented Programming*, pages 268–286, Utrecht, Netherlands, 1992.

[Freeman-Benson *et al.*, 1990a] Bjorn Freeman-Benson, John Maloney, and Alan Borning. An Incremental Constraint Solver. *Communications of the ACM*, 33(1):54–63, January 1990.

[Freeman-Benson *et al.*, 1990b] Bjorn Freeman-Benson, John Maloney, and Alan Borning. The DeltaBlue Algorithm: An Incremental Constraint Hierarchy Solver. Technical Report 89-08-06, Department of Computer Science and Engineering, University of Washington, February 1990.

[Freeman-Benson and Wilson, 1990] Bjorn Freeman-Benson and Molly Wilson. DeltaStar: A General Algorithm for Incremental Satisfaction of Constraint Hierarchies. Technical Report 90-05-02, University of Washington, Seattle, May 1990.

[Freeman-Benson *et al.*, 1992] Bjorn N. Freeman-Benson, Molly Wilson, and Alan Borning. DeltaStar: A General Algorithm for Incremental Satisfaction of Constraint Hierarchies. In *Proceedings 11th IEEE Phoenix Conference on Computers and Communications*, Scottsdale, Arizona, March 1992.

[Freuder, 1978] E. C. Freuder. Synthesizing Constraint Expressions. *Communications of the ACM*, 21:958–966, 1978.

[Freuder, 1982] E. C. Freuder. A Sufficient Condition of Backtrack-free Search. *Journal of the ACM*, 29(1):24–32, 1982.

[Gaschnig, 1977] J. A. Gaschnig. A General Backtrack Algorithm that Eliminates Most Redundant Tests. In *IJCAI'77: Proceedings of the Fifth International Joint Conference on Artificial Intelligence*, Cambridge, Massachusetts, August 1977.

[Ghezzi and Jazayeri, 1987] Carlo Ghezzi and Mehdi Jazayeri. *Programming Language Concepts.* Wiley & Sons, New York, 1987. Second Edition.

[Goldberg and Robson, 1983] Adele Goldberg and David Robson. *Smalltalk-80: The Language and Its Implementation.* Addison-Wesley Publishing Company, Reading, Massachusetts, 1983.

[Gore, 1990] Jacob Vadim Gore. *Constraint-Driven Programming in a Strongly-Typed Object-Oriented Language.* PhD thesis, Department of Computer Science, Northwestern University, Evanston, IL., 1990.

[Hailpern, 1986] Brent Hailpern. Multiparadigm Languages and Environments. *IEEE Software*, 3(1):6–9, January 1986.

[Haralick and Elliot, 1980] R. M. Haralick and G. L. Elliot. Increasing Tree Search Efficiency for Constraint Satisfaction Problems. *Artificial Intelligence*, 14:263–313, 1980.

[Harel, 1992] David Harel. *Algorithmics: The Spirit of Computing.* Addison-Wesley Publishing Company, Reading, Massachusetts, 1992. Second Edition.

[Heintze *et al.*, 1992] Nevin C. Heintze, Joxan Jaffar, Spiro Michaylov, Peter J. Stuckey, and Roland H. C. Yap. *The CLP(R) Programmer's Manual.* IBM T. J. Watson Research Center, PO Box 704, Yorktown Heights, NY 10598, September 1992.

[Henderson, 1980] Peter Henderson. *Functional Programming: Application and Implementation.* Prentice Hall International, Englewoods Cliffs, New Jersey, 1980.

[Hill and Lloyd, 1994] Patricia Hill and John Lloyd. *The Gödel Programming Language.* MIT Press, Cambridge, Massachusetts, 1994.

[Hoare, 1987] C. A. R. Hoare. An Overview of Some Formal Methods of Program Design. *IEEE Computer*, 20(10):30–37, 1987.

[Horn, 1992a] Bruce Horn. Constraint Patterns as Bases for Object-Oriented Constraint Programming. In *OOPSLA'92: Proceedings of The 1992 ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, Vancover, British Colombia, October 1992.

[Horn, 1992b] Bruce Horn. Properties of User Interface Systems and the Siri Programming Language. In Brad Myers, editor, *Languages for Developing User Interfaces*, pages 211–236. Jones and Bartlett, Boston, 1992.

[Hudak and Wadler, 1988] P. Hudak and P. Wadler. Report on the Functional Programming Language Haskell. Technical Report YALEU/DCS/RR-627, Department of Computer Science, Yale University, New Haven, CT, November 1988.

[Hwang, 1993] Kai Hwang. *Advanced Computer Architecture: Parallelism, Scalability, Programmability.* McGraw-Hill, Inc., New York, 1993.

[Iverson, 1962] K. E. Iverson. *A Programming Language.* Wiley & Sons, New York, 1962.

[Jaffar and Lassez, 1987] Joxan Jaffar and Jean-Louis Lassez. Constraint logic programming. In *POPL'87: Proceedings 14th ACM Symposium on Principles of Programming Languages*, pages 111–119, Munich, 1987. ACM.

[Jaffar *et al.*, 1992] Joxan Jaffar, Spiro Michaylov, Peter J. Stuckey, and Roland H. C. Yap. An Abstract Machine for CLP($\mathcal{R}$). In *PLDI'92: Proceedings ACM SIGPLAN Symposium on Programming Language Design and Implementation*, pages 128–139, San Francisco, CA, June 1992.

[Jensen and Wirth, 1985] Kathleen Jensen and Niklaus Wirth. *Pascal User Manual and Report.* Springer-Verlag, New York, third edition, 1985.

[Johnson and Rees, 1992] Paul Johnson and Ceri Rees. Reusability Through Fine-grain Inheritance. *Software–Practice and Experience*, 22(12):1049–1068, December 1992.

[Justice, 1995] Timothy P. Justice. Applicability of multiparadigm programming to compiler construction tools. Ph.D. Proposal, Department of Computer Science, Oregon State University, 1995. To be presented to the Ph.D. committee early 1995.

[Justice *et al.*, 1993] Timothy P. Justice, Rajeev K. Pandey, and Timothy A. Budd. Compiler Implementation in the Multiparadigm Language Leda. Technical Report 93-60-20, Department of Computer Science, Oregon State University, Corvallis, Oregon, December 1993.

[Justice *et al.*, 1994] Timothy P. Justice, Rajeev K. Pandey, and Timothy A. Budd.

A Multiparadigm Approach to Compiler Construction. *SIGPLAN Notices*, 29(9):29–37, September 1994.

[Kernighan and Ritchie, 1978] B. W. Kernighan and D. M. Ritchie. *The C Programming Language.* Prentice Hall, Englewoods Cliffs, New Jersey, 1978.

[Kernighan and Ritchie, 1988] B. W. Kernighan and D. M. Ritchie. *The C Programming Language, Second Edition.* Prentice Hall, Englewoods Cliffs, New Jersey, 1988.

[Kim and Lochovsky, 1989] Won Kim and Frederick H. Lochovsky, editors. *Object-Oriented Concepts, Databases, and Applications.* Addison-Wesley Publishing Company, Reading, Massachusetts, 1989.

[Kowalski, 1974] R. Kowalski. Predicate Logic as a Programming Language. In *Proceedings of the IFIP 74*, pages 569–574, 1974.

[Kowalski, 1979] R. Kowalski. Algorithn = Logic + Control. *Communication of the ACM*, 22(7):424–431, July 1979.

[Kuhn, 1970] Thomas S. Kuhn. *The structure of Scientific Revolutions.* The University of Chicago Press, Chicago, IL, 1970. Second Edition.

[Kumar, 1992] Vipin Kumar. Algorithms for Constraint-Satisfaction Problems: A Survey. *A.I. Magazine*, 13(1):32–44, Spring 1992.

[Lauriere, 1978] Jean-Louis Lauriere. A Language and A Program for Stating and Solving Combinatorial Problems. *Artificial Intelligence*, 10(1):29–127, February 1978.

[Ledbetter and Cox, 1985] L. Ledbetter and Brad J. Cox. Software-IC's. *Byte Magazine*, 10(6):307–316, 1985.

[Leler, 1986] Wim Leler. *Specification and Generation of Constraint Satisfaction Systems Using Augmented Term Rewriting.* PhD thesis, Department of Computer Science, University of North Carolina at Chapel Hill, Chapel Hill, North Carolina, 1986.

[Leler, 1988] Wim Leler. *Constraint Programming Languages: Their Specification and Generation.* Addison-Wesley, Reading, Massachusetts, 1988.

[Lenzerini *et al.*, 1991] Maurizio Lenzerini, Daniele Nardi, and Maria Simi, editors. *Inheritance Hierarchies in Knowledge Representation and Programming Languages.* John Wiley & Sons, Chichester, West Sussex, England, 1991.

[Lopez *et al.*, 1994a] Gus Lopez, Bjorn Freeman-Benson, and Alan Borning. Implementing Constraint Imperative Programming Languages: The Kaleidoscope'93. In *OOPSLA'94: Proceedings of the 1994 Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 259–271, Portland, Oregon, October 1994. ACM.

[Lopez *et al.*, 1994b] Guy Lopez, Bjorn Freeman-Benson, and Alan Borning. Kaleidoscope: A Constraint Imperative Programming Language. In B. Mayoh, E. Tyugu, and J.Penjaam, editors, *Constraint Programming: Proceedings 1993 NATO ASI Parnu, Estonia*, NATO Advanced Science Institute Series, pages 305–321. Springer-Verlag, 1994.

[Mackworth, 1977a] Alan K. Mackworth. Consistency in Networks of Relations. *Artificial Intelligence*, 8(1):99–118, 1977.

[Mackworth, 1977b] Alan K. Mackworth. On Reading Sketch Maps. In *Proceedings 5th International Joint Conference on Artificial Intelligence*, pages 598–606, August 1977.

[Mackworth, 1992] Alan K. Mackworth. Constraint Satisfaction. In Stuart C. Shapiro, editor, *Encyclopedia of Artificial Intelligence*, pages 285–293, New York, 1992. John Wiley & Sons.

[MacLennan, 1987] Bruce J. MacLennan. *Principles of Programming Languages.* Holt, Rinehart and Winston, New York, 1987.

[MacLennan, 1990] Bruce J. MacLennan. *Functional Programming Practice and Theory.* Addison-Wesley Publishing Company, Reading, Massachusetts, 1990.

[Madsen *et al.*, 1993] Ole Lehermann Madsen, Birger Møller-Pederson, and Kristen Nygaard. *Object-Oriented Programming in the Beta Programming Language.* Addison-Wesley Publishing Company Inc., Wokingham, England, 1993.

[Maloney, 1991] John Harold Maloney. *Using Constraints for User Interface Construction.* PhD thesis, Department of Computer Science and Engineering, University of Washington, Seattle, Washington, 1991. Also published as Technical Report 91-08-12.

[Maloney *et al.*, 1989] John Maloney, Alan Borning, and Bjorn Freeman-Benson. Constraint Technology for User-Interface Construction in ThingLab II. In *Proceedings of the 1989 ACM Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 381–388, New Orleans, Louisiana, October 1989. Also published as University of Washington Computer Science Technical Report 89-05-02 (May 1989).

[McGregor, 1979] J. McGregor. Relational Consistency Algorithms and their Application in Finding Subgraph and Graph Isomorphisms. *Information Sciences*, 19:229–250, 1979.

[Meyer, 1987] Bertrand Meyer. Reusability: The Case for Object-Oriented Design. *IEEE Software*, 4(2):50–64, March 1987.

[Meyer, 1988] Bertrand Meyer. *Object-Oriented Software Construction.* Prentice Hall International, Englewoods Cliffs, New Jersey, 1988.

[Meyer, 1993] Bertrand Meyer. *Eiffel the Language.* Prentice Hall International, Hertforshire, England, 1993.

[Milner *et al.*, 1990] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML.* MIT Press, Cambridge, Massachusetts, 1990.

[Montanari, 1974] Ugo Montanari. Networks of Constraints: Fundamental Properties and applications to Picture Processing. *Information Sciences*, 7(2):95–132, April 1974.

[Myers, 1992] Brad A. Myers. State of the Art in User Interface Software Tools. Technical Report CMU-CS-92-114, School of Computer Science, Carnegie Mellon University, February 1992.

[Nemhasuer *et al.*, 1989] G. L. Nemhasuer, A. H. G. Rinnooy Kan, and M. J. Todd. *Optimization.* Elsevier Science Publishing Co., Amsterdam, Holland, 1989.

[Newell and Simon, 1972] Allen Newell and Herbert Alexander Simon. *Human Problem Solving.* Prentice Hall, Englewoods Cliffs, New Jersey, 1972.

[O'Donnel, 1985] M.J. O'Donnel. *Equational Logic as a Programming Language.* MIT Press, Cambridge, Massachusetts, 1985.

[Pandey, 1993] Rajeev K. Pandey. Sparta: A Programming Environment for the multiparadigm Language Leda. Ph.D. Proposal, Department of Computer Science, Oregon State University, May 1993.

[Pandey *et al.*, 1993] Rajeev Pandey, Wolfgang Pesch, Jim Shur, and Masami Takikawa. A Revised Leda Language Definition. Technical Report 93-60-02, Department of Computer Science, Oregon State University, Corvallis, Oregon, January 1993.

[Pesch and Shur, 1991] Wolfgang Pesch and Jim Shur. A Leda Language Definition. Technical Report 91-60-09, Department of Computer Science, Oregon State University, Corvallis, Oregon, September 1991.

[Peyton Jones, 1987] Simon L. Peyton Jones. *The Implementation of Functional Programming Languages.* Prentice-Hall International, Englewoods Cliffs, New Jersey, 1987.

[Placer, 1988] John R. Placer. *G: A Language Based On Demand-Driven Stream Evaluation.* PhD thesis, Department of Computer Science, Oregon State University, Corvallis, Oregon, November 1988.

[Placer, 1991a] John Placer. Multiparadigm Research: A New Direction in Language Design. *ACM SIGPLAN Notices*, 26(3):9–17, March 1991.

[Placer, 1991b] John Placer. The Multiparadigm Language G. *Computer Language*, 16(3/4):235–258, 1991.

[Plauger, 1992] P. J. Plauger. *The Standard C Library.* Prentice Hall, Englewoods Cliffs, New Jersey, 1992.

[Polya, 1945] G Polya. *How To Solve It.* Princeton University Press, Princeton, New Jersey, 1945. Second Edition.

[Prosser, 1988] Patrick Prosser. Reactive Factory Scheduling as a Dynamic Constraint Satisfaction Problem. Technical Report AISL-31-88, University of Strathclyde, August 1988.

[Prosser *et al.*, 1992] Patrick Prosser, Chris Conway, and Claude Muller. A Constraint Maintenance System for the Distributed Resource Allocation Problem. *Intelligent Systems Engineering*, pages 76–83, Autumn 1992.

[Reade, 1989] Chris Reade. *Elements of Functional Programming.* Addison-Wesley Publishing Company Inc., Wokingham, England, 1989.

[Roberts and Goldstein, 1977] R. B. Roberts and I. Goldstein. The FRL Primer. Technical Report AIM-408, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, Massachusetts, April 1977.

[Sannella *et al.*, 1993] Michael Sannella, John Maloney, Bjorn Freeman-Benson, and Alan Borning. Multi-Way Verses One-Way Constraints in User Interfaces: Experience With the DeltaBlue Algorithm. Technical Report 92-07-05a, Department of Computer Science and Engineering, University of Washington, February 1993. It is a slightly revised version of Technical Report 92-07-05, July 1992.

[Saraswat, 1993] Vijay A. Saraswat. *Concurrent Constraint Programming.* MIT Press, 1993.

[Saraswat *et al.*, 1990] Vijay A. Saraswat, Kenneth M. Kahn, and Jacob Levy. Janus: A Step Towards Distributed Constraint Programming. In *Proceedings North American Conference on Logic Programming*, Austin, Texas, October 1990.

[Satoh and Aiba, 1991] Ken Satoh and Akira Aiba. Computing Soft Constraints by Hierarchical Constraint Logic Programming. Technical Report TR-610, Institute for New Generation Computer Technology, Tokyo, January 1991.

[Schaffert *et al.*, 1986] C. Schaffert, T. Cooper, B. Bullis, M. Killian, and C. Wilpot. An Introduction to Trellis/Owl. In *OOPSLA '86: Proceedings of the 1986 Conference on Object-Oriented Progamming Systems, Languages, and Applications*, Portland, Oregon, October 1986. Published as ACM SIGPLAN Notices, 21(11), November, 1986.

[Shriver and Wegner, 1988] Bruce Shriver and Peter Wegner, editors. *Research Directions in Object-Oriented Programming.* MIT Press, Cambridge, Massachusetts, 1988.

[Simon, 1981] Herbert Alexander Simon. *The Sciences of the Artificial.* MIT Press, Cambridge, Massachusetts, 1981.

[Snyder, 1986a] Alan Snyder. CommonObjects An Overview. In *Proceedings of the Object Oriented Programming Workshop*, October 1986. Published as ACM SIGPLAN Notices 21(10), October, 1986.

[Snyder, 1986b] Alan Snyder. Encapsulation and Inheritance in Object-Oriented Programming Languages. In *OOPSLA '86: Proceedings of the 1986 Conference on Object-Oriented Progamming Systems, Languages, and Applications*, Portland, Oregon, October 1986. Published as ACM SIGPLAN Notices 21(11), November, 1986.

[Snyder, 1991] Alan Snyder. Inheritance in Object Oriented Programming Languages. In Maurizio Lenzerini, Daniele Nardi, and Maria Simi, editors, *Inheritance Hierarchies in Knowledge Representation and Programming Languages*, chapter 10, pages 153–171. John Wiley & Sons, Chichester, West Sussex, England, 1991.

[Stafik *et al.*, 1986] Mark J. Stafik, Daniel G. Bobrow, and Kenneth M. Kahn. Integrating Access-Oriented Programming into a Multi-paradigm Environment. *IEEE Software*, January 1986.

[Steele, 1980] Guy Lewis Steele. *The Definition and Implementation of a Computer Programming Language Based on Constraints.* PhD thesis, Massachusetts Institute of Technology, August 1980. Also published as MIT Artificial Intelligence Laboratory Technical Report 595 and as MIT VLSI Memo 80-32.

[Steele Jr., 1990] Guy Lewis Steele Jr. *Common LISP The Language, Second Edition.* Digital Press, Bedford, Massachusetts, 1990.

[Steele Jr. and Sussman, 1975] Guy Lewis Steele Jr. and Gerald G. Sussman. Scheme: An Interpreter for the Extended Lambda Calculus. Technical Report Memo 349, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, Massachusetts, April 1975.

[Sussman and Steele, 1980] G. Sussman and Guy Lewis Steele. CONSTRAINTS–A Language for Expressing Almost-Hierarchical Descriptions. *Artificial Intelligence*, 14(1):1–39, January 1980.

[Sutherland, 1963a] Ivan Sutherland. Sketchpad: A Man-Machine Graphical Communication System. In *Proceedings of the Spring Joint Computer Conference*, pages 329–345. IFIPS, 1963.

[Sutherland, 1963b] Ivan E. Sutherland. *SKETCHPAD: A Man-Machine Graphical Communication System.* PhD thesis, Department of Electrical Engineering, Massachusetts Institute of Technology, January 1963.

[Szypersky *et al.*, 1993] Clemens Szypersky, Stephen Omohundro, and Stephan Murer. Engineering a Programming Language: The Type and Class System of Sather. Technical Report TR-93-064, International Computer Science Institute, University of California, Berkeley, November 1993.

[Touretzky, 1986] D. Touretzky. *The Mathematics of Inheritance System.* Morgan Kaufmann, Los Altos, Ca., 1986.

[Tsang, 1993] E. Tsang. *Foundations of Constraint Satisfaction.* Academic Press Limited, London, England, 1993.

[Turner, 1985] David. A. Turner. Miranda: A Non-Strict Functional Language With Polymorphic Types. In *Proceedings of the IFIP International Conference on Functional Programming Languages and Computer Architecture*, pages 1–16, Nancy, France, September 1985. Springer Verlag. Published as Springer Lecture Notes in Computer Science, vol. 201.

[Turner, 1986] David. A. Turner. An Overview of Miranda. *ACM SIGPLAN Notices*, 21(12):158–166, December 1986.

[Ullman, 1976] J. R. Ullman. An Algorithm for Subgraph Isomorphism. *J. ACM*, 23:31–42, 1976.

[Ullman, 1994] Jeffrey D. Ullman. *ML Programming.* Prentice Hall, Englewoods Cliffs, New Jersey, 1994.

[US Department of Defense, 1983] US Department of Defense. Ada Programming Language. Technical Report ANSI/MIL-STD-1815A, American National Standards Institute, Washington DC, January 1983.

[Van Hentenryck, 1991] Pascal Van Hentenryck. Constraint Logic Programming. *Knowledge Engineering Review*, 6(3):151–194, September 1991.

[Van Hentenryck, 1993] Pascal Van Hentenryck. Scheduling and Packing in the Constraint Language cc(FD). Technical Report CS-93-02, Department of Computer Science, Brown University, January 1993.

[Walinsky, 1989] Clifford Walinsky. CLP(Σ^*): Constraint Logic Programming with Regular Sets. In Giorgio Levi and Maurizio Martelli, editors, *ICLP'89: Proceedings 6th International Conference on Logic Programming*, pages 181–196, Lisbon, Portugal, June 1989. MIT Press.

[Wallace, 1994] Mark Wallace. Applying Constraints for Scheduling. In B. Mayoh, E. Tyugu, and J.Penjaam, editors, *Constraint Programming: Proceedings 1993 NATO ASI Parnu, Estonia*, NATO Advanced Science Institute Series, pages 161–180. Springer-Verlag, 1994.

[Waltz, 1975] D. Waltz. Understanding Line Drawings of Scenes With Shadows. In P. H. Winston, editor, *The Psychology of Computer Vision*, pages 19–91. McGraw-Hill, Cambridge, Mass., 1975.

[Wegner and Zdonik, 1988] P. Wegner and S. Zdonik. Inheritance as an Incremental Modification Mechanism or What Like Is and Isn't Like. In *ECOOP '88: Proceedings of the European Conference on Object Oriented Programming*, Oslo, Norway, August 1988. Springer Verlag.

[Welland, 1983] R. C. Welland. *Methodical Programming in COBOL.* Pitman, London, England, 1983.

[Wikström, 1987] Åke Wikström. *Functional Programming Using Standard ML.* Prentice-Hall International, Englewoods Cliffs, New Jersey, 1987.

[Wilson and Borning, 1993] Molly Wilson and Alan Borning. Hierarchical Constraint Logic Programming. *Journal of Logic Programming*, 16(3):277–318, July 1993. (Also published as Technical Report 93-01-02 from the University of Washington, Seattle).

[Wilson and Clark, 1988] Lesily B. Wilson and Robert G. Clark, editors. *Comparative Programming Languages.* Addison Wesley, Reading, Massachusetts, 1988.

[Wirfs-Brock *et al.*, 1990] Rebecca Wirfs-Brock, Brian Wilkerson, and Lauren Wiener. *Designing Object-Oriented Software.* Prentice Hall, Englewoods Cliffs, New Jersey, 1990.

[Wirth, 1985] Niklaus Wirth. *Programming in Modula-2, 3rd Corrected Edition.* Springer-Verlag, Berlin, Germany, 1985.

[Zamel and Budd, 1993] Nabil M. Zamel and Timothy A. Budd. Integrating Constraints into a Multiparadigm Language. In *Proceedings of InfoScience '93*, pages 402–409, Seoul, Korea, October 1993.

APPENDICES

Appendix A
The DeltaBlue Constraint Solver

This appendix gives a brief introduction to the DeltaBlue constraint solver. It begins by describing the characteristics of this algorithm and how it solves its constraints. It then covers how the algorithm can be used by listing its services and showing an example that uses the algorithm to solve some constraints. The appendix closes by showing how Electra hides all the unnecessary details of the algorithm to simplify the declaration of constraints.

DeltaBlue is an incremental multi-way constraint-satisfaction algorithm [Maloney, 1991, Freeman-Benson *et al.*, 1990b, Sannella *et al.*, 1993]. It was originally devised by Bjorn Freeman-Benson [Freeman-Benson *et al.*, 1992, Freeman-Benson *et al.*, 1990a]. The first implementation of DeltaBlue was done in Smalltalk, but later it was reimplemented in Lisp, C, and C++ by Borning and his group at the University of Washington. This algorithm solves systems of constraints using a technique known as *local propagation.* Local propagation simply means that once a value of a variable is determined then it is propagated along the constraint network for the purpose of utilizing it in finding solutions for other constraints that have that variable as a participant. This technique requires that each constraint must provide methods for determining the values of its participants. For example, the constraint:

$$A + B = C$$

must provide methods for determining the value of each variable if the other two are known. The methods that are needed for the above constraint are:

- If the variables A and B are known then the method to use is

$$C \leftarrow A + B$$

- If the variables A and C are known then the method to use is

$$B \leftarrow C - A$$

- If the variables B and C are known then the method to use is

$$A \leftarrow C - B$$

If the values of the variables A or B were to be changed then the system could maintain the above constraint by executing the first method yielding a new value for the variable C. If the variable C were participating in another constraint such as the constraint

$$C + D = E$$

then the new value of the variable C would be propagated to this constraint in order to calculate a new value for the variable D, and this process would continue until the change had propagated through the constraint network.

Solvers that are built using the local propagation technique are limited in their capabilities. They cannot solve all possible sets of constraints, especially those that represent simultaneous equations. However, they have the advantage of being easy to build, efficient, and general. Their generality stems from the fact that the defined methods for determining variable values can perform arbitrary computation.

DeltaBlue solves its constraints incrementally. This means that whenever there is a change in the set of constraints the algorithm does not start its satisfaction from scratch but rather it takes advantage of previous computations. This allows for increasing the frequency of changing the set of constraints without enormous delays in response time. This is a very useful capability especially for interactive applications that require frequent changes and fast response time such as user interfaces.

The DeltaBlue algorithm has two limitations:

- It cannot handle cycles of constraints. When a cycle is discovered, the algorithm signals an error asking the user for assistance. To resolve the situation, the user might have to remove one or more constraints to eliminate the cycle.
- It only permits methods with a single output variable. This limits its ability to interact with more powerful constraint solvers.

The DeltaBlue interface provides the following functions:

- Create variable.
- Create constant.
- Destroy variable.
- Print variable.
- Create constraint.
- Destroy constraint.
- Print constraint.

These functions can be combined to create more complicated functions.

Figure A.1 shows a listing of the C function `TempertureConverter()` which we extracted from the file `TestDeltaBlue.c`. This file is provided with the DeltaBlue

package. The purpose of the listed function is to express the relationship between Celsius and Fahrenheit temperatures. Its behavior can be summarized in the following points:

1. Initialize DeltaBlue.
2. Create a variable named `C` and set it to zero.
3. Create a variable named `F` and set it to zero.
4. Create a temporary variable named `t1` and set it to one.
5. Create a temporary variable named `t2` and set it to one.
6. Create the constant number 9.
7. Create the constant number 5.
8. Create the constant number 32.
9. Create the constraint `t1 = 9 * C`, and set its strength to be required
10. Create the constraint `t1 = t2 * 5`, and set its strength to be required
11. Create the constraint `F = t2 + 32`, and set its strength to be required
12. Print the values of the variables `C` and `F` after the above constraints have been inserted.
13. Change the value of the variable `C` to 0.
14. Print the values of the variables `C` and `F`.
15. Change the value of the variable `F` to 212.
16. Print the values of the variables `C` and `F`.
17. Change the value of the variable `F` to -40.
18. Print the values of the variables `C` and `F`.
19. Change the value of the variable `F` to 70.
20. Print the values of the variables `C` and `F`.

In order to increase the degree of declarativeness that is expressed by the constraint construct, the compiler-solver interface that is provided by Electra relieves the programmer from worrying about tedious details such as initializing DeltaBlue and the creation of variables and constraint components. Figure A.2 lists an Electra program that performs the same functionality as the one performed by the DeltaBlue C code. Notice how the constraint `CandF` declaratively expresses the conversion relation.

```
void TempertureConverter()
{
    Variable celcius, fahrenheit, t1, t2, nine, five, thirtyTwo;
    Constraint addC, multC1, multC2;

    InitDeltaBlue();
    celcius = Variable_Create("C", 0);
    fahrenheit = Variable_Create("F", 0);
    t1 = Variable_Create("t1", 1);
    t2 = Variable_Create("t2", 1);
    nine = Variable_CreateConstant("*const*", 9);
    five = Variable_CreateConstant("*const*", 5);
    thirtyTwo = Variable_CreateConstant("*const*", 32);

    multC1 = MultiplyC(celcius, nine, t1, S_required);
    multC2 = MultiplyC(t2, five, t1, S_required);
    addC = AddC(t2, thirtyTwo, fahrenheit, S_required);

    Variable_Print(celcius); printf(" = ");
    Variable_Print(fahrenheit); printf("\n\n");

    printf("Changing celcius to 0:\n ");
    Assign(celcius, 0);
    Variable_Print(celcius); printf(" = ");
    Variable_Print(fahrenheit); printf("\n\n");

    printf("Changing fahrenheit to 212:\n ");
    Assign(fahrenheit, 212);
    Variable_Print(celcius); printf(" = ");
    Variable_Print(fahrenheit); printf("\n\n");

    printf("Changing celcius to -40:\n ");
    Assign(celcius, -40);
    Variable_Print(celcius); printf(" = ");
    Variable_Print(fahrenheit); printf("\n\n");

    printf("Changing fahrenheit to 70:\n ");
    Assign(fahrenheit, 70);
    Variable_Print(celcius); printf(" = ");
    Variable_Print(fahrenheit); printf("\n\n");
}
```

Figure A.1. DeltaBlue Code Expressing the Relationship Between C and F.

```
var
   C, F: integer;

 constraint CandF : required;
  assert
     (F − 32) ∗ 5 = C ∗ 9  ;
 end;

begin
   print(" C  ", C , " =  F ", F , "\n");

   print("Changing Celcius to 0. \n");
   C := 0;
   print(" C  ", C , " =  F ", F , "\n");

   print("Changing Fahrenheit to 212. \n");
   F := 212;
   print(" C  ", C , " =  F ", F , "\n");

   print("Changing Celcius to -40. \n");
   C := −40;
   print(" C  ", C , " =  F ", F , "\n");

   print("Changing Fahrenheit to 70. \n");
   F := 70;
   print(" C  ", C , " =  F ", F , "\n");
end;
```

Figure A.2. An Electra Code Expressing the Relationship Between C and F.

Appendix B
The Electra Language Syntax

The following is a listing of the grammer that describes the syntax of the Electra language:

⟨program⟩ ⟶ ⟨declarations⟩ **begin** ⟨statements⟩ **end;**

⟨declarations⟩ ⟶ ε
| ⟨declarations⟩ ⟨declaration⟩

⟨declaration⟩ ⟶ ⟨constdeclarations⟩
| ⟨vardeclarations⟩
| ⟨typedeclarations⟩
| ⟨functiondeclaration⟩
| ⟨classdeclaration⟩
| **include** ⟨SCONSTANT⟩ ;
| ⟨constraintdeclaration⟩

⟨constdeclarations⟩ ⟶ **const** ⟨constantDefinitions⟩

⟨constdefinitions⟩ ⟶ ⟨constdefinition⟩
| ⟨constdefinitions⟩ ⟨constdefinition⟩

⟨constdefinition⟩ ⟶ ⟨id⟩ := ⟨expression⟩ ;

⟨vardeclarations⟩ ⟶ **var** ⟨variableDefinitions⟩

⟨variableDefinitions⟩ ⟶ ⟨variableDefinition⟩
| ⟨variableDefinitions⟩ ⟨variableDefinition⟩

⟨variableDefinition⟩ ⟶ ⟨idlist⟩ : ⟨type⟩ ;

⟨idlist⟩ ⟶ ⟨id⟩
| ⟨idlist⟩ , ⟨id⟩

⟨typedeclarations⟩ ⟶ `type` ⟨typeDefinitions⟩

⟨typeDefinitions⟩ ⟶ ⟨typeDefinition⟩
| ⟨typeDefinitions⟩ ⟨typeDefinition⟩

⟨typeDefinition⟩ ⟶ ⟨id⟩ : ⟨type⟩ ;

⟨type⟩ ⟶ ⟨id⟩
| ⟨id⟩ [⟨typelist⟩]
| `function` ⟨opttypelist⟩
| `function` ⟨opttypelist⟩ `->` ⟨type⟩

⟨opttypelist⟩ ⟶ ()
| (⟨typelist⟩)

⟨typelist⟩ ⟶ ⟨storageForm⟩ ⟨type⟩
| ⟨typelist⟩ , ⟨storageForm⟩ ⟨type⟩

⟨functiondeclaration⟩ ⟶ ⟨functionHead⟩ ⟨declarations⟩ ⟨body⟩ ;
| ⟨guardedFuncHeadAndGuard⟩
⟨declarations⟩
⟨body⟩ ;

⟨guardedFuncHeadAndGuard⟩ ⟶ ⟨guardedFuncHead⟩ ⟨guard⟩ ;

⟨functionHead⟩ ⟶ ⟨functionname⟩ ⟨valueArguments⟩
⟨optReturnType⟩ ;

⟨guardedFuncHead⟩ ⟶ ⟨guardedFuncName⟩ ⟨valueArguments⟩
⟨optReturnType⟩ ;

⟨guardedFuncName⟩ ⟶ `guarded function` ⟨id⟩ ⟨typeArguments⟩

⟨functionname⟩ ⟶ `function` ⟨id⟩ ⟨typeArguments⟩

⟨guard⟩ ⟶ `[` ⟨expression⟩ `]`

⟨typeArguments⟩ ⟶ ε
`[` ⟨argumentList⟩ `]`

⟨argumentList⟩ ⟶ ⟨storageForm⟩ ⟨idlist⟩ : ⟨type⟩
⟨argumentList⟩ , ⟨storageForm⟩ ⟨idlist⟩ : ⟨type⟩

⟨storageForm⟩ ⟶ ε
| `byName`
| `byRef`

⟨valueArguments⟩ ⟶ ()
| (⟨argumentlist⟩)

⟨optReturnType⟩ ⟶ ε
| ⟨returnType⟩

⟨returnType⟩ ⟶ `->` ⟨type⟩

⟨classdeclaration⟩ ⟶ ⟨classheading⟩
⟨excludeStatement⟩
⟨declarations⟩ `end;`

⟨classheading⟩ ⟶ ⟨classQualifications⟩ ;
| ⟨classQualifications⟩ `of` ⟨id⟩ ;
| ⟨classQualifications⟩ `of` ⟨id⟩ `[` ⟨typelist⟩ `]` ;

⟨classQualifications⟩ ⟶ ⟨classStart⟩ ⟨typeArguments⟩

⟨classStart⟩ ⟶ `class` ⟨id⟩

⟨excludeStatement⟩ ⟶ ε
| **exclude** ⟨idlist⟩ ;

⟨constraintdeclaration⟩ ⟶ ⟨fixableConstraint⟩
| ⟨satisfiableConstraint⟩

⟨fixableConstraint⟩ ⟶ ⟨fixableConsHeader⟩
⟨fixableConsAssertion⟩
⟨fixPart⟩
end;

⟨fixableConsHeader⟩ ⟶ **constraint** ⟨id⟩ ;

⟨fixableConsAssertion⟩ ⟶ **assert** ⟨expression⟩ ;

⟨fixPart⟩ ⟶ **fix** ⟨nonReturnStatements⟩

⟨satisfiableConstraint⟩ ⟶ ⟨satisfiableConsHeader⟩
⟨satisfiableConsAssertion⟩
end;

⟨satisfiableConsHeader⟩ ⟶ **constraint** ⟨id⟩ : ⟨strength⟩ ;

⟨satisfiableConsAssertion⟩ ⟶ **assert** ⟨andSatAssertion⟩ ;

⟨andSatAssertion⟩ ⟶ ⟨satAssertion⟩
| ⟨andSatAssertion⟩ **&** ⟨satAssertion⟩

⟨satAssertion⟩ ⟶ (⟨satAssertion⟩)
| ⟨satExpression⟩ ⟨RelationalOP⟩ ⟨satExpression⟩

⟨satExpression⟩ ⟶ ⟨satTerm⟩
| ⟨satExpression⟩ + ⟨satTerm⟩
| ⟨satExpression⟩ - ⟨satTerm⟩

⟨satTerm⟩ ⟶ ⟨satFactor⟩

| ⟨satTerm⟩ * ⟨satFactor⟩
| ⟨satTerm⟩ / ⟨satFactor⟩
| - ⟨satFactor⟩

⟨satFactor⟩ ⟶ (⟨satExpression⟩)
| ⟨id⟩
| ⟨ICONSTANT⟩

⟨strength⟩ ⟶ **required**
| **strong**
| **medium**
| **weak**

⟨body⟩ ⟶ **begin** ⟨statements⟩ **end**
| **beginend**

⟨statements⟩ ⟶ ⟨statement⟩ ;
| ⟨statements⟩ ⟨statement⟩ ;

⟨nonReturnStatements⟩ ⟶ ⟨nonReturnStatement⟩ ;
| ⟨nonReturnStatements⟩
⟨nonReturnStatement⟩ ;

⟨statement⟩ ⟶ ⟨reference⟩ := ⟨expression⟩
| **return**
| **return** ⟨expression⟩
| **begin end**
| **begin** ⟨statement⟩ **end**
| **if** ⟨expression⟩ **then**
⟨statement⟩
| **if** ⟨expression⟩ **then**
⟨statement⟩
else ⟨statement⟩
| **while** ⟨expression⟩ **do** ⟨statement⟩
| **for** ⟨expression⟩ **do** ⟨nonReturnStatement⟩
| **for** ⟨expression⟩ **to** ⟨expression⟩

```
                    do ⟨nonReturnStatement⟩
                  | for ⟨reference⟩ := ⟨expression⟩
                    to ⟨expression⟩
                    do ⟨statement⟩
                  | ⟨procedureCall⟩
                  | ε

⟨nonReturnStatement⟩ ⟶ ⟨reference⟩ := ⟨expression⟩
                  | return
                  | return ⟨expression⟩
                  | begin end
                  | begin ⟨nonReturnStatement⟩ end
                  | if ⟨expression⟩ then
                    ⟨nonReturnStatement⟩
                  | if ⟨expression⟩ then
                    ⟨nonReturnStatement⟩
                   else ⟨nonReturnStatement⟩
                  | while ⟨expression⟩ do ⟨nonReturnStatement⟩
                  | for ⟨expression⟩ do ⟨nonReturnStatement⟩
                  | for ⟨expression⟩ to ⟨expression⟩
                    do ⟨nonReturnStatement⟩
                  | for ⟨reference⟩ := ⟨expression⟩
                    to ⟨expression⟩
                    do ⟨nonReturnStatement⟩
                  | ⟨procedureCall⟩
                  | ε

⟨optexpressionList⟩ ⟶ ε
                  | ⟨expressionList⟩

⟨expressionList⟩ ⟶ ⟨expression⟩
                  | ⟨expressionList⟩ , ⟨expression⟩

⟨expression⟩ ⟶ ⟨andExpression⟩
                  | ⟨expression⟩ | ⟨andExpression⟩
```

⟨andExpression⟩ ⟶ ⟨notExpression⟩
| ⟨andExpression⟩ & ⟨notExpression⟩

⟨notExpression⟩ ⟶ ⟨relationalExpression⟩
| ~ ⟨notExpression⟩
| ⟨reference⟩ `is` ⟨id⟩
| ⟨reference⟩ `is` ⟨id⟩ (⟨idlist⟩)

⟨relationalExpression⟩ ⟶ ⟨binaryExpression⟩
| ⟨binaryExpression⟩ < ⟨binaryExpression⟩
| ⟨binaryExpression⟩ <= ⟨binaryExpression⟩
| ⟨binaryExpression⟩ > ⟨binaryExpression⟩
| ⟨binaryExpression⟩ >= ⟨binaryExpression⟩
| ⟨binaryExpression⟩ = ⟨binaryExpression⟩
| ⟨binaryExpression⟩ <> ⟨binaryExpression⟩
| ⟨binaryExpression⟩ == ⟨binaryExpression⟩
| ⟨binaryExpression⟩ ~= ⟨binaryExpression⟩
| ⟨reference⟩ <- ⟨binaryExpression⟩

⟨binaryExpression⟩ ⟶ ⟨plusExpression⟩
| ⟨binaryExpression⟩ ⟨BINARYOP⟩
⟨plusExpression⟩

⟨plusExpression⟩ ⟶ ⟨timesExpression⟩
| ⟨plusExpression⟩ + ⟨timesExpression⟩
| ⟨plusExpression⟩ - ⟨timesExpression⟩

⟨timesExpression⟩ ⟶ ⟨functionCall⟩
| ⟨timesExpression⟩ * ⟨functionCall⟩
| ⟨timesExpression⟩ / ⟨functionCall⟩
| ⟨timesExpression⟩ % ⟨functionCall⟩
| - ⟨functionCall⟩

⟨procedureCall⟩ ⟶ ⟨functionCall⟩ (⟨optexpressionList⟩)
| `cfunction` ⟨id⟩ (⟨optexpressionList⟩)

⟨functionCall⟩ ⟶ ⟨basicExpression⟩

| `defined` (⟨expression⟩)
| ⟨functionCall⟩ (⟨optexpressionList⟩)
| `cfunction` ⟨id⟩ (⟨optexpressionList⟩)

⟨basicExpression⟩ ⟶ ⟨reference⟩
| ⟨ICONSTANT⟩
| ⟨RCONSTANT⟩
| ⟨SCONSTANT⟩
| (⟨expression⟩)
| ⟨functionExpressionHead⟩
⟨declaration⟩
⟨body⟩
| ⟨basicExpression⟩ [⟨typelist⟩]
| [⟨expressionList⟩]

⟨reference⟩ ⟶ ⟨id⟩
| ⟨id⟩ : ⟨type⟩
| ⟨functionCall⟩ . ⟨id⟩

⟨functionExpressionHead⟩ ⟶ `function` ⟨valueArguments⟩ ⟨optReturnType⟩

⟨id⟩ ⟶ ⟨letter⟩ ⟨alphanumeric⟩

⟨alphanumeric⟩ ⟶ ⟨letter⟩
| ⟨digit⟩
| ⟨alphanumeric⟩ ⟨letter⟩
| ⟨alphanumeric⟩ ⟨digit⟩

⟨digit⟩ ⟶ `0` | `1` | `2` | `3` | `4`
`5` | `6` | `7` | `8` | `9`

⟨ICONSTANT⟩ ⟶ ⟨digit⟩
| ⟨digits⟩ ⟨digit⟩

⟨RCONSTANT⟩ ⟶ ⟨digits⟩ . ⟨digits⟩
| ⟨digits⟩ . ⟨digits⟩ `E` ⟨sign⟩ ⟨digits⟩

⟨sign⟩ ⟶ - | +

⟨digits⟩ ⟶ ⟨digit⟩
| ⟨digits⟩ ⟨digit⟩

⟨letter⟩ ⟶ A | B | C | D | E | F | G | H | I | J
| K | L | M | N | O | P | Q | R | S | T
| U | V | W | X | Y | Z
| a | b | c | d | e | f | g | h | i | j
| k | l | m | n | o | p | q | r | s | t
| u | v | w | x | y | z

⟨SCONSTANT⟩ ⟶ ' ' ⟨alphanumeric⟩ ' '