An Investigation of Constraint-Based Component-Modeling for Knowledge Representation in Computer-Aided Conceptual Design

by

Mark Andrew Kolb

S.B., Massachussets Institute of Technology, 1984

S.M., Massachussets Institute of Technology, 1986

SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

Doctor of Philosophy

in

Aeronautics and Astronautics

at the

Massachusetts Institute of Technology

February 1990

Signature of Author _____

Department of Aeronautics and Astronautics
January 12, 1990

Certified by _____

Professor Mark Drela
Department of Aeronautics and Astronautics
Thesis Supervisor

Certified by _____

Dr. Antonio L. Elias
Orbital Sciences Corporation
Thesis Committee

Certified by _____

Professor Earll M. Murman
Department of Aeronautics and Astronautics
Thesis Committee

Certified by _____

Professor Robert W. Simpson
Department of Aeronautics and Astronautics
Thesis Committee

Accepted by _____

Professor Harold Y. Wachman
Chairman, Departmental Graduate Committee

An Investigation of Constraint-Based Component-Modeling for Knowledge Representation in Computer-Aided Conceptual Design

by

Mark Andrew Kolb

Submitted to the Department of Aeronautics and Astronautics
on January 12, 1990
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy in Aeronautics and Astronautics.

Abstract

The earliest computer programs used for engineering design focused on *detailed geometric design*. Subsequently, computer programs for algorithmically performing the *preliminary design* of specific well-defined classes of objects became commonplace. However, due to the need for extreme flexibility, it appears unlikely that conventional programming techniques will prove fruitful in developing computer aids for engineering *conceptual design*.

The use of symbolic processing techniques, such as object-oriented programming and constraint propagation, facilitates such flexibility. Object-oriented programming allows programs to be organized around the objects and behavior to be simulated, rather than around fixed sequences of function- and subroutine-calls. Constraint propagation allows declarative statements to be understood as designating multi-directional mathematical relationships among all the variables of an equation, rather than as uni-directional assignment to the variable on the left-hand side of the equation, as in conventional computer programs.

The research presented here has concentrated on applying object-oriented programming and constraint propagation to the development of a general-purpose computer aid for engineering conceptual design. Object-oriented programming techniques are utilized to implement a user-extensible database of design components. The mathematical relationships which model both the geometry and physics of these components are managed via constraint propagation. In addition to this component-based hierarchy, special-purpose data structures are provided for describing component interactions and supporting state-dependent parameters.

In order to investigate the utility of this approach to conceptual design, three sample design problems from the field of aerospace engineering have been implemented using the prototype design tool, *Rubber Airplane*: a long-endurance surveillance aircraft, a subsonical transport aircraft, and a small-payload launch vehicle. The additional level of organizational structure obtained by representing design knowledge in terms of components is observed to provide greater convenience to the program user, and to result in a database of engineering information which is easier both to maintain and to extend.

Thesis Supervisor: Mark Drela
Title: Assistant Professor of Aeronautics and Astronautics

Acknowledgements

I wish to acknowledge the assistance and support of the following individuals, and express to them my sincere gratitude and appreciation for their contributions to this work:

Antonio Elias, who has provided much guidance over the years; the contributions of this work are in great part due to his original inspiration.

Prof. Mark Drela, who generously consented to adopt this "orphaned" project and serve as thesis supervisor; his encouragement and fresh perspective on the subject matter have made this dissertation a more complete document than it could have been without his participation.

Profs. Robert Simpson and Earll Murman, the remaining members of my Thesis Committee, for their willingness to share both their time and their advice.

Prof. Norman Ham, who served on both my General Examination and Thesis Defense Committees, and Profs. David Gossard and John Dugundji, who also served on my Thesis Defense Committee; their service is greatly appreciated.

The Advanced Plans and Programs Office of the NASA Ames Research Center, which sponsored this research, and, more specifically, George Kidwell, technical monitor for my grant, who provided technical and moral support, in addition to NASA's financial support.

Harold "Guppy" Youngren, who graciously donated his time, his knowledge, and even his code, that I might have a better understanding of the vortex-lattice method, upon which to base the implementation presented here.

Ed Rogan and Parker McElveen, both formerly of Lockheed-Georgia, for their early interest in and support of my work; their enthusiasm has been a source of encouragement throughout this effort.

John Pararas, Lyman Hazelton, and Joakim Karlsson, my colleagues at the MIT Flight Transportation Laboratory, who, through their help, suggestions, enthusiasm, and friendship, have made this time spent at FTL most enjoyable.

The staff and congregations of Cambridgeport Baptist Church and Saint James's Episcopal Church for their guidance, their fellowship, and their prayers.

My parents, who have each in their own way contributed to my personal growth and education.

Most especially, to my wife Jean, for her loving care, undying patience, and steadfast devotion. *Whom we love best, to them we can say least.*

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Problem Statement

The goal of engineering design is the description of a physical device which performs a given task while satisfying the limitations of the governing specifications. Engineering design can typically be characterized by three stages:

Conceptual Design: Determination of the feasibility of various potential designs, and identification of the dominant tradeoffs.

Preliminary Design: Thorough verification of the most promising designs, with the objective of identifying the design best suited for full analysis.

Detailed Design: Development of a complete description of the final design.

In aerospace design, where considerable resources are required for detailed design, the costs associated with committing such resources have naturally led to increased emphasis on improving productivity and reliability in the conceptual and preliminary design stages. One ready source for such improvements is the use of computers, due to their enhanced speed and accuracy in numerical calculations, as well as their superior ability to manage complexity.

While the benefits of computers for numerical applications have long been recognized and exploited, it is only recently that computers have begun to be utilized to help manage the complexity of large-scale engineering problems. While database technology has aided information management for decades, the use of computers to help account for causality and other dependencies among data in such applications as planning, fault diagnosis, and design is comparatively new.

In engineering design, the earliest applications of computers were in performing discipline-specific analyses (e.g., aerodynamics, structures), through a fixed sequence of calculations

which modeled the relevant physical phenomena. The task of integrating the results of such analyses, however, was left to the human designers.

The first programs intended specifically for design were in the area of geometric design (i.e., drafting), beginning with the pioneering work of Ivan Sutherland in the early 1960's [36]. Such Computer-Aided Design (CAD) programs typically allow the user to manipulate various points, lines, and curves in three dimensions via a set of straightforward transformations (e.g., translation, rotation, extrusion) in order to represent the geometry of an engineering artifact. This simple and intuitive approach affords great flexibility to the user, and is therefore applicable to a wide range of mechanical design tasks.

However, the CAD approach is limited to geometric analysis, and is therefore best suited to problems in constructive geometry and the detailed design of, for example, the interfaces between design components which have been previously sized—on the macroscopic scale—by other means. This is because, in the overall design of many engineering artifacts, geometry is itself dependent upon multiple underlying analyses. In aircraft design, for example, wing geometry (e.g., position, planform dimensions, cross-section, skin thickness, arrangement of internal supports) is itself dependent upon such factors as choice of materials, cruising speed and altitude, wing weight, total vehicle weight, and location of the wing with respect to the vehicle center of gravity. While efforts have been made to integrate certain forms of engineering analysis into conventional CAD packages (most notably finite-element thermal and structural analyses), current CAD systems do not offer general support for non-geometric analyses.

The question of interest, then, is how might the support and flexibility afforded by CAD systems in detailed geometric design be made available to the engineer faced with a new problem in conceptual or preliminary design?

The research described herein is focused on considering answers to this question, specifically with regard to conceptual design. The issue of flexibility is particularly critical in conceptual design, for, insofar as the goal of conceptual design is to explore alternatives, the computational requirements cannot be specified at the outset. A computer aid for conceptual design must therefore be able to compensate for this initial lack of knowledge, while still being able to support the variety of engineering analyses (e.g., geometry, performance, structures, aerodynamics, propulsion) required to support the design task. This work will examine two specific approaches to providing the required support and flexibility:

- object-oriented programming, and
- constraint propagation.

Object-oriented programming techniques will be exploited as a means of flexible data representation, allowing for the incremental description and development of the design problem.

Constraint propagation is utilized as a means of removing directionality from the mathematical analyses, thus providing additional flexibility.

These two approaches are combined in a prototype computer-based design tool, *Rubber Airplane*, which allows solutions to engineering design problems to be incrementally developed through the manipulation of constraint-based descriptions of the physical components and analytical models which comprise them. The resulting models allow the designer to propose various alternative solutions to a design task—perhaps based on varying computational paths—in order to test the consequences of design decisions.

1.2 Conventional Approaches to Computer-Assisted Design

Before examining the alternative approaches investigated in this research, it is perhaps advantageous to first review past efforts in the area of computer aids for engineering design. As indicated above, the earliest applications for computers in the area of design were basically drafting tools. Because shape information and spatial transformations can be readily represented numerically, geometry-oriented CAD systems, most of which employ fairly conventional programming techniques, are able to provide a high degree of flexibility. However, the level of detail provided by such systems is generally best suited to the later stages of engineering design.

With respect to conceptual and preliminary design, however, the use of computers (in the domain of aerospace engineering, at least) has historically centered around conventional sequential programs for performing a predetermined series of calculations which model the geometry and physics of a given design. Typically, a set of input values for such parameters as basic dimensions, desired performance characteristics, and mission profile is required, based upon which various structural, aerodynamic, performance, and propulsion analyses may be performed. The sequence of these analyses is fixed, as is the set of variables for which input values may be specified (i.e., the **base variables**). For this reason, such programs are fairly limited in application, being restricted to vehicles derived from a particular baseline design (e.g., twin-engine, propeller-driven general aviation aircraft [18]), or, in some cases, to a more general class of vehicle-types (e.g., cargo transport aircraft [19]). This limited applicability often makes such programs unacceptable for conceptual design, where implicit assumptions on basic vehicle configuration and/or mission profile may be overly restrictive.

In recognition of this deficiency, various attempts to widen the range of applicability of such programs have focused on the integration of large numbers of independently-developed, discipline-specific programs, linked by a common database of design variables (e.g., the ACSYNT program in use at NASA's Ames Research Center [16]), or, more recently, by means of an expert system which oversees the execution of the programs (e.g., Boeing

Aerospace's Preliminary Design Tool [7,26]). One shortcoming of the database approach, however, is that the user is generally required to specify the interface among the various modules (i.e., which subprograms to use, and the order in which to apply them). This usually requires that each user have some familiarity with the capabilities of *all* of the various database modules; unfortunately, this requirement makes such systems difficult both to use and to maintain. For example, the Aircraft Design & Analysis System (ADAS), developed at the Delft University of Technology [2], provides the user with access to an extensive database of single-equation design relationships associated with the preliminary design of subsonic aircraft, as derived from the text of Reference [38].[1] However, while it can be useful to manipulate the design program at the level of individual equations, use of ADAS requires manual specification of the sequencing of the individual relationships to be used; for large design problems, this can be a formidable task, indeed.

On the other hand, the use of expert systems to manage the interface between subprograms shows much promise for alleviating such burdens on the user. The Engineous program [37], for instance, is an expert system which links a set of engineering programs for the design of aircraft jet engines, including structural, aerodynamic, and thermal analyses. The rules which comprise the Engineous program are responsible not only for guiding the sequence of application of the individual subprograms, but also for monitoring the various programs' outputs and arranging for nested, iterative cycling among the subprograms, based on performance requirements.

Ultimately, though, the flexibility of both the database and the expert system approaches to subprogram integration are limited by the inherent directionality of the constituent subprograms. While the ordering between subprograms is variable, the sequence of operations within a single subprogram is fixed; thus, while there may be multiple sets of base variables consistent with all feasible arrangements of the subprograms, complete freedom in choice of base variables is not possible. This observation is part of the motivation for the research described here. Practically speaking, though, most large engineering organizations have considerable resources invested in the sorts of discipline- and vehicle-specific programs discussed above; in such cases, program integration is a much more economical alternative to improving computer support for design than development of experimental systems which rely on more advanced programming techniques.

[1] Note that by relying on a single source for the analytical methods, problems with compatibility among modules are reduced.

1.3 Object-Oriented Programming

1.3.1 Description

Conventional computer languages allow the programmer to represent the solution to a problem as a sequence of instructions or operations to be performed in order to compute the desired result. These operations are performed on a set of variables (which serve as labels or place-holders), and may involve simple arithmetic calculations, or more complex conditional or iterative constructs; additionally, subroutines or functions may typically be defined to capture frequently-used subsequences of instructions. Nevertheless, such programs can ultimately be reduced to a series of sequential operations which implement the desired analysis.

An alternative approach to computer programming, referred to as object-oriented programming, relies instead on a description of the programming problem in terms of objects to be manipulated, rather than variables to which values are assigned. Objects are defined by **instance variables** whose values characterize their properties, and by **methods** which describe the operations which may be performed on them. Together, a set of corresponding instance variables and methods delineate a particular **class** of objects, individual **instances** of which are actually manipulated by the program. And while instances share the same methods and instance variables, the specific values of these instance variables can vary among instances.

To demonstrate by way of an example, consider the class of objects `automobile`. This object-class could be described in terms of instance variables such as engine-size, the number of cylinders in the engine, fuel capacity, fuel available, empty weight, total weight, number of doors, number of passengers, color, gear shift location, etc., as well as methods for adding and removing passengers and fuel, accelerating and decelerating, etc. (The exact details would depend upon the nature of the problem being modeled.) Based upon this class, individual `automobile` instances would then be created: one instance might be a red four-door with a V8 engine, while a second might be a blue two-door with only four cylinders. Thus, the first instance would have the values "red", "4", and "8" for its instance variables "color", "number of doors", and "number of cylinders", respectively; the second instance would have the values "blue", "2", and "4" for those instance variables. The various methods mentioned above could then be applied to these instances as part of, say, a traffic simulation to monitor fuel consumption and introduction of pollutants, based on traffic signal locations. Thus, application of the acceleration method to a particular instance would be required to update its instance variable for available fuel, just as adding or removing a passenger would affect the instance's "total weight" instance variable.

An additional useful feature of most object-oriented programming languages is the abil-

ity to specify **inheritance** paths when defining new object classes. Inheritance allows one to describe new object classes in terms of previous classes, and aids program modularity by allowing code common to multiple classes to be shared among them. Classes which are inherited are referred to as **superclasses**, while inheriting classes are referred to as **subclasses**. Typically, subclasses inherit all of the instance variables and methods defined for their superclasses, though it is usually possible to override default values of inherited instance variables, as well as provide subclass-specific methods which replace or modify the inherited methods. Such modification of inherited properties is referred to as **specialization**. For this reason, any given class is considered to be "more specific" than its superclasses (since it may specialize them), and "less specific" than its subclasses.

Thus, if one wished to extend the traffic simulation example introduced above to include other vehicle types, it might be appropriate to create a lower-level `moving-object` superclass, which all of the vehicle types could inherit. This class could provide instance variables for such common attributes as position and velocity, as well as methods for accelerating and decelerating. Subclasses of `moving-object` might then include classes such as `motorcycle`, `bicycle`, and `truck`, in addition to the original `automobile` class. Some object-oriented programming languages also permit **multiple inheritance** (i.e., the ability of a class to specify more than one superclass) in which case it might be advisable to introduce an additional `ground-vehicle` superclass, which might provide such instance variables as "color", "number of passengers", and "number of wheels", as well as methods for adding and removing passengers. Each subclass could then specify additional vehicle-specific instance variables, such as the "number of doors", "number of cylinders", and "gear shift location" "number of wheels" instance variables suggested above for the `automobile` class. And while the acceleration and deceleration methods for the `moving-object` class might only change an instance's "velocity" instance variable, the `automobile`, `truck`, and `motorcycle` classes would require a specialized acceleration method which also decreases an instance's "available fuel" instance variable.[2]

The primary advantage of object-oriented programming is, therefore, in the area of **data abstraction**, the ability to encapsulate some subset of a program's functionality in such a way that interaction with other parts of the program is simplified. Specifically, object-oriented programming allows parts of a program to be represented as objects, whose interfaces are specified by the object's methods: no knowledge of internal structure or representation is required by users of these objects. In addition, by providing a common set of methods for a group of object-types, the program code which manipulate these objects need not even know what specific kind of object is being manipulated: the objects them-

[2]Though no such specialized method would be required for the `bicycle` class, nor would any of the proposed classes require a specialized fuel-changing deceleration method.

selves thus become responsible for the details of their responses to a request to perform a given operation (i.e., method). Note, then, that in the example traffic simulation introduced above, all of the various vehicle-types provide acceleration and deceleration methods. When the program wishes to change the speed of a vehicle or set of vehicles, it merely requests that the vehicles apply the corresponding methods; it need not determine which vehicles are trucks and which are bicycles in order to perform the appropriate vehicle-specific acceleration procedures, since these procedures are already associated with the objects to which they apply. Inheritance and specialization can therefore be viewed as means for facilitating data abstraction, insofar as their use requires the specification of a common interface across multiple object-classes, due to the inherited methods.

Finally, as suggested above, a primary requirement of a computer system for supporting conceptual design is flexibility. An additional advantage of object-oriented programming is the ability to make available a large variety of object-types (i.e., classes), which may be instantiated as needed. For design applications, then, one might consider providing classes which represent the various possible design components, which the user may incorporate into his design as he sees fit. The system might then also provide methods which allow these components to be manipulated in order to size and position them, in accordance with the appropriate engineering analyses.

In this way, the object-oriented approach makes feasible computer aids for design which provide a flexible environment for representing the sort of incrementally evolving design which typifies the conceptual design process. The next point to consider, then, is the means by which the required engineering analyses may be incorporated into such a shifting model of the design problem. As these analyses are primarily mathematical in nature, constraint propagation is suggested as a flexible means for managing the mathematical relationships which govern the design solution.

1.3.2 Historical Background

Object-oriented programming has many of its foundations in the programming language, SIMULA [10]. The first completely object-oriented programming language was Smalltalk [14]; originally, Smalltalk ran only on special-purpose, dedicated hardware, and featured an interactive, window-based programming environment which was itself implemented primarily in Smalltalk. Based on the success of the Smalltalk language, various object-oriented extensions to other languages have been implemented, such as the Flavors [39] system for LISP, which pioneered the use of multiple inheritance. More recent examples of such extensions include the Common Lisp Object System (CLOS) [3], which features so-called

multi-methods[3], and the C++ extension to the C programming language [9].

Object-oriented programming is also related to Artificial Intelligence research in knowledge representation on the theory of frames [25]. In Minsky's original paper on the subject, a **frame** is considered to be "a data-structure for representing a stereotyped situation, like being in a certain kind of living room, or going to a child's birthday party". Thus, frames are intended to provide a means for associating related pieces of information, which may be both descriptive and procedural in nature. In object-oriented programming, descriptive knowledge is normally represented by instance variables, while methods are used to represent procedural information. The concept of class-instantiation is less directly applicable to frames, however, since frames typically include descriptive information which is invariant from situation to situation, as well as that which is dependent upon circumstances.[4] Finally, it should also be noted that in the field of rule-based systems, the term "frame" has adopted a more restrictive meaning; here, a frame refers specifically to a data structure with various named **slots** which are pointers to either static or dynamic values, as well as a set of associated rules which are triggered by references to these slots.

Numerous applications in simulation, AI, systems programming, and graphics have applied the principles of object-oriented programming. The following section will describe several object-oriented programs developed for engineering design.

1.3.3 Object-Oriented Design Tools

As indicated above, a major focus of conceptual design is the sizing and positioning of the various physical components which comprise an engineering artifact, such that the resulting combination performs the desired task. In light of this, the applicability of object-oriented programming to representing this aspect of the design process is clear: a set of objects (i.e., the components) is being manipulated (i.e., "sizing" and "positioning"). Furthermore, at least some of these manipulations are object-specific; though a single, universal representation for the "positioning" operation is conceivable, the process of "sizing" is typically component-dependent (e.g., sizing the tires of a vehicle has little in common with the possible approaches for sizing, say, a lifting surface). Thus, there is a need for specialization. It is also possible to group sets of similar objects (e.g., wings, horizontal and vertical stabilizers, winglets, and canards are all forms of airfoils), suggesting opportunities to employ inheritance.

In recent years, a number of computer aids for engineering design have been developed

[3] A multi-method is a method which dispatches based on the types of one *or more* of its arguments, in contrast to more standard methods which are associated with just a single object-class

[4] However, this distinction is disappearing, as well. CLOS, for example, allows for the definition of class variables, whose values are shared by all instances of a class (as opposed to instance variables, for which each instance of a class has an individual—though not necessarily unique—value).

which attempt to exploit one or more features of object-oriented programming. For the purpose of contrast with the present work, a brief review of these programs is presented below.

GRADE

The GRADE program, developed at Lockheed-Georgia [8], provides the user with the means to specify the geometry of the various components of a transport aircraft (e.g., wing, tail, fuselage, engines, cargo box), graphical representations of which are updated based upon the supplied dimensions. Once the component dimensions are provided, their values may be used, in combination with a description of the desired mission profile, as the inputs to a conventional design analysis program, GASP [19]. The analysis program then verifies the feasibility of the design, and can vary a subset of the input dimensions in order to optimally satisfy the mission performance requirements. Finally, these amended values for the dimensions may be used by GRADE to revise its display of the component geometries, which the designer may further modify, repeating the entire process iteratively.

GRADE does not itself utilize object-oriented programming techniques, nor is GRADE responsible for any design analysis. Instead, it serves primarily as a graphical interface to the (preexisting) GASP aircraft sizing program. The innovative contribution of GRADE, however, is that this interface is object-oriented: the user is provided with a choice of possible components to include, and is then guided through the selection of the chosen components' dimensions. GRADE is also capable of maintaining some dependency information among the components, specifically in the form of attachments. (For instance, the location of any wing-mounted engines will change as the position of the wing is varied.) Additionally, output of the underlying analysis program is presented by means of the same user-specified combination of components.

The Concept Modeler

Serrano's "Concept Modeler" [32] is the successor to his earlier research on the constraint-based MATHPAK program (see Section 1.4.2, below). The Concept Modeler expands upon the capabilities of MATHPAK by providing a set of basic objects which the user may select among for incorporation into a design. Associated with each object-type is a set of parameters and a set of constraints which govern these parameters. The program supports the interactive addition and removal of both objects and constraints, and a graph-based form of constraint propagation (again, see Section 1.4.2) is employed to permit redirection of constraint calculations.

The Concept Modeler employs a building-block metaphor, insofar as the user is provided with a set of basic object-types which may be instantiated and combined as desired in order

to solve a particular design problem. These object-types include: shaft, bearing, gear, link, support, pivot, ground, spring, damper, cam, follower, and vector. (The program is geared towards mechanical engineering conceptual design.) In addition, specialized interfaces are provided for connecting pairs of these objects (e.g., connecting a gear to a shaft, or a pair of gears to one another).

Like GRADE, however, The Concept Modeler is limited in its application of object-oriented programming techniques. While the program itself is written in object-oriented LISP, the program-user is not able to take advantage of the underlying object-oriented representations: no means has been implemented by which the user might define new object-types or interfaces, or add parameters to existing objects.

Commercial Systems

Whereas GRADE relies on a conventional sizing program for design analysis, and Serrano's Concept Modeler utilizes constraint propagation, two recent commercial programs for object-oriented design, ICAD Inc.'s ICAD program [29] and Wisdom Systems' Concept Modeller [24], both employ a rule-based approach for representing design relationships. Additionally, the two programs provide a complete object-oriented language for describing design components, and arranging them in part/sub-part hierarchies. Based upon a set of geometric primitives (e.g., block, cylinder, cone, extrusion, body of revolution), the user is able, by means of inheritance, to specify arbitrarily complex part descriptions, in terms of inherited classes, part-specific parameters, and rules for computing parameter values. These rules take the form of LISP expressions for computing a parameter, based upon values for other parameters, which are assumed known. Thus, these required parameters represent the antecedents of a rule, and the computed value is its consequent. Note that there are no limitations on the contents of the expressions in these rules; they may be used to model geometric as well as physical phenomena.

Relationships between parts (i.e., connections within an assembly, relative sizing, etc.) are specified by means of the aforementioned part/sub-part hierarchy. All part definitions may specify sub-parts, and the rule which governs any parameter of a part within a given part/sub-part hierarchy may reference the value of parameters of any other part within the hierarchy, by specifying the appropriate path through the part/sub-part tree. In addition, means are provided for symbolically describing the relative orientations of parts and sub-parts; specifically, one part may literally be specified as being "above", "below", "in front of", etc., another part.

While access to the capabilities of a full object-oriented language for describing design components adds a great deal of flexibility to these two programs, the use of uni-directional rules for computing parameter values limits their applications to problems in which the

design path is at least partially known in advance; i.e., the design of a given sub-assembly, as represented by a single part/sub-part hierarchy, must follow a particular sequence of operations, as embedded in the associated rules. As such, the approach is not well-suited to conceptual design, though it may be appropriate to certain classes of preliminary design tasks.[5]

The rule-based approach, though, is not without its advantages, foremost among which is its convenience for implementing demand-driven calculation—i.e., the ability to delay calculation of a given parameter until its value is needed, thus reducing the load on the computer. (Parameter values may be required for geometry display, or by request of the user.) Demand-driven calculation is implemented in rule-based systems by employing backward-chaining to perform rule evaluations: backward-chaining orders rule-application according to rule consequents, rather than rule antecedents. Thus, given a request for the value of a particular parameter, the rule which computes it is examined. If values are available for all of the parameters which are required to apply the rule, it is applied, and thus the requested value is computed. If values have yet to be computed for any of the required (antecedent) parameters, though, their rules are in turn examined for possible application, which may then trigger further rule applications, recursively. In this fashion, a chain of rule applications is established to perform the supporting calculations needed to compute the requested parameter.

In contrast, a forward-chaining system would apply a given rule immediately upon determining that values for all of the antecedent parameters are available. Since this rule application will provide a value for an additional parameter (i.e., the rule's consequent parameter), additional rule applications may result. Thus, forward-chaining causes all calculations to be performed as they become possible; backward-chaining delays calculations until they are needed in order to compute a required value. Finally, note that use of backward-chaining assumes that it is always possible to arrange the rules so that a required computation becomes possible—i.e., that a chain of rule applications can be established leading back to a set of completely known parameters.

1.4 Constraint Propagation

1.4.1 Description

As indicated at the beginning of this chapter, conventional programming languages are used to solve problems based on a sequence of instructions. The sequential nature of such languages makes computer programs based on these languages highly directional; indeed

[5]For instance, both programs have been successfully applied by engineering firms who are routinely required to provide custom solutions to fairly limited classes of design problems, based on an inventory of standardized parts (e.g., heat exchangers, boilers, fans) [29,24].

the individual instructions which comprise these programs are themselves highly directional. For example, while the simple algebraic statement, $a = b + c$, suggests a set of relationships among the three variables, a, b, and c, specifically,

$$\begin{aligned} a &= b + c \\ b &= a - c \\ c &= a - b \end{aligned}$$

the same statement, as part of a conventional computer program, represents the uni-directional assignment to the variable a of the sum of the previously-assigned values of the other two variables, b and c. Thus, although the **declarative** form of a mathematical relationship (e.g., $a = b+c$) implies multiple **imperative** forms (such as those listed above), conventional programming languages are limited to use of the single, given declarative form. If a different (i.e., imperative) form is desired, the program must be re-written.

While this approach is usually adequate for most programming problems, there are cases in which it is desirable to specify a non-directional relationship among program variables, and have the computer determine which form is appropriate to apply, based on the available information. The body of techniques adopted for implementing this alternate computational approach are collectively referred to as "constraint propagation". As such, two basic tasks are required of a constraint propagation system, specifically:

1. Derivation of the various imperative forms of a relationship based on a specified declarative form.

2. Monitoring the values of the variables governed by a relationship, and determining when to apply a particular form of the relationship in order to calculate a value for one of its parameters.

Various methods have been used for performing each of these two tasks, some of which are described below in Section 1.4.2.

The advantage to using constraint propagation to represent the mathematical equations which govern a problem is—as with object-oriented programming techniques—added flexibility. In conventional programs, the exact direction in which a relationship is applied, as well as the order in which each equation in a set of equations is executed, must be specified in advance. With constraint propagation systems, only the content of each mathematical relationship is specified in advance; the direction and sequencing of equation-application is determined by the computer at run-time, based on available data. Thus, the relationships are treated as **constraints** to be satisfied, rather than instructions to be executed.

For example, given the declarative form $a = b + c$, if values are available for both b and c, then a may be computed using the specified declarative form. If, on the other hand, values for a and b are known, then the imperative form $c = a - b$ may be used to compute a value for c. In the first case, b and c are the **base variables** of the constraint, while a is the **derived variable**. In the second case, a has become a base variable (along with b), and c is the derived variable.

Above, it is also indicated that constraint propagation may be used to determine an appropriate ordering for the application of a set of relationships. Given a set of simultaneous equations, constraint propagation may be used to determine the sequence in which the equations should be applied in order to compute values for as many derived variables as possible, based on the specified base variables. For example, given the set of constraints,

$$\begin{aligned} a &= b + c \\ c &= d - e \\ d &= c - a \end{aligned}$$

and base variables a and b, constraint propagation may begin by using the first equation to calculate a value for c, as above. Once this equation has been used to compute c, though, the third equation—in its declarative form—may next be used to compute a value for d. This value for d, in combination with the value computed for c, may then be used to compute a value for e, using an imperative form of the second equation, specifically, $e = d - c$. Thus, in addition to selecting the directions in which the relationships are applied, constraint propagation is also used to establish the order in which they are applied.

It is because of its flexibility in handling mathematical relationships that the constraint propagation approach is particularly well-suited to computer applications in conceptual design. As discussed above, computer aids for conceptual design must be able to cope with the continual evolution and refinement of the designer's model of both the design problem *and* its solution. As new ideas are developed and examined, corresponding analytical models must be introduced to facilitate this examination. Constraint propagation provides a means for flexibly managing these mathematical models in a changing environment.

First of all, constraint propagation permits the analyses to be performed to be described via declarative statements of the mathematical relationships which model the analyses,[6] but does not require that the equations be applied in this declarative form: the designer is free to choose whichever design variables he considers relevant as base variables, relying on the computer to determine the exact means by which values for the remaining output variables

[6]Note that, in the implementation described in Chapter 3, declarative constraint specifications may be based on either analytic or numerical analyses.

are calculated. In addition, it allows for the incremental addition of new analyses (or the incremental removal of unwanted analytical models) to the evolving description of the design problem, since constraint propagation may be used to determine the sequencing of the new analyses, as well.

1.4.2 Implementation Schemes

As suggested above, constraint propagation provides a means of avoiding the underlying directionality of conventional computer programs, as well as the implicit directionality of the rule-based approach described in the Section 1.3.3. A number of techniques for implementing constraint propagation have been investigated, including

- local propagation of known states,
- rule-based inversion,
- graph transformation and term-rewriting,
- symbolic algebra, and
- numerical techniques.

Each of these approaches is described below, in conjunction with the various research efforts that have applied them.

Local Propagation of Known States

In this approach, as developed by Steele [34] in what is widely regarded as the seminal work on constraint propagation, each constraint is monitored until values are available for all of the associated parameters, save one.[7] The constraint is then used to compute a value for this remaining parameter. Assignment of a value to this parameter may result in other constraints being reduced to the state of having exactly one unknown parameter, and thus these constraints become ready for application as well.

For example, consider the set of constraints,

$$\begin{aligned} F(w,x,y) &= 0 \\ G(x,y) &= 0 \\ H(x,y,z) &= 0 \end{aligned}$$

[7] Actually, there are certain cases in which a constraint may be applied even if there is more than one unknown. See, for example, in the discussion of rule-based inversion below, those cases for the multiplication operator in which only one multiplicand is known, but its value is zero.

where each of the constraints is given in so-called **normal form** (i.e., all parameters appear on the left-hand side of the equation, and the right-hand side of the equation is zero). Specifying parameters whose values are known with a superscript "K", and those whose values are unknown with a superscript "U", if initially only the value of x is known, the system may be described thus:

$$\begin{aligned} F(w^U, x^K, y^U) &= 0 \\ G(x^K, y^U) &= 0 \\ H(y^U, z^U) &= 0 \end{aligned}$$

Here, only one constraint has exactly one unknown parameter, and each of the two remaining constraints has two unknown parameters. Applying local propagation of known states, then, the second equation, involving constraint G, may be applied to compute a value for parameter y. The set of unused constraints is then reduced to

$$\begin{aligned} F(w^U, x^K, y^K) &= 0 \\ H(y^K, z^U) &= 0 \end{aligned}$$

where the parameter y is now labeled as being a known parameter. Thus, both of the remaining constraints now have exactly one unknown parameter, so that constraint F may used to compute a value for parameter w, and constraint H may be used to compute a value for parameter z.

A set of equations to be solved using local propagation of known states may therefore be viewed as a network of linked constraints, with the parameters serving as arcs in this network. If flow through this network is assumed to flow along arcs away from the constraint which computes a parameter, flow out of a constraint node is observed to become possible only when all but one of the arcs into a node carry flow. The evolution of flow through the network of constraints introduced in the example above is depicted in Figure 1.1, starting from (a) the initial state, through (b) propagation of constraint G, and concluding with (c) propagation of constraints F and H.

Note, though, that this approach is inadequate for systems of equations which include cycles. Given, for example, a pair of constraints which share the same two unknowns, local propagation is unable to solve the simultaneous equations, since neither of the two constraints has exactly one unknown. The same is true of larger systems; mathematically, however, systems of equations which contains the same number of parameters as equations are generally solvable, particularly when they represent physically realizable systems.

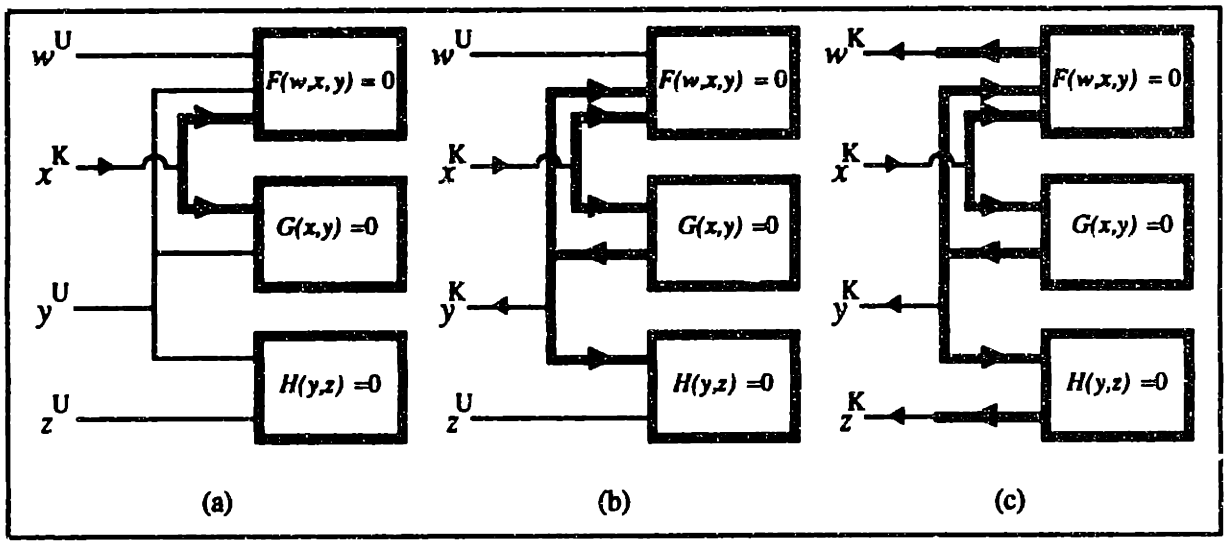

Figure 1.1: Local propagation of known states.

An advantage of local propagation, though, is its ability to keep track of dependencies within the constraint network; since computations only become possible when a given constraint has only a single unknown parameter, it is a simple matter when performing the corresponding calculation to note which constraint is responsible for the value assignment. In a similar fashion, local propagation also facilitates incorporation of retraction. When the value of a known parameter is retracted, it is marked as unknown, and any parameters computed by constraints which previously relied on the retracted parameter's value are also retracted. This retraction may, in turn, trigger further retractions. In this way, all dependent values—and *only* dependent values—are retracted, without having to resort to re-solving the entire constraint network.

Rule-Based Inversion

While local propagation of constraints allows for the automation of constraint application, it does not indicate the means by which declarative specifications of constraints may be applied imperatively. The approach taken by Steele [34] was to require the network of constraints to be built up from combinations of primitive arithmetic operations, for which individual inversion rules were provided. Supported primitive operations for integer arithmetic included addition, multiplication, minimization, maximization, and conditional equality. The inversion rules for addition, for example, are:

Addition Constraint: $c = a + b$

- *Given a and b:* $c = a + b$.
- *Given b and c:* $a = c - b$.

- *Given c and a:* $b = c - a$.

The inversion rules for multiplication, though, are somewhat more complicated:

Multiplication Constraint: $c = a \times b$

- *Given a and b:* $c = a \times b$.
- *Given* $a = 0$*:* $c = 0$.
- *Given* $b = 0$*:* $c = 0$.
- *Given b and c,* $b \neq 0$*:* $a = c/b$.
- *Given a and c,* $a \neq 0$*:* $b = c/a$.

Note that two cases are left undefined (specifically, those in which division by zero would be required). Inversion of the maximization and minimization operators also leaves certain cases undefined.[8]

In this manner, once local propagation determines that a given single-operator constraint may be ready for application, the corresponding inversion rules are consulted. When the known parameters of the constraint satisfy one of the preconditions of its inversion-rules, the corresponding calculation may be performed.

Graph Transformation and Term Rewriting

Two related approaches which also take advantage of a rule-based specification of mathematical properties are **graph transformation** and **term rewriting**. Both techniques rely on pattern-matching to invert constraints as well as propagate them, by examining a system of equations and a set of variable-assignments, and determining possible substitutions. Assigned values are substituted directly, and then rules for algebraic transformations are applied in order to isolate unknown variables, to facilitate further substitutions. If such transformations allow the value of an unknown to be computed, the corresponding assignment is made.

After substituting, the resulting system of equations is itself examined for possible substitutions. In this way, the substitution process serves to reduce the system of equations (resulting in fewer unknowns, and fewer individual equations). Reduction is applied recursively, until no further substitutions are possible. Recursive substitution will ultimately terminate under one of three conditions:

[8] The maximum- and minimum-value operators can signal contradictions, as well. Consider the case, for example, in which values for a and c are known for the constraint $c = \max(a, b)$. If $a < c$, then it must be the case that $b = c$. If $a = c$, then the value of b is indeterminate. If $a > c$, though, a contradiction exists within the constraint network.

- The system of equations has been solved.
- Not all parameters have been computed, but no further reductions are possible. In this case, either the set of transformation rules is incomplete, or the original system of equations was underconstrained.
- All parameters have been computed, but some equations remain unused. In this case, either the original system of equations included some redundant equations, or, if the unused equations contradict any of the computed values, the original system of equations was overconstrained.

In the case where the original system was underconstrained, additional constraints or variable assignments are required in order to solve it completely. If the system was overconstrained, some constraints or variable assignments must be retracted in order to achieve consistency.

As a simple example of this process, consider the set of equations,

$$\begin{aligned} x - 2z &= 5y - 2y^2 \\ x &= 4w \\ w &= x - 3z \\ z &= 1 \end{aligned}$$

where the single variable-assignment, $z = 1$, is simply regarded as a fourth equation, in addition to the three mathematical constraints. To solve this system of equations, transformation rules are required for constants, multiplication, addition, and quadratic equations. Substituting in for the constant value of z, the system is reduced to

$$\begin{aligned} x - 2 &= 5y - 2y^2 \\ x &= 4w \\ w &= x - 3 \end{aligned}$$

Two different substitutions are possible at this point, since x and w appear isolated on the left-hand side of the second and third equations, respectively. Choosing to substitute for x yields the system,

$$\begin{aligned} 4w - 2 &= 5y - 2y^2 \\ x &= 4w \\ w &= 4w - 3 \end{aligned}$$

where the second equation has been retained intact, since it will be needed to compute x should the value of w become known. At this point, the third equation has been reduced to the point where it involves only a single unknown, w. Through a series of algebraic transformations,

$$\begin{aligned} w = 4w - 3 &\implies -3w = -3 \\ &\implies w = 1 \end{aligned}$$

it may be determined that the value of w is 1. Substituting in for this constant value, then, the system of equations is reduced to:

$$\begin{aligned} 4 - 2 &= 5y - 2y^2 \\ x &= 4 \times 1 \end{aligned}$$

Applying a few transformations concerning arithmetic operations on constant values, it is readily determined that the value of x is 4, and only a single equation remains:

$$2 = 5y - 2y^2$$

At this point, rules involving only arithmetic will be inadequate. In this case, the system must observe that, by rearranging terms slightly, the quadratic form,

$$2y^2 - 5y + 2 = 0$$

results. It is then a simple matter to apply the appropriate algebraic transformation for quadratic equations to isolate the final remaining unknown parameter, y. Once this parameter is isolated, a series of arithmetic transformations must be applied in order to compute its value:

$$\begin{aligned} 2y^2 - 5y + 2 = 0 &\implies y = \frac{5 \pm \sqrt{(-5)^2 - (4 \times 2 \times 2)}}{2 \times 2} \\ &\implies y = \frac{5 \pm \sqrt{25 - 16}}{4} \\ &\implies y = \frac{5 \pm \sqrt{9}}{4} \\ &\implies y = \frac{5 \pm 3}{4} \\ &\implies y = 2, \frac{1}{2} \end{aligned}$$

Note that the pattern-matcher which triggers transformation rules must be able to recognize each of the reducible expressions within these equations, and that rules must be available for each of the required transformations. On the other hand, as seen in this example, there are often alternative substitution paths which may be followed; thus, provision can be made for recovering from choices which result in dead-ends. This ability to examine alternate branches during the solution process also makes this approach amenable to parallel-processing implementations.

Unfortunately, though, it is difficult to maintain dependency information when employing this approach, because transformation is often accompanied by loss of information. As transformations are applied, the original constraints are destroyed; additionally, assignment of parameter values occurs as a side effect of the substitution process. Thus, there is no context from which dependencies may be inferred. Furthermore, because of this inability to track dependencies, retraction of assigned values generally requires re-solving the entire system of equations. Another difficulty with this approach is ensuring completeness and consistency. As suggested above, a large number of transformation rules is required to thoroughly support the algebraic transformation of systems of non-linear equations; straightforward substitution will fail in cases where multiple non-linear constraints are involved, suggesting the need for rules which pertain only to certain classes of non-linear systems. In addition, circularity within the rule-base must be avoided, to prevent the possible repeated application of sets of rules which perform complementary transformations.

Graph transformation and term rewriting do have certain advantages over local propagation, though, due primarily to their ability to examine the global interactions within systems of equations. The most significant of these advantages is the ability to solve systems of equations which contain cycles. From the perspective of these alternative approaches, local propagation may be viewed as applying only those transformations which concern the substitution of constant values for individual constraint parameters. By adding the ability to substitute symbolic expressions for constraint parameters, transformation systems are able to break down cycles among the constraints. Furthermore, by adding explicit rules for transforming complex expressions—such as the quadratic equation encountered in the example above—transformation systems are also able to solve certain non-linear equations for which local propagation is inadequate. Finally, in the case where the original system of equations is underconstrained, even though some parameters will remain unknown, the transformation process will yield a simplified set of equations, derived from the original system. This simplified system may be made available to the user to guide in the selection of additional parameter assignments, in order to completely constrain the system.

The earliest work on graph transformation for constraint management was done by Gosling at Carnegie-Mellon University [15]. Here, propagation of constraints was used in

support of a program for graphical layout. In this program, graph transformation is applied in those cases where local propagation is inadequate, and is employed to convert networks which cannot be solved using local propagation into networks which can be. A more recent application of the graph transformation approach is Serrano's MATHPAK system for mechanical design [31], in which constraint propagation is used to govern the shape, position, and dimensions of three-dimensional solids which are combined as elements of mechanical drawings. In using MATHPAK, the user builds up a drawing from a combination of three-dimensional primitives; parameters can be assigned to the dimensions of these primitives, and interactively linked via constraints. Given a set of initial values for some of the parameters, the MATHPAK program attempts to solve the entire constraint network via substitution methods. Constraints are represented as trees linking operators and parameters, and transformations are performed through direct substitution of sub-trees.

Term-rewriting relies on pattern-matching applied to actual constraint specifications (i.e., mathematical expressions), rather than to graph-based representations of constraint networks. Research by Leler at the University of North Carolina at Chapel Hill on this approach resulted in the BERTRAND programming language [23], which may be used both for the specification of constraint problems, and for the definition of transformation rules. Because the language for describing problems and the language for defining problem-solving rules are unified, BERTRAND enables the use of more complicated, aggregate data-structures—similar to the `record` programming construct in the Pascal programming language, `struct` in C, or `defstruct` in LISP—than are permitted in other constraint-propagation systems. In addition to simple variables, then, aggregate data structures may be constrained, thus simplifying problem representation. Furthermore, the transformation language is also capable of specifying rules which govern the transformations themselves. This permits the use of high-level constraint operations, such as conditionalization on boolean parameter-values.

Symbolic Algebra

Symbolic algebra is the application of computers to the solution of mathematical equations expressed in symbolic form. Whereas conventional programs are designed to perform computations on numerical data based on symbolic expressions, the function of symbolic algebra systems is to manipulate the expressions themselves, performing such operations as symbolic integration and differentiation, simplification, and determination of roots. As such, symbolic algebra has broad applicability; inversion of constraint specifications in support of constraint propagation is but one of its uses.

Graph transformation and term-rewriting represent a subset of the techniques available for performing symbolic algebra. The most complete symbolic algebra systems generally uti-

lize multiple schemes for guiding the transformation of mathematical expressions, including rule-based substitution and transformation methods, as well as more mundane approaches such as table-lookup.[9] Examples of symbolic algebra systems include Macsyma [4] and Mathematica [40].

Numerical Techniques

An alternative approach to performing constraint inversion is to apply numerical techniques. Given a general constraint specification of the form,

$$y = F(x_1, x_2, \ldots, x_i, \ldots)$$

it is a straightforward process to transform it into the normal form,

$$0 = F(x_1, x_2, x_3, \ldots, x_i, \ldots) - y = F_N(y, x_1, x_2, \ldots, x_i, \ldots)$$

In attempting to satisfy a constraint, then, the transformed constraint, F_N, may be viewed as an error term. Various numerical techniques, such as relaxation or the Newton-Raphson method, may be applied in order to minimize this error, thereby satisfying the constraint. The advantages of this approach are its generality—there are no limitations on constraint linearity, for example—and ease of implementation. Unfortunately, though, these numerical methods can suffer from stability problems, insofar as they require adequate initial conditions on which to base the numerical analysis.

Numerical techniques have been applied in several constraint propagation systems, including Sutherland's SketchPad program [36] for geometric design and the *Paper Airplane* program [11,20] for engineering preliminary design, as well as the original version of Borning's ThingLab program [5] for computer-aided simulation. ThingLab is particularly noteworthy for its integration of constraint-propagation into the SmallTalk environment for object-oriented programming, to support the interactive development and operation of computer-based simulations. In ThingLab, constraints may be used to govern the properties of object-classes defined for a particular simulation problem; the resulting model of the behavior being simulated is therefore non-directional, due to its underlying constraint-based description. The simulation, or perhaps just parts of it, can therefore be run "backwards", affording greater freedom to experiment with the system being modeled.

Hybrid Approaches

In practice, most constraint propagation systems rely on multiple techniques, in order to improve both efficiency and completeness. Indeed, in his presentation of the desired charac-

[9] For example, efficient algorithms for performing symbolic integration typically include explicit checks for well-known integrals, mimicking the approach most humans take in attempting to solve an integration problem (i.e., consulting the CRC Handbook).

teristics of a constraint-based system for supporting architectural design, Gross [17] points out the advantages gained by not relying on a single methodology, so that cooperating techniques can accommodate the one another's shortcomings. As mentioned above, Gosling's use of graph transformation was introduced to help treat these cases for which local propagation is inadequate. In this case, use of a second approach improves completeness. In the area of efficiency, exclusive reliance on symbolic algebra for performing constraint inversion can degrade program responsiveness, because there is a great deal of overhead inherent in such powerful techniques (e.g., there is a large body of possible transformations to be considered). For this reason, use of symbolic algebra is reserved for those cases in which other techniques—such as a less general, more domain-specific set of rules for graph transformation or term-rewriting—are insufficient.

Similarly, because numerical techniques have general applicability but are prone to stability problems, they are also employed in a number of systems as secondary methods. For example, the MARKSYMA program [30] for parametric design, developed at the Rensselaer Polytechnic Institute, employs term-rewriting for solving polynomial equations, resorting to numerical techniques for solving non-polynomial constraints and systems of simultaneous equations. The MATHPAK program, discussed above, also allows the use of numerical techniques when its implementation of graph transformation fails. Interestingly, even when numerical iteration is used to solve the constraints, MATHPAK still relies on the underlying graph-transformation algorithms to compute constraint derivatives—required when calculating, for example, the Jacobian matrix for a system of equations—analytically, via symbolic differentiation.

1.4.3 Constraint Propagation for Design

As indicated in the preceding sections, a number of constraint propagation systems have been implemented specifically for design applications. Paper Airplane and MARKSYMA allow the user to specify a set of parameters and equations which model a design problem, and, upon selection of a set of parameters to be used as input variables, apply constraint propagation to compute values for the remaining parameters. MATHPAK follows a similar approach, but also includes the ability to associate parameter values with geometric dimensions, allowing for graphical display of the designed artifact. Serrano's Concept Modeler (see Section 1.3.3), the successor to MATHPAK, provides a set primitive object-types, taken from the domain of mechanical engineering conceptual design, whose governing equations are specified as constraints.

Curiously, an additional feature which all of these constraint-based design tools share is that all constraint propagation is performed in unison (i.e., in "batch-mode" style), rather than interactively. That is, constraint propagation is delayed until a complete set of param-

eters, equations, and input values is provided, and then, upon final instruction by the user, the entire network of constraints is solved simultaneously. In contrast, other constraint propagation programs, such as Steele's, propagate individual constraints as soon as they become ready for propagation.

1.5 Overview of the Dissertation

Noting the potential advantages of these two programming techniques—object-oriented programming and constraint propagation—it has been the intent of this research effort to investigate the practicality and utility of their application to aiding in the solution of realistic problems in engineering conceptual design. Much work has therefore been devoted to the implementation of a computer program, dubbed *Rubber Airplane*, which utilizes these techniques in support of conceptual design.

This dissertation, then, is intended to document the development of this program, and its performance on a set of representative design problems. The current chapter has attempted to establish the motivation for this research, by identifying past and current trends in computer aids for engineering design, and suggesting improvements which can be derived from alternative approaches. Chapter 2 describes **component-modeling**, an application of object-oriented programming to the description of design problems in terms of components and their interactions. Component definitions are based-on class-like associations of component-specific attributes and the constraints which relate those attributes, in order to model the relevant physics and geometry. Interactions among components are specified by similar object-structures, referred to as **design links**, which are also defined in terms of attributes and constraints. Chapter 2 also introduces the notion of **design states**, which provide a means for automatically incorporating stepwise time-dependency on the values of designated component attributes.

Chapter 3 describes the implementation of constraint propagation employed in the *Rubber Airplane* computer program. Specifically, an augmented form of local propagation of known states is described, which applies a heuristically-derived assumption on the nature of engineering design constraints to efficiently detect cycles within the constraint network. Such cycles are then solved as simultaneous equations. Constraint inversion and solution of simultaneous equations is performed using numerical techniques. Furthermore, all constraint propagation, including detection and solution of cycles, is performed interactively.

Chapter 4 discusses the application of *Rubber Airplane* to three test cases. These representative design tasks focus on the configuration and sizing of three aerospace vehicles, a long-endurance, manned surveillance aircraft, a commercial transport aircraft, and a small-payload launch vehicle. Included is an account of the various strategies employed

in implementing the requisite analyses for these design problems, as well as observations regarding the benefits and shortcomings of the component-modeling approach in supporting the design analyses. Note that, in order to demonstrate the range of analyses which may be employed in using this approach, a complete LISP-based implementation of the vortex-lattice method for computing airfoil aerodynamics has been implemented for the two aircraft test cases, and integrated with the *Rubber Airplane* program.

Chapter 5 summarizes the results of this research, and compares its contributions with related efforts. Areas meriting further investigation are also suggested. Finally, for completeness, discussions of the program's support for design-parameter dimensions and units, interactive geometry display, and window-based user interface are presented in Appendices A, B, and C, respectively.

Chapter 2

Component Modeling

2.1 Motivation

2.1.1 The Function-Modeling Approach: *Paper Airplane*

Prior to commencing work on the current project, the author was involved in the development of the *Paper Airplane* program, mentioned in Section 1.4.2. *Paper Airplane* was a direct application of constraint propagation to engineering preliminary design: the program allowed the user to enter a set of variables and equations, referred to respectively as **design variables** and **design functions** and, upon fixing values for certain of the design variables, constraint propagation was employed to compute values for the remaining variables, by means of the available design functions.

Insofar as the interaction between the program and the user occurs at the level of individual functions and variables, then, *Paper Airplane* may be said to adhere to a **function-modeling** approach, in which design problems are described directly in terms of the mathematical equations which model them. The design programs ADAS [2], MARKSYMA [30], and MATHPAK [31], also discussed in Chapter 1, are similarly observed to employ the function-modeling approach.

Near the end of the *Paper Airplane* project, the author began investigating the possibility of adding a "long term memory" [11] to the program, for storing a library of commonly-used functions and variables. Soon after commencing this line of research, however, it was discovered that the resulting library, based upon the individual design functions and design variables to be used by the program, was unwieldy. Building up the description of a design problem from the elements of this database required attention to be focused at a rather fine level of detail, making the process time-consuming and awkward. To overcome these inconveniences, an alternative approach was sought.

2.1.2 Component-Modeling

The primary shortcoming of the function-modeling approach is its lack of sufficient organizational structure. It provides little means for unifying related information, beyond associating design functions with the design variables they govern. Thus, the paradigm presents no straightforward scheme for grouping interconnected design functions, or for relating design variables which are conceptually linked, though not *mathematically* related. For example, the design variables which depict the planform geometry of an airfoil—i.e., span, root and tip chords, aspect ratio, taper ratio, and sweep—are all needed to completely describe the planform, though not all are directly dependent upon one another. A group of three equations is required to specify the mathematical relationship among the first five of these parameters, and the final parameter, sweep, is completely independent of the others (at least, as far as defining the geometry is concerned). In building a library of design information, it is desirable to make the relationship among these design variables and design functions explicit; no clear means for doing so is afforded by the function-modeling approach.

Note, though, that the common feature of the functions and variables presented in this example is that they provide a partial description of a particular design component, specifically, an airfoil. Based on this observation, and inspired by the component-based interface used in GRADE (see Section 1.3.3), an alternative approach is suggested, based on **component-modeling**. Observing that engineered artifacts may often be readily described in terms of the physical components which comprise them, organizing design information based upon components, rather than upon the individual functions and variables which describe them, should result in a database which is easier to use. In solving a design task using this approach, the designer is able to model the problem by combining the appropriate physical objects required; in contrast, the function-modeling approach requires greater effort, since it focuses attention at a finer level of detail (i.e., the individual equations and parameters).

Furthermore, the component-modeling approach is able to take advantage of object-oriented programming techniques, such as inheritance and specialization. Such techniques provide significant opportunities for simplifying database maintenance and development, by allowing those persons who are responsible for collecting and systematizing design information to take advantage of the underlying component-based associations. When extending the database, for example, specialization and inheritance allow new component descriptions to be incrementally developed from existing component definitions. Insofar as inheritance encourages sharing of code, program maintenance is also facilitated: since parameters or calculations common to multiple components may be specified just once as part of a component superclass, making improvements or correcting errors in such code is simplified. Addition-

ally, because design parameters are directly associated with the physical components they describe, specification—and, therefore, display—of component geometry, as derived from these parameters, is straightforward.

In utilizing object-oriented programming techniques, then, components are represented by classes, which may be instantiated in order to build up an appropriate design representation. It is the actual instances which are manipulated and sized to solve a given design problem: the classes themselves are not altered: the use of instantiable classes to store static design knowledge results in a database which is not itself changed during the actual design process. Note that the principle of maintaining a database of domain knowledge (here, the library of classes which represent the available design components) which is independent of the algorithms which will process this data is not unique to this approach: on the contrary, the benefits of this technique have long been recognized in the domain of expert systems [6], where rule specification is independent of the inference algorithms which will be used to process these rules.

Finally, it should be noted that this approach is not necessarily original to this work. A number of the object-oriented design programs discussed in Section 1.3.3 of Chapter 1 employ a similar strategy. The predefined component-types available in Serrano's Concept Modeler [32], and the user-defined part/sub-part hierarchy utilized in the ICAD [29] and Wisdom Systems [24] programs, both represent alternative implementations of the component-modeling approach.

2.1.3 Intent

In investigating the utility of the component-modeling approach, it has been the goal of this research to develop a software tool which supports realistic engineering design based on an extensible database of instantiable **design components**. Thus, several basic requirements must be met, including support for:

- object-oriented description of design components, including relevant parameters, as well as component geometry;
- specification of the mathematical constraints which govern the integration of these components into a viable design solution;
- instantiation of component descriptions; and
- satisfaction of the governing equations, based upon user-supplied values for a subset of the component parameters.

The prototype design tool, *Rubber Airplane*, has satisfied these requirements, integrating them into a graphical, mouse-driven interface for instantiating design components, and

displaying their attributes and geometry.

During the process of developing this program, however, several deficiencies in the original approach were uncovered. First, there are often numerous design variables and design functions which are not directly associated with any one individual component. Examples from aircraft design include:

- parameters which describe the vehicle as a whole, such as gross weight, flight speed, and cruising range;
- parameters which do not describe physical characteristics of the design, such as atmospheric properties, and development and operating costs;
- constraints governing these non-physical parameters; and
- constraints governing multiple components, such as the attachment of one component to another, or the aerodynamic interference between two or more components.

To address this inadequacy, a second type of object, the **design link**, has been introduced. Design links do not represent the physical objects which comprise the design, but rather represent the interrelationships between those physical objects. Design links may specify their own parameters, thus providing for the definition of design variables which represent the gross properties or non-physical characteristics of a design. In addition, however, design links may also define constraints which govern not only their own parameters, but those of other components, as well. In this way, the mathematical constraints required to model the integration of multiple constraints may be specified.

While the problem of non-physical parameters and multi-component constraints was recognized and addressed fairly early in the project, a second shortcoming took longer to discover. This difficulty concerned the representation of time-dependent behavior, and became problematic while attempting to implement the mission-performance modules for the first test case (see Chapter 4). Mission-performance analysis, such as the calculation of overall fuel consumption, often depends strongly upon the gross properties of the vehicle, such as net weight, thrust, lift, and drag. However, these properties vary with time over the course of the mission profile. Thus, there is a specific value for vehicle weight associated with takeoff, another with cruise, and one or more with each of the various climb and descent phases. Initially, mission performance for the first test case was implemented by means of a design link with individual weight, lift, thrust, and drag parameters for each of the individual flight conditions. For this particular mission, there were nine such flight conditions. This solution was quickly judged unacceptable, however, because of its inflexibility. Such a comprehensive design link is too strongly tied to an individual design problem; a design

involving only five flight conditions would require development of another completely new, problem-specific mission-performance design link.

To overcome this inadequacy, a *third* object-type has been implemented, the so-called **design state**. Design states are provided for the purpose of representing the various phases of a vehicle's mission; thus, in the aircraft example mentioned above, design states representing the various flight conditions (e.g., takeoff, landing, cruise, etc.) would be available for instantiation. In conjunction with the addition of design states, component and link parameters may be optionally specified as **state-dependent** parameters, indicating that multiple copies of such parameters should be created, one for every design state present. Thus, parameters which have multiple values, dependent upon the flight condition (e.g., vehicle weight, thrust, lift, and drag, as suggested above), may be implemented as state-dependent parameters, and appropriate state-specific instances of these parameters—as well as the constraints which govern them—will be created automatically, as new design states are added. In this way, a flexible means of incorporating stepwise time-dependent phenomena into the design model is introduced.

In the remainder of this chapter, details of the implementation of this approach will be presented. Section 2.2 will present the implementation of design components, Section 2.3 will cover design links, and Section 2.4 will discuss design states. Note that, for convenience, the generic term "**design entity**" will be used to refer to design components, design links, and design states, collectively.

The prototype design tool, *Rubber Airplane*, has been implemented using the programming language LISP. A superset of Common Lisp [35], including the Flavors [39] extension for object-oriented programming has been used. The program runs on both Symbolics 3600 and Texas Instruments Explorer Lisp Machines. LISP macros are provided for defining component-, link-, and state-classes via a text editor; this data is represented internally by means of `defstruct` data structures, as are attribute and constraint specifications. Actual instances of design entities are represented as Flavor instances, as are the data structures representing the attributes and constraints of these design entities.

2.2 Component Representation

2.2.1 Overview of Design Components

As indicated above, component-modeling is introduced as a means of organizing related portions of design information. Thus, a component is to be defined in terms of the set of design variables and design functions which are used to describe it. As such, the component-modeling approach does not replace the function-modeling approach; rather, it augments it. Nevertheless, it is useful to introduce an alternative terminology. Thus, in the context of

component-modeling, design variables are referred to as **attributes** of the components with which they are associated. Similarly, the design functions associated with a component are referred to as its **constraints**.

In keeping with the object-oriented programming paradigm, design components are represented as objects. Thus, component objects, to be manipulated by the designer, will be instances of corresponding **component-classes**. To facilitate component-definition, multiple inheritance is supported, such that new component-classes may reference one or more component-class superclasses. Component subclasses inherit both the attributes and constraints of their superclasses, subject to specialization. Inheritance paths are based upon computation of a **class precedence list** for each class, which provides a complete and unambiguous ordering of its superclasses.

In this way, instantiation of a given component-class results in the creation of a new component object, with a set of attributes and constraints as specified by the component-class and its subclasses. Values of these attributes may be varied, triggering calculations based on the corresponding constraints. If component geometry has been specified, changes in attribute values also cause the geometry display to be updated appropriately.

2.2.2 Attribute Representation

Attribute Specification

Two basic types of component attributes are provided in *Rubber Airplane*, based on the values which may be assigned to them. Values of **scalar attributes** must be floating-point numbers. **Discrete attributes** take on specific values as selected from a corresponding **value-list**. Support for intermediate types of parameters, such as integers (whose values are numerical, though the set of possible values is not continuous), or composite parameters, such as vectors, matrices, and complex numbers, is not present, though, as discussed below, extension of the current system to incorporate such attribute-types is not infeasible. Nevertheless, the majority of component attributes may be represented as scalar attributes: since there are no restrictions on the computations performed by *Rubber Airplane* constraints, scalar attributes may be rounded to represent integral values, or combined as array-elements, as desired.

In defining scalar attributes, the numerical algorithms employed to perform constraint inversion require specification of an **order-of-magnitude** value, in order to guide selection of seed-values. In addition, suggested **low-value** and **high-value** bounds may be specified, to improve seed-value selection for these algorithms. When not explicitly declared, approximate bounds are calculate by scaling the (required) order-of-magnitude value. Note that these bounds are not strictly enforced, though the program does notify the user in the event that they are violated. Additionally, an initial value for the parameter may be specified; if

none is provided, the order-of-magnitude value is used.

To improve program utility, a units-conversion package has been incorporated into *Rubber Airplane*. As such, definition of scalar attributes also requires specification of the dimensions of each attribute, as well as declaration of the units in which the attribute's order-of-magnitude, low-value, and high-value are given. Compatibility between corresponding attribute dimensions and units is required.

As indicated above, discrete attributes are limited to values chosen from their value lists. Thus, the primary element of the definition of a discrete attribute is the specification of its value-list. Value lists may be specified as explicit lists of LISP data, or as symbols, whose LISP values are assumed to be lists. Value-lists represented by symbols are "recomputed" each time they are required, by evaluating the symbol.[1] This feature is provided in order to facilitate modification of the value-list in cases where the same value-list is referred to by multiple discrete attributes, or where the list must be constructed at run-time. Like scalar attributes, an initial value may be specified; otherwise, the initial value defaults to the first element of the attribute's value-list.

Discrete attributes are assumed to be dimensionless. Nevertheless, there are no restrictions on the contents of discrete attribute value-lists. When value-lists are comprised of data-structures with no standard printed representation, however, special accommodations must be made for displaying them, as well as for storing discrete-attribute values in data files between work sessions. Thus, discrete-attribute definitions may also specify **value-printer** and **value-saver** properties. The value-printer specification should be a function which, given a value from the corresponding attribute's value-list, returns a string representation of that value. The value-saver specification should be a function which, when similarly given a value from the value-list, returns a LISP form which evaluates to that value. These two properties thus provide for the presentation and retrieval of arbitrarily complex values for discrete attributes.

Finally, user-supplied documentation may be associated with either type of attribute specification. This option is provided to support annotation of attribute-dependent explanatory text, such as intended usage, or references to source material.

Of course, supporting only attribute specifications which represent floating-point scalar values or list-based discrete values can be restrictive. As mentioned above, it is possible to simulate additional data-types using the supported types, but this requires additional work on the part of the constraint programmer, since he must handle the details of the simulation himself, within the corresponding constraint definitions. It is certainly the case that these additional data-types are routinely encountered in engineering design. For example, integer

[1] It is perhaps more technically accurate to say that such value-lists are "dereferenced" each time they are accessed.

values are required when employing discretization to model complex continuous phenomena. Similarly, moments of inertia of the physical objects comprising a design are most readily represented via matrices.

Thus, in order to relieve the user of such burdens, it would be advantageous to provide direct support for all the types of data required. In the case of vectors, matrices, and complex numbers, it is primarily a matter of implementing techniques for composing scalar attributes into aggregate data structures. Means for accessing, modifying, and displaying such composite data-types would be required. Implementing integer-valued attributes would likely be more difficult, since they assume an infinite range of numerical values—similar to floating-point scalar attributes—as selected from a non-continuous set of possible values—much like discrete attributes. They thus share the representational simplicity of numerical data with the computation difficulties associated with discrete data (i.e., problems with constraint inversion).

Attribute Declaration

The *Rubber Airplane* program provides two means for defining the attributes to be associated with a design entity. The first is by means of the **defattribute** macro, to be describe here. For convenience, attribute declarations may also be incorporated into the class definitions of the design entities with which they are to be associated. For details on the latter approach, see Sections 2.2.5, 2.3.3, and 2.4.2.

Two forms of the **defattribute** macro are recognized, one for defining scalar attributes, and one for defining discrete attributes. The basic form for defining scalar attributes is:

```
(defattribute (entity-class attribute-name)
   :documentation documentation-string
   :comment comment-string
   :low-value suggested-low-value
   :order-of-magnitude suggested-order-of-magnitude
   :high-value suggested-high-value
   :value default-initial-value
   :dimensions dimensions-string
   :units units-string)
```

where the ordering of the keyword arguments—i.e., those preceded by an identifying keyword (a symbol preceded by a colon, such as "**:documentation**", "**:comment**", "**:value**", and "**:dimensions**")—is insignificant. Furthermore, all keyword arguments are optional, with the exception of the *suggested-order-of-magnitude*, *dimensions-string*, and *units-string* arguments. In fact, the *units-string* argument may also be omitted if the attribute is non-

dimensional, as indicated by specifying an empty string for the attribute's *dimensions-string*. The syntax for specifying dimensions and units is described in Appendix A.

The *entity-class* argument is required to identify the component-, link-, or state-class with which the attribute is to be associated, and is specified as a LISP symbol. The *attribute-name* may be specified as a string or as a symbol. A string may be used to control capitalization when displaying the attribute's name, but must not contain any characters not normally allowed in symbols (e.g., spaces, tabs, carriage returns), since attributes are referenced as symbols within constraints. In addition, for reasons to be presented below (see Section 2.3.3), the character "@" may not be used in the name of an attribute, regardless of whether it is specified as a symbol or as a string. Note that the *Rubber Airplane* program performs explicit checks on the validity of attribute-names to ensure compliance with these restrictions. The remaining elements of the attribute definition correspond directly to the scalar-attribute properties discussed in the preceding section, with the exception of the *comment-string* argument. The *comment-string* argument is provided for annotation of the code itself rather than the attribute; as indicated above, the *documentation-string* argument is provided for supplying commentary to be passed on to attribute instances.

The format for defining discrete attributes with the **defattribute** macro is

```
(defattribute (entity-class attribute-name)
   :documentation documentation-string
   :comment comment-string
   :value-list value-list-specification
   :value default-initial-value
   :value-printer value-printer-function
   :value-saver value-saver-function)
```

Again, all keyword arguments are optional, with the exception of the *value-list-specification*. The meaning and syntax of the *entity-class* and *attribute-name* arguments are the same as for scalar-attributes. Similarly, the remaining arguments serve to delineate the properties of discrete attributes discussed in the preceding section, except for the *comment-string* argument, which serves the same purpose as it does when defining scalar attributes. Note that, if not specified, the *value-printer-function* and *value-saver-function* arguments effectively default to the **identity** function.[2]

Finally, note that because the two forms of the **defattribute** macro have different syntactic requirements—specifically, each scalar attribute definition requires specification of the *suggested-order-of-magnitude*, *dimensions-string*, and *units-string* arguments, while discrete

[2]The LISP **identity** function is a function of one argument, which performs no computation, but merely returns its argument, unchanged. It is thus functionally equivalent to the LISP expression, `(lambda (x) x)`.

attribute definitions require specification of the *value-list-specification* argument—discerning the user's intent when employing this macro is a straightforward process. Ambiguous cases which satisfy all or part of both sets of requirements signal an error.

Example Attribute Definitions

As an example of the definition of a scalar attribute, consider the following **defattribute** specification for the **span** attribute of an **airfoil** component-class for representing lifting surfaces:

```
(defattribute (airfoil span)
   :documentation "Span of an airfoil planform."
   :comment "Is this a good value for order-of-magnitude?"
   :low-value 10 :order-of-magnitude 25 :high-value 100
   :value 30 :dimensions "1" :units "ft")
```

As indicated above, the *comment-string* argument is ignored. This attribute is designated as having the dimensions of length, by means of the "1" specification for the *dimensions-string* argument. The *units-string*, "ft", indicates that the specifications for the *suggested-low-value*, *suggested-order-of-magnitude*, *suggested-high-value*, and *default-initial-value* are all given in units of feet. (For further details regarding the specification of dimensions and units, see Appendix A.) As an example of a non-dimensional scalar attribute, the following specification for the **aspect-ratio** attribute of the same **airfoil** component-class is presented:

```
(defattribute (airfoil "Aspect-ratio")
   :documentation "Measure of planform two-dimensionality."
   :low-value 1 :order-of-magnitude 10 :high-value 30
   :dimensions "")
```

Note the absence of a *units-string* specification. Additionally, the *default-initial-value* specification is omitted, indicating that the *suggested-order-of-magnitude*, 10, should be used.

Finally, as an example of the definition of a discrete attribute, here is one possible specification for a **flaps-deployed?** attribute for the **airfoil** component-class is presented:

```
(defattribute (airfoil flaps-deployed?)
   :documentation "Flag indicates deployment status of airfoil flaps."
   :value-list '(t nil)
   :value-printer yes-no-string
   :value nil)
```

```
(defun yes-no-printer (boolean)
   "Returns an appropriate string, based on the value of BOOLEAN."
   (if boolean "Yes" "No"))
```

The *value-list-specification* for this attribute is simply a list of LISP symbols representing the logical values for *true* and *false*, `t` and `nil`. The *default-initial-value* is `nil`, indicating that, by default, the flaps are not deployed. Note that in addition to the **defattribute** specification, definition of an auxiliary LISP function, **yes-no-string**, is included. This function serves as the *value-printer-function* for the **flaps-deployed?** attribute; as its documentation string indicates, this auxiliary function, defined using standard LISP syntax, simply returns the appropriate string, "Yes" or "No", corresponding to the logical value of its argument. Note that since the *value-list-specification* for this attribute consists of standard symbols, no specification of the *value-saver-function* is required. Indeed, the *value-printer-function* argument is specified here merely for aesthetic purposes, and is also not strictly required.

2.2.3 Constraint Representation

Constraint Specification

Both equality and inequality constraints may be associated with design entities in *Rubber Airplane*. The **body** of a constraint is the sequence of calculations which implements the mathematical relationship to be represented by the constraint. Specifically, the body of a constraint specifies the series of operations to be performed based on the values of a set of **input parameters** in order to compute values which constrain the **output parameters**. Note, however, that the adjectives "input" and "output" refer only to the single, unidirectional form of the constraint designated by its body; the use of constraint propagation means that an alternative form of the constraint may ultimately be employed to constrain an input parameter, using previously-assigned values for the output parameters. The exact form of the constraint to be applied depends upon the user's choice of base variables.

In the case of constraints associated with component-classes, input and output parameters must refer to attributes already defined for the component-class. In specifying an attribute to serve as an input or output parameter, the required units must also be specified. (The syntax for specifying units is described in Appendix A). Appropriate conversions are automatically incorporated into the constraint, as needed. Finally, as was the case with attribute specifications, documentation may be associated with a constraint specification.

Constraint Declaration

The **defconstraint** macro is provided for defining *Rubber Airplane* constraints. The basic format of a **defconstraint** statement is:

(defconstraint (*entity-class constraint-name constraint-type*)
(*(output1 output-units1) (output2 output-units2) ...*)
(*(input1 input-units1 local-name1) (input2 input-units2 local-name2) ...*)
documentation-string
constraint-body)

The *entity-class* argument identifies the component-, link-, or state-class with which the constraint is to be associated, and is specified as a LISP symbol. The *constraint-name* may be specified as either a symbol or a string. Note that, unlike attribute-names (see Section 2.2.2), there are no restrictions on the contents of constraint-names specified as strings.

The *constraint-type* argument indicates the type of the constraint; two types of equality constraints and four types of inequality constraints are supported. Thus, the *constraint-type* argument must be one of six recognized keywords, **:equality**, **:one-way**, **:greater-than**, **:less-than**, **:greater-than-zero**, and **:less-than-zero**. A *constraint-type* of **:equality** is used to designate equality constraints which may be inverted during constraint propagation. Equality constraints which are not to be inverted are specified by declaring the value **:one-way** for the *constraint-type* argument.[3] If the constraint is an inequality constraint specifying a lower bound on the output parameter,[4] based upon some function of the input parameters, the *constraint-type* should be **:greater-than**. Similarly, if an upper bound is to be computed, the *constraint-type* should be **:less-than**. For convenience, the *constraint-type* argument may be specified as either **:greater-than-zero** or **:less-than-zero** if the desired bound is simply the constant value, zero. Finally, note that the default value for the *constraint-type* argument is **:equality**; if no *constraint-type* is explicitly declared, it is assumed that the constraint is an equality constraint, and that it may be inverted.

Thus, the first argument to the **defconstraint** macro is a list which identifies the constraint, and indicates its type. The second argument is a list of the output-variables for the computation indicated in the *constraint-body*. Each element of this list is itself a list, the first element of which is the name of the attribute to be computed, specified as a LISP symbol, and the second element of which is a string indicating the units in which the computed value will be returned (see Appendix A for details). The program verifies the compatibility of the indicated units with the declared dimensions of the corresponding

[3] As discussed in Chapter 3, all constraints involving discrete attributes, as well as those having multiple output parameters, must be declared **:one-way** constraints.

[4] Inequality constraints may have only one output parameter.

attribute. Note that only constraints with a *constraint-type* of **:one-way** may specify multiple outputs. Other constraints may specify only a single output-variable; in such cases, the outer set of parentheses may be omitted.

The third argument to the **defconstraint** macro is a list of the input-variables for the computation embodied in the *constraint-body*. As with the output-variables, each element of this list is itself a list. These sublists are comprised of either two or three elements:

- The first element of the sublist should be a symbol naming the attribute which is to serve as an input-variable for the constraint.
- The second element of the sublist should be a string (see Appendix A) indicating the units in which the value of this attribute is to be supplied to the *constraint-body* when applying the constraint. Compatibility of the indicated units with the declared dimensions of the attribute is verified by the *Rubber Airplane* program.
- The third element of the sublist, which is optional, should be a symbol which will serve as an alias for the attribute's name within the *constraint-body*.

There are no restrictions on the number of input-variables which may be specified for a constraint. As with output parameters, though, if only a single input parameter is to be specified, the outer set of parentheses may be omitted.

The *documentation-string* argument allows user-specified commentary to be associated with a constraint definition. The *constraint-body* argument describes the body of the constraint, as defined above. It is specified as a series of LISP expressions representing the desired sequence of operations for computing the values of the output-variables based upon the values of the input-variables. Input-variables are referred to within the *constraint-body* by their names, expressed as symbols, or by the appropriate aliases, when declared (see above). The value or values returned by the final expression in the *constraint-body* should correspond to the computed values for the output-variables; thus, a form which returns multiple values, such as the LISP **values** function, is required in the case of multiple-output constraints. Finally, note that no specification of input variables or the *constraint-body* argument is required for constraints whose *constraint-type* is either **:greater-than-zero** or **:less-than-zero**.

Example Constraint Definitions

The first example constraint definition implements the mathematical relationship which defines the **aspectratio** attribute introduced in Section 2.2.2 in conjunction with a hypothetical **airfoil** component-class:

```
(defconstraint (airfoil "Definition of Aspect Ratio" :equality)
      (aspect-ratio "")
      ((span "m") (wing-area "m2" S))
   "Implements the definition of aspect ratio as an equality constraint."
   (/ (* span span) S)))
```

The constraint is named by the string "Definition of Aspect Ratio". As defined, the output parameter for this constraint is the non-dimensional **aspect-ratio** attribute. The input parameters are the **span** and **wing-area** attributes, whose values are expected to be in metric units when the constraint is applied. The value for **aspect-ratio** is actually computed by dividing the square of the **span** by the **wing-area**, at least when the constraint is to be applied as defined. Note also the use of the local alias, "S", for the **wing-area** attribute within the body of the constraint. The remaining component of this constraint definition, the string which appears between the input parameter specifications and the *constraint-body*, is the constraint definition's *documentation-string* argument.

As an example of a constraint with multiple output parameters, consider the following **:one-way** constraint, which purports to calculate the aerodynamic coefficients for the **airfoil** component-class, based on its planform dimensions.

```
(defconstraint (airfoil aerodynamic-coefficients :one-way)
      ((lift-slope "deg -1") (drag-slope "deg -1") (l-over-d ""))
      ((aspect-ratio "") (span "ft") (sweep "deg")
       (root-chord "ft") (tip-chord "ft"))
   "Computes the lift and drag coefficients, by calling
    MAGIC-AERODYNAMICS-FUNCTION."
   (multiple-value-bind (cl-alpha cd-alpha)
      (magic-aerodynamics-function
            aspect-ratio sweep root-chord tip-chord span)
      (values cl-alpha cd-alpha (/ cl-alpha cd-alpha))))
```

Here, the hypothetical **magic-aerodynamics-function** is expected to return two values, the slopes of the lift and drag curves, based on the planform dimensions, expressed in English units. The constraint itself returns multiple values, the two slopes (in units of inverse degrees) and their quotient, corresponding to the three output parameters, **lift-slope**, **drag-slope**, and **l-over-d**.

The following code implements a simple inequality constraint, governing the sign of the **aspect-ratio** attribute:

```
(defconstraint (airfoil "Sign of Aspect Ratio" :greater-than-zero)
```

```
    (aspect-ratio "")
  "Ensures correct sign for the aspect-ratio attribute.")
```

Since the *constraint-type* argument has a value of **:greater-than-zero**, no input-parameters or *constraint-body* are required. A slightly more complicated inequality constraint, which simply compares the values of the value of **l-over-d** attribute to some user-specified minimum, **minimum-l-over-d**, might be defined as follows:

```
(defconstraint (airfoil "Minimum L/D" :greater-than)
      (l-over-d "") (minimum-l-over-d "")
  "Enforces minimum bound on the lift-to-drag ratio."
  minimum-l-over-d)
```

Note that the *constraint-body* consists merely of the name of the input parameter, "**minimum-l-over-d**". Also note that since only one input parameter is specified, the outer set of parentheses—which would otherwise be required to delimit the input parameters—has been omitted. (The same is true of the outer set of parentheses for the output parameters, which have been omitted for all of these examples, except for the **aerodynamic-coefficients** constraint.) Finally, as an example of an inequality constraint which performs a computation, the following alternative implementation of the previous example is presented:

```
(defconstraint (airfoil "Minimum Lift" :greater-than)
      (lift "N") ((drag "N") (minimum-l-over-d ""))
  "Enforces a minimum bound on the airfoil's lift,
  based on MINIMUM-L-OVER-D."
  (* drag minimum-l-over-d))
```

Rather than directly bounding the **l-over-d** attribute, this (admittedly contrived) constraint employs the **minimum-l-over-d** attribute to establish an inequality relationship between the airfoil's **lift** and **drag** attributes.

2.2.4 Implementation of Inheritance

Issues in Supporting Multiple Inheritance

As discussed above, object-oriented programming techniques are employed to simplify definition of component-, link-, and state-classes, by facilitating incremental development and promoting database modularity. The key to these advantages is inheritance among classes, specifically, inheritance of both attributes and constraints by design-entity subclasses. Additionally, inherited attributes and constraints may be overridden, when desired, via specialization: an attribute or constraint of a superclass which has the same name as one defined

locally by the subclass will *not* be inherited. Local specifications are preferred over inherited specifications. Furthermore, specializations, unless they are themselves overridden, are also inherited.

In the case where several superclasses of a given class specialize the same attribute or constraint, though, which particular specialization is to be inherited? It is the role of the class precedence list, introduced in Section 2.2, to make this determination. The class precedence list is a ranking of all the superclasses—direct or indirect—associated with a class; in addition, the class precedence list for a class includes the class itself, at the top of this ranking. The exact set of attribute and constraint specifications to be associated with a class may therefore be completely determined by traversing its class precedence list. First, the specifications local to the class itself are incorporated. Next, those of the second class in the precedence list are incorporated, except for those specifications which share the same name as a specification which has already been incorporated. This procedure is applied, in turn, to each class in the precedence list, adding specifications only for those attribute- and constraint-names not already present.

Single inheritance allows at most one superclass to be declared for each new class. Construction of the class precedence list for such cases is straightforward. The first member of the class precedence list is, of course, the class itself. The second member is its superclass. The third is the superclass's superclass, and so on until a base class (i.e., one with no superclasses of its own) is encountered.

An appropriate algorithm for constructing the class precedence list in the case of multiple inheritance is not so obvious. When multiple superclasses are declared as the direct superclasses of a new class, some means is required for ordering the superclasses' superclasses. This task is made more difficult by the fact that a given class's superclasses may themselves share superclasses. The ordering of direct superclasses must be preserved; additionally, inheritance relationships among the various direct and indirect superclasses must be maintained.

Rather than attempt to devise a new algorithm for computing class precedence lists under multiple inheritance, it was decided to take advantage of the collective efforts of various members of the computer science community by instead adopting a published algorithm. Specifically, the algorithm developed for the Common Lisp Object System (CLOS) [3], recently accepted by the American National Standards Institute (ANSI) as part of the Common Lisp language specification, has been implemented.

Computation of the Class Precedence List

This algorithm begins with the **local precedence order**, which is derived directly from the definition of a class. The class definition is required to specify an ordered list of direct

superclasses. Furthermore, each class is assumed to implicitly include the top-level class—denoted (in typical LISP fashion) by the symbol "t"—as the final member of its list of direct superclasses. The class t thus serves as a sort of universal superclass. For each class, then, the local precedence order is determined simply by adding the class itself to the beginning of this ordered list. Thus, if a class C specifies as its direct superclasses the ordered list $(C_1, C_2, C_3, \ldots, C_n)$, class C's local precedence order is the ordered list $(C, C_1, C_2, C_3, \ldots, C_n, \mathtt{t})$. The goal of the algorithm is to determine a total ordering on C and all of its superclasses which remains consistent with the local precedence order of each of those classes.

Based on the local precedence order of class C, a set of **local precedence pairs** is generated, in which each pair denotes the precedence of one class over its immediate successor. Continuing the notation introduced above, which is itself based on the notation of Reference [3], this set may be written as

$$R_C = \{(C, C_1), (C_1, C_2), (C_2, C_3), \ldots, (C_{n-1}, C_n), (C_n, \mathtt{t})\}$$

To compute the class precedence list for a class and all of its superclasses, the union of these sets of local precedence pairs is required. Denoting S_C as the set of class C and all of its superclasses, direct and indirect, the union of all local precedence pairs, designated R, is

$$R = \bigcup_{c \in S_C} R_C$$

In cases where the individual sets of local precedence pairs are consistent, the union R will provide a partial ordering on S_C; a topological sorting of R will generate the class precedence list for S. If the individual sets are not consistent, however, topological sorting will fail, due to the presence of circularity among the superclasses, and R is itself said to be inconsistent.

Topological sorting is performed by examining the precedence pairs, in order to identify a class c in S_C such that no other class precedes c according to the pairs present in R. The first such class will be C, the class being defined. When such a class c is found, such that no pairs of the form (b, c) are present in R, all other pairs referring to c, which will of course be of the form (c, d), are removed from R. The class c is also removed from S_C, and added to the end of the class precedence list currently under construction. This process is repeated, until no such classes c—i.e., classes with no predecessors—are found.

If the process has stopped and R is empty, then the sorting has been completed successfully. Only the top-level class t will remain in S_C, and, upon adding class t to the end of the class precedence list, construction of the class precedence list is concluded. If this is not the case, however, then the set R is inconsistent, since each class remaining in S_C, with the exception of t, has a predecessor among those remaining classes. Circularity is present,

insofar as there must be a chain of classes $c_1, \ldots, c_n$, such that c_i precedes c_{i+1}, and c_n precedes c_1. The precedence pairs in R include a loop, and no consistent class precedence list may be derived.

Finally, note that it may sometimes be the case that there are multiple classes remaining in S_C which have no predecessors according to the partial ordering specified by R. Since it is desired that the process of computing the class precedence list be deterministic and therefore predictable, a selection criterion is required. Thus, given a choice among several classes with no predecessors, the class which has a direct subclass latest in the class precedence list, as it has been computed so far, is chosen. Since, for each class, there is an ordering among its direct superclasses, it will be the case that only one such class among the possible candidates will be found. If there is *no* such candidate class, then it is again the case that R is inconsistent; it does not specify a complete partial ordering.

To be more specific, let $N_1, \ldots, N_m$, $m \geq 2$, be the classes in S_C which have no predecessors in R, and $(C_1, \ldots, C_n)$, $n \geq 1$, be the class precedence list constructed so far. Class C_1 will be the most specific class (i.e., C itself, where C is the class for which the class precedence list is being computed), and C_n is the least specific. Selection among the N_i classes is therefore determined by finding the largest value for j, where $1 \leq j \leq n$, such that there exists a k, where $1 \leq k \leq m$, for which N_k is a direct superclass of C_j. Class N_k is the class to be added next to the class precedence list for C. As indicated in Reference [3],

> The effect of this rule for selecting from a set of classes with no predecessors is that the classes in a simple superclass chain are adjacent in the class precedence list and that classes in each relatively separated subgraph are adjacent in the class precedence list. For example, let T_1 and T_2 be subgraphs whose only element in common is the class J. Suppose that no superclass of J appears in either T_1 or T_2. Let C_1 be the bottom of T_1; and let C_2 be the bottom of T_2. Suppose C is a class whose direct superclasses are C_1 and C_2 in that order, then the class precedence list for C will start with C and will be followed by all classes in T_1 except J. All the classes of T_2 will be next. The class J and its superclasses will appear last.

In using this algorithm, then, a "clustering" effect is observed. All superclasses of the first direct superclass precede those of the second direct superclass, except those which are shared by both. The same is true of the second and third direct superclasses. Shared superclasses appear after the unshared superclasses of the last direct superclass which inherits them.

It is observed, then, that the CLOS algorithm addresses both of the concerns mentioned above. Ordering of direct superclasses is preserved, as are the inheritance relationships among direct superclasses and their own superclasses. Additionally, means for detecting

Class	Superclasses	Local Precedence Pairs
component	*none*	{(component, t)}
distribution	*none*	{(distribution, t)}
planform	component	{(planform, component), (component, t)}
thickness	distribution	{(thickness, distribution), (distribution, t)}
camber	distribution	{(camber, distribution), (distribution, t)}
wing	planform camber thickness	{(wing, planform), (planform, camber), (camber, thickness), (thickness, t)}
fin	thickness planform	{(fin, thickness), (thickness, planform), (planform, t)}
tail	wing fin	{(tail, wing), (wing, fin), (fin, t)}

Table 2.1: Classes and superclasses for class precedence list examples.

circular inheritance paths—in the form of inconsistencies within the partial ordering of the inherited classes—are provided.

Examples

To illustrate the procedure described in the preceding section, consider the set of classes and superclasses presented in Table 2.1. To compute the class precedence list for component-class **wing**, the sets S and R must first be constructed, where, as indicated above, S is the set consisting of class **wing** and all of its superclasses, and R is the union of the sets of local precedence pairs for all of the sets in S. Applying the information in Table 2.1, then, it is seen that for class **wing**,

$$\begin{aligned} S &= \{\texttt{wing}, \texttt{planform}, \texttt{camber}, \texttt{thickness}, \texttt{component}, \texttt{distribution}, \texttt{t}\} \\ R &= \{(\texttt{wing}, \texttt{planform}), (\texttt{planform}, \texttt{camber}), (\texttt{camber}, \texttt{thickness}), \\ &\quad (\texttt{thickness}, \texttt{t}), (\texttt{planform}, \texttt{component}), (\texttt{component}, \texttt{t}), \\ &\quad (\texttt{camber}, \texttt{distribution}), (\texttt{thickness}, \texttt{distribution}), (\texttt{distribution}, \texttt{t})\} \end{aligned}$$

The class **wing** has no precedents in R, and is therefore chosen first. This class is then removed from S, as are all pairs which include it in R. The sets are then reduced to

$$\begin{aligned} S &= \{\texttt{planform}, \texttt{camber}, \texttt{thickness}, \texttt{component}, \texttt{distribution}, \texttt{t}\} \\ R &= \{(\texttt{planform}, \texttt{camber}), (\texttt{camber}, \texttt{thickness}), (\texttt{thickness}, \texttt{t}), \\ &\quad (\texttt{planform}, \texttt{component}), (\texttt{component}, \texttt{t}), (\texttt{camber}, \texttt{distribution}), \\ &\quad (\texttt{thickness}, \texttt{distribution}), (\texttt{distribution}, \texttt{t})\} \end{aligned}$$

and the resulting precedence list, so far, is **(wing)**. Class **planform** is the next class to be chosen according to the algorithm, yielding the result **(wing planform)**, and the sets are reduced to

$$\begin{aligned} S &= \{\texttt{camber}, \texttt{thickness}, \texttt{component}, \texttt{distribution}, \texttt{t}\} \\ R &= \{(\texttt{camber}, \texttt{thickness}), (\texttt{thickness}, \texttt{t}), (\texttt{component}, \texttt{t}), \\ &\qquad (\texttt{camber}, \texttt{distribution}), (\texttt{thickness}, \texttt{distribution}), (\texttt{distribution}, \texttt{t})\} \end{aligned}$$

At this stage, there are two classes—**camber** and **component**—which have no precedents in R, so the one which has a direct subclass closest to the end of the current class precedence list should be added next. The only direct subclass of class **camber** which is already in the class precedence list is **wing**. Class **planform** is a direct subclass of class **component**, and since **planform** follows **wing** in the class precedence list, **component** is to be added next.

The current result, therefore, is **(wing planform component)**. Removing class component and its pairs from S and R, respectively, results in

$$\begin{aligned} S &= \{\texttt{camber}, \texttt{thickness}, \texttt{distribution}, \texttt{t}\} \\ R &= \{(\texttt{camber}, \texttt{thickness}), (\texttt{thickness}, \texttt{t}), \\ &\qquad (\texttt{camber}, \texttt{distribution}), (\texttt{thickness}, \texttt{distribution}), (\texttt{distribution}, \texttt{t})\} \end{aligned}$$

Therefore, class **camber** is added next, yielding **(wing planform component camber)**, and sets S and R are reduced to

$$\begin{aligned} S &= \{\texttt{thickness}, \texttt{distribution}, \texttt{t}\} \\ R &= \{(\texttt{thickness}, \texttt{t}), (\texttt{thickness}, \texttt{distribution}), (\texttt{distribution}, \texttt{t})\} \end{aligned}$$

Class **thickness** is added next, yielding

$$\begin{aligned} S &= \{\texttt{distribution}, \texttt{t}\} \\ R &= \{(\texttt{distribution}, \texttt{t})\} \end{aligned}$$

and a class precedence list of **(wing planform component camber thickness)**. The final selection from R is, therefore class **distribution**, giving the result **(wing planform component camber thickness distribution)**. The sets S and R have now been reduced to

$$\begin{aligned} S &= \{\texttt{t}\} \\ R &= \{\} \end{aligned}$$

Class S has only one remaining element, the universal superclass **t**. Class R has been reduced to the empty set. According to the algorithm, then, computation of the class

precedence list has been successful, and the only remaining step is the addition of class t to the class precedence list for **wing**, yielding the final result (**wing planform component camber thickness distribution t**).

As indicated above, however, it is certainly possible to specify a set of superclasses which cannot be ordered. Indeed, the superclasses specified for class **tail** in Table 2.1 are inconsistent, since the superclass **wing** specifies that class **planform** should precede class **thickness**, while superclass **fin** specifies the reverse ordering. Classes **wing** and **fin** are not themselves inconsistent, but their combination, as required by class **tail**, is inconsistent. No class precedence list can be constructed for class **tail**; attempting to do so results in an error.[5]

2.2.5 Definition of Component-Classes

Component Definition Syntax

As discussed above, components are defined as classes in terms of the attributes and constraints with which they are associated. Additionally, component-classes may specify one or more superclasses, whose attributes and constraints are to be inherited, subject to specialization. The actual mechanism for defining new component-classes in *Rubber Airplane* is the **defcomponent** macro. The syntax for this macro is as follows:

(**defcomponent** *entity-class*
attribute-specifications
superclasses
option
option
...)

The *entity-class* argument should be a LISP symbol which names the component-class. This symbol is also used when employing the **defattribute** and **defconstraint** macros to associate attributes and constraints with the component-class (see Sections 2.2.2 and 2.2.3). The *superclasses* argument should be a list of symbols, each of which identifies a component-class which is to serve as a direct superclass of the new component-class. As indicated in the Section 2.2.4, the ordering of classes within the list of superclasses *is* significant.

The *attribute-specifications* argument should be a list of attribute specifications, similar in syntax to attribute definitions declared via the **defattribute** macro. As mentioned in Section 2.2.2, this argument provides an alternative means for associating attributes with

[5]Of course, one simple means for reconciling this inconsistency is to simply change the ordering of the superclasses for either **wing** or **fin**. This change would validate the use of these two classes as mutual superclasses of class **tail**.

component-classes. It is provided for the purpose of syntactic similarity with other LISP extensions for object-oriented programming, such as the **defflavor** and **defclass** macros of Flavors [39] and CLOS [3], respectively. Each attribute specification is itself a list, of the form

(attribute-name keyword keyword-arg keyword keyword-arg ...)

where *attribute-name* is a symbol or string which names the attribute, as in **defattribute**, and the alternating keywords and keyword-arguments are identical to those recognized by **defattribute** (see Section 2.2.2). Both scalar and discrete attributes may be defined in this manner.

The final arguments to the **defcomponent** macro is a set of options which allow for further description of the component-class. A listing of the supported **defcomponent** options appears in Table 2.2. Note that it is not necessary to specify any options. To take advantage of the mouse-based user-interface for component-class instantiation, however, it is necessary to at least declare a library category for the class, via the **:category** option. Furthermore, use of the **:required-attributes** option allows a component-class to define constraints which refer to attributes not directly associated with the class; it is assumed that the corresponding attributes, with the specified dimensions, will be defined by some other class in any class precedence list which includes this component-class.

The Base Component-Class

A final element to the definition of *Rubber Airplane* component-classes is the presence of a base component-class, called "**design-component**". *Rubber Airplane* requires all instantiable component-classes to include this class in their class precedence lists; it is thus similar in nature to CLOS's universal superclass, "**t**" (see Section 2.2.4). Class **design-component** need not be a direct superclass of each component-class, but each instantiable class (i.e., those for which the **:abstract-component** option is not specified) must include it as either a direct or indirect superclass.

The role of the **design-component** class is to provide standard attributes for those properties which are common to all design components: reference position, center of gravity, mass, and weight. The code which defines this component-class, including supporting constraints, appears in Figures 2.1–2.2. Note that **design-component** is an abstract component-class, and therefore it cannot be directly instantiated.

In addition to **design-component**, another built-in component-class, **design-component-attachment**, is also provided. This class is intended to provide a simple means for relating the position of two component-instances, by declaring one of the two to be the **superior** of the other, which is itself referred as to the **inferior** component. The definition

Option	Description
(:documentation *string*)	Specifies *string* as the documentation-string for the component-class.
(:comment *string*)	Specifies *string* as an ignored comment. Provided for annotation of the **defcomponent** code itself.
(:category *cat1 cat2 ... catn*)	Declares a hierarchical path of library categories for locating the component-class. Category *cat1* will be a top-level category, and *cat2* through *catn* will be successive sub-categories. Categories not already present are automatically created.
:abstract-component	Declares that the component-class is an abstract class, which may not be instantiated, but may serve as a superclass for another component-class.
(:required-components *comp1 comp2 ...*)	Declares *comp1*, *comp2*, ... as component-classes which must be present as (indirect) superclasses of the component-class, upon instantiation. The arguments *comp1*, *comp2*, ... should be symbols naming defined component-classes. Only valid in conjunction with the :abstract-component option.
(:required-attributes (*att1 dims1*) (*att2 dims2*) ...)	Declares *att1*, *att2*, ... as attributes, with corresponding dimensions *dims1*, *dims2*, ..., which must be present as (inherited) attributes of the component-class, upon instantiation. Arguments *att1*, *att2*, ... should be symbols or strings, and arguments *dims1*, *dims2*, ... should be dimension-strings. Only valid in conjunction with the :abstract-component option.
:no-attachment	Declares that instances of the component-class should not have a superior component associated with them. (See Section 2.2.5 for a discussion of component superiors.)

Table 2.2: Supported options for the **defcomponent** macro.

```
(defcomponent design-component
     (("Position-X"
       :documentation
       "Absolute X-coordinate for the component's reference position."
       :low-value -100 :order-of-magnitude 10 :high-value 100 :value 0
       :dimensions "1" :units "m")
      ("Position-Y"
       :documentation
       "Absolute Y-coordinate for the component's reference position."
       :low-value -100 :order-of-magnitude 10 :high-value 100 :value 0
       :dimensions "1" :units "m")
      ("Position-Z"
       :documentation
       "Absolute Z-coordinate for the component's reference position."
       :low-value -100 :order-of-magnitude 10 :high-value 100 :value 0
       :dimensions "1" :units "m"))
     ()
  :abstract-component
  (:documentation "Defines position attributes for design components."))

(defattribute (design-component "Mass")
  :documentation "Mass of the component."
  :low-value 0.01 :order-of-magnitude 100 :high-value 10000 :value 1
  :dimensions "m" :units "kg")
(defattribute (design-component "Weight")
  :documentation "Weight of the component."
  :low-value 0.01 :order-of-magnitude 100 :high-value 10000 :value 10
  :dimensions "f" :units "lbf")

(defconstraint (design-component "Gravity") (weight "N") ((mass "kg"))
  "Computes the weight of a component at sea level on Earth."
  (let ((g 9.807))
    (* g mass)))
```

Figure 2.1: LISP definition for the base component-class, **design-component**.

```
(defattribute (design-component "CG-X")
  :documentation
  "Absolute X-coordinate for the component's center-of-gravity position."
  :low-value -100 :order-of-magnitude 10 :high-value 100 :value 0
  :dimensions "1" :units "m")
(defattribute (design-component "CG-Y")
  :documentation
  "Absolute Y-coordinate for the component's center-of-gravity position."
  :low-value -100 :order-of-magnitude 10 :high-value 100 :value 0
  :dimensions "1" :units "m")
(defattribute (design-component "CG-Z")
  :documentation
  "Absolute Z-coordinate for the component's center-of-gravity position."
  :low-value -100 :order-of-magnitude 10 :high-value 100 :value 0
  :dimensions "1" :units "m")

(defattribute (design-component "Moment-X")
  :documentation "X-component of the moment arm of the component's mass."
  :low-value -1000000 :order-of-magnitude 10 :high-value 1000000
  :dimensions "m 1" :units "kg m")
(defattribute (design-component "Moment-Y")
  :documentation "Y-component of the moment arm of the component's mass."
  :low-value -1000000 :order-of-magnitude 10 :high-value 1000000
  :dimensions "m 1" :units "kg m")
(defattribute (design-component "Moment-Z")
  :documentation "Z-component of the moment arm of the component's mass."
  :low-value -1000000 :order-of-magnitude 10 :high-value 1000000
  :dimensions "m 1" :units "kg m")

(defconstraint (design-component "X Moment") (moment-x "kg m")
          ((mass "kg") (cg-x "m"))
  "Computes the x-component of the moment arm about the origin."
  (* mass cg-x))
(defconstraint (design-component "y Moment") (moment-y "kg m")
          ((mass "kg") (cg-y "m"))
  "Computes the y-component of the moment arm about the origin."
  (* mass cg-y))
(defconstraint (design-component "z Moment") (moment-z "kg m")
          ((mass "kg") (cg-z "m"))
  "Computes the z-component of the moment arm about the origin."
  (* mass cg-z))
```

Figure 2.2: Additional attribute and constraint definitions for the base component-class, design-component.

```
(defcomponent design-component-attachment
  () ()
  :abstract-component
  (:required-attributes (position-x "1") (position-x@superior "1")
                        (position-y "1") (position-y@superior "1")
                        (position-z "1") (position-z@superior "1"))
  (:documentation
        "Provides simple attachment constraints for design components."))

(defattribute (design-component-attachment "Attach-X")
  :documentation "X position of object relative to its superior."
  :low-value -100 :order-of-magnitude 10 :high-value 100 :value 0
  :dimensions "1" :units "m")
(defattribute (design-component-attachment "Attach-Y")
  :documentation "Y position of object relative to its superior."
  :low-value -100 :order-of-magnitude 10 :high-value 100 :value 0
  :dimensions "1" :units "m")
(defattribute (design-component-attachment "Attach-Z")
  :documentation "Z position of object relative to its superior."
  :low-value -100 :order-of-magnitude 10 :high-value 100 :value 0
  :dimensions "1" :units "m")

(defconstraint (design-component-attachment attachment-x) (position-x "m")
          ((attach-x "m") (position-x@superior "m"))
  "Relates the component's Position-X to the Position-X of its superior."
  (+ attach-x position-x@superior))
(defconstraint (design-component-attachment attachment-y) (position-y "m")
          ((attach-y "m") (position-y@superior "m"))
  "Relates the component's Position-Y to the Position-Y of its superior."
  (+ attach-y position-y@superior))
(defconstraint (design-component-attachment attachment-z) (position-z "m")
          ((attach-z "m") (position-z@superior "m"))
  "Relates the component's Position-Z to the Position-Z of its superior."
  (+ attach-z position-z@superior))
```

Figure 2.3: LISP definition for class `design-component-attachment`, a second built-in *Rubber Airplane* component-class, which provides a simple means for attaching one component to another.

of this component-class is presented in Figure 2.3. Whenever such a declaration is made, the attributes and constraints of class **design-component-attachment** are automatically added to the inferior component. In addition, aliases for the **position-x**, **position-y**, and **position-z** attributes of the superior component are also added to the inferior, where they are named **position-x@superior**, **position-y@superior**, and **position-z@superior**, respectively. As indicated in Figure 2.3, these aliases are referred to by the **attachment-x**, **attachment-y**, and **attachment-z** constraints of class **design-component-attachment**, and are listed as **:required-attributes** for the class. The *Rubber Airplane* program supports the interactive declaration and retraction of superior and inferior component-instances.

Example Component Definitions

Two example component definitions have already been presented: those for *Rubber Airplane*'s two built-in component-classes, **design-component** and **design-component-attachment** (see Figures 2.1– 2.3). As an example of a user-defined component-class, consider the following definition of an abstract component class, **planform-mixin**:

```
(defcomponent planform-mixin
      ((span
          :documentation "Span of an airfoil planform."
          :comment "Is this a good value for order-of-magnitude?"
          :low-value 10 :order-of-magnitude 25 :high-value 100
          :value 30 :dimensions "l" :units "ft")
       (wing-area
          :documentation "Two-dimensional wing surface area."
          :low-value 100 :order-of-magnitude 750 :high-value 2000
          :dimensions "l2" :units "ft2")
       ("Aspect-ratio"
          :documentation "Indication of airfoil two-dimensionality."
          :low-value 1 :order-of-magnitude 10 :high-value 30
          :dimensions "")
       ("Taper-ratio"
          :documentation "Ratio of tip-chord to root-chord."
          :low-value 0 :order-of-magnitude 1 :high-value 1
          :dimensions ""))
      ()
   :abstract-component
   (:comment "Need to add root-chord and tip-chord attributes.")
```

```
(:category aerodynamics lifting-surfaces mixins)
(:documentation
   "Provides attributes for describing the planform of an airfoil."))
```

This abstract class[6] is intended to provide its subclasses with the attributes required to represent the planform geometry of an airfoil. It is not itself instantiable, since it provides only a partial description of airfoil components. Note also that, for completeness, various constraints should be added to this component-class, such as the following:

```
(defconstraint (planform-mixin "Definition of Aspect Ratio" :equality)
      (aspect-ratio "")
      ((span "m") (wing-area "m2"))
   "Implements the definition of aspect ratio as an equality constraint."
   (/ (* span span) wing-area)))

(defconstraint (planform-mixin "Definition of Taper Ratio" :equality)
      (taper-ratio "")
      ((tip-chord "m") (root-chord "m"))
   "Implements the definition of taper ratio as an equality constraint."
   (/ tip-chord root-chord)))
```

Thus, not only does the abstract component-class **planform-mixin** provide the required attributes, it also implements the appropriate mathematical relationships between them.

If we assume that an abstract component-class for describing airfoil cross-sections, class **cross-section-mixin**, is also available, then it is possible to combine these superclasses into an instantiable **airfoil** class, as follows:

```
(defcomponent airfoil
      ()
      (cross-section-mixin planform-mixin design-component)
   (:category aerodynamics lifting-surfaces)
   (:documentation
      "Combines planform and cross-section descriptions
       to provide a complete airfoil description."))
```

[6]Note that the use of the term "mixin" in the name of this class follows the tradition of Flavors [39], where it is conventional to refer to an uninstantiable class which implements a single, particular feature as a "mixin", since it is only useful when mixed together with other classes. Thus, since **planform-mixin** describes a specific, general aspect of airfoils, its name has been chosen to reflect its intended role in the class hierarchy.

This component-class need not provide any attributes or constraints of its own, since it inherits the necessary specifications from its superclasses: **airfoil-section-mixin** provides the cross-section representation, **planform-mixin** implements the planform data, and **design-component**—whose presence here is required because **airfoil** is not an abstract class, and none of its other superclasses inherit class **design-component**—defines attributes and constraints for specifying the position and mass-properties of the component-class.

One might consider that the use of the abstract superclasses **cross-section-mixin** and **planform-mixin** in this example is inefficient: why not combine their attributes and constraints directly into the **airfoil** component-class and do away with the abstract classes altogether, since *all* airfoils must include planform and cross-section information, anyway? Actually, there is an advantage to separating the two aspects of airfoil design into individual, abstract component-classes. This advantage follows from the object-oriented approach adopted in *Rubber Airplane*, and manifests itself in the form of code reusability

It is because airfoil planform and cross-section geometry are independent of one another that the separation is possible: there are no geometrical constraints which depend upon both descriptions. Furthermore, while the set of parameters required to describe the planform is basically fixed, multiple cross-section representations are possible. For instance, the shape of the cross section could be constant along the span. Alternatively, it might vary continuously between one fixed cross-section at the root of the airfoil, and a second fixed cross-section at the tip. In the first case, attributes and constraints depicting but a single cross-section are required. In the second case, two sets of attributes and constraints are required. In *Rubber Airplane* these two approaches might be implemented by providing two different component-classes, **single-cross-section-mixin** and **dual-cross-section-mixin**. Combining either of these two abstract classes with the **planform-mixin** and **design-component** component-classes would result in a valid **airfoil** component-class. In fact, two different airfoil component-classes, say **single-section-airfoil** and **dual-section-airfoil** could co-exist in the component-class hierarchy. As indicated above, because planform and cross-section geometry are independent, no changes to class **planform-mixin** are needed to make it compatible with one or the other cross-section descriptions. The definition of component-class **planform-mixin** can therefore be shared by both airfoil classes. If planform representation had been included in the definition of the airfoil component-classes, it would be necessary to implement it twice.

2.3 Design Link Representation

2.3.1 Overview of Design Links

As mentioned in Section 2.1.3, there are several cases in which it is desirable to specify design parameters and constraints which are not associated with any one particular component. Vehicle gross properties, flight conditions, and various non-physical characteristics (e.g., development costs) are examples of such "component-less" attributes. Similarly, design constraints on such parameters, as well as constraints which simultaneously relate the attributes of multiple components, have no obvious association with a single component-class. Forcing such attributes and constraints to be associated with a specific component-class would not facilitate program modularity, since it would introduce unwanted dependencies between component-classes. For this reason, a second type of design entity, the design link, is introduced. Class definitions for design links may specify their own attributes and constraints, in the same manner as component-classes. In addition, however, design links can also specify a set of design entities, the attributes of which may be referenced by the design link's constraints. These design entities are referred to as the **linkages** of the design link.

In addition to specifying constraints which access the attributes of a link-class's linkages, link-classes may also define **collector constraints**, which declare an attribute of the link to be either the sum or product of a collection of attributes obtained by applying a **collection predicate** to *all* of the attributes of *all* of a design's components, links, and states, not just those specified as linkages. All attributes which satisfy the collection predicate are included in the collector constraint. Collector constraints therefore provide a means for implementing summation, Π-product, minimization, and maximization constraints of the forms

$$\begin{aligned} y &= \textstyle\sum_{i=1}^{n} x_i &&= x_1 + x_2 + \ldots + x_n \\ y &= \textstyle\prod_{i=1}^{n} x_i &&= x_1 \cdot x_2 \cdot \ldots \cdot x_n \\ y &= \min_{i=1}^{n} x_i &&= \min(x_1, x_2, \ldots, x_n) \\ y &= \max_{i=1}^{n} x_i &&= \max(x_1, x_2, \ldots, x_n) \end{aligned}$$

Collector constraints provide a flexible means for introducing constraints which are independent of the types of objects present in a design. Knowledge of the types of each component in an aircraft design, for example, is not needed in order to compute the vehicle's gross weight: a collector constraint which sums of all the component's weight attributes provides a design-independent method for performing the required calculation.

2.3.2 Linkages

The `:required-attributes` option to the `defcomponent` macro, as describe in Section 2.2.5, provides one means for defining constraints which reference attributes not directly defined by a component-class. This feature basically permits component-classes to define constraints

on inherited attributes. Design links provide an alternate means for defining constraints which refer to attributes not directly provided by their owners: linkages.

The definition of each link-class includes specification of a set of linkages. Linkage specifications are used to identify those design entities which may be accessed as linkages by actual instances of the link-class. Each linkage specification is a list of the form

(linkage-name predicate-keyword predicate-argument predicate-argument ...)

The first element of the specification, the *linkage-name*, should be a LISP symbol; this symbol may be used in the link-class's constraints to access the attributes of the design entity which will serve as a linkage. For reasons discussed below, this symbol may not contain the character, "@". The second element should be a keyword, which names one of the recognized linkage predicates. A listing of the linkage predicates provided by *Rubber Airplane* appears in Tables 2.3–2.4. The remaining elements of the specification, the *predicate-argument* elements, serve as the arguments to the predicate chosen as the *predicate-keyword* argument. A description of the arguments appropriate to each recognized predicate is given in Tables 2.3–2.4.

When the link-class is instantiated, all component-, link-, and state-class instances present in the current design are examined, to see if they satisfy any of the linkage specifications. The user is then presented with a menu giving the name of each linkage specification and a list of those design entities which satisfy the corresponding predicate; one design entity is then selected for each of the linkage specifications to serve as the corresponding linkage for the new link-class instance.

As examples of valid linkage specifications, consider the following:

```
(wing :class airfoil)
(bottom-component :attributes height position-z)
(complicated-example :and (:class design-component)
                          (:not (:class airfoil))
                          (:or (:attributes lift)
                               (:attributes induced-drag))
```

In the first example, a linkage named "**wing**" is specified, which should be an instance of component-class **airfoil**. In the second case, which might be appropriate for a design link intended to stack one or more components, the **bottom-component** specification will match any design entity which has an attribute named "**height**" and an attribute named "**position-z**". The final example—which illustrates the use of the boolean combinatorial predicates, **:and**, **:or**, and **:not**—will select any design component (i.e., instances of class **design-component**) which is not of class **airfoil**, and has either a **lift** attribute or an **induced-drag** attribute (or both).

Predicate	Description
(:class *class-name*)	Selects design entities which are instances of class *class-name* or one of its subclasses. The argument *class-name* should name a defined component-, link-, or state-class.
(:superior-class *class-name*)	Selects component-instances whose superiors (i.e., the component to which they are attached, see Section 2.2.5) are instances of class *class-name* or one of its subclasses. The argument *class-name* should name a defined component-class.
(:attributes (*att dims*) (*att dims*) ...)	Selects design entities with attributes whose names and dimensions match all those specified by the *att* and *dims* arguments to the predicate. One or more attribute/dimensions-pair arguments are required, for which the *att* specification should be a LISP symbol corresponding to the desired attribute name, and the *dims* specification should be a string indicating the required dimensions, using the format described in Appendix A.
(:and *pred pred* ...)	Selects design entities which satisfy all of the linkage predicates specified by the *pred* arguments. The predicates are applied in the order in which they appear in the specification; predicate application continues until one of the predicates proves false, or all are applied successfully. Each *pred* argument should itself be a list of the form (*predicate-keyword* *predicate-argument predicate-argument* ...) where *predicate-keyword* is a keyword naming a linkage predicate, and the *predicate-argument* arguments supply the appropriate arguments for the linkage predicate.

Table 2.3: Linkage predicates provided for defining link-classes.

Predicate	Description
(:or *pred pred* ...)	Selects design entities which satisfy at least one of the linkage predicates specified by the *pred* arguments. The predicates are applied in the order in which they appear in the specification; predicate application continues until one of the predicates proves true, or all are determined to be false. The format of the *pred* arguments is the same as for the :and linkage predicate.
(:not *pred*)	Selects design entities which do not satisfy the linkage predicate specified by the *pred* argument. The format of the *pred* argument is the same as for the argument of the :and linkage predicate.
(:predicate *LISP-pred*)	Selects design entities which satisfy the LISP predicate indicated by the *LISP-pred* argument. The *LISP-pred* argument should be a symbol or lambda-expression identifying a LISP function of one argument, which should accept a *Rubber Airplane* design entity as its argument, and return a boolean value indicating predicate satisfaction.

Table 2.4: Linkage predicates provided for defining link-classes (continued).

Finally, note that the **:predicate** linkage predicate is provided so that selectors which cannot be defined using the other linkage specifications may nevertheless be specified by giving the link-class implementor access to the underlying LISP software which supports the *Rubber Airplane* program. Use of the **:predicate** linkage predicate typically requires some knowledge of the internal programming details of *Rubber Airplane*, however; it is therefore not intended for casual use. In fact, in implementing the test cases discussed in Chapter 4, there has not been any need to resort to its use. So, while it has been made available—for completeness's sake, if nothing else—its use has not been justified by practice.

2.3.3 Definition of Link-Classes

Link Definition Syntax

Like components, design links are defined as classes, based on their attributes, constraints, and superclasses. Link-class definitions are specified using the **deflink** macro, the syntax of which is

```
(deflink entity-class
        linkage-specifications
        attribute-specifications
        superclasses
    option
    option
    ...)
```

Note that the arguments to the **deflink** macro are very similar to those of **defcomponent**. As with **defcomponent**, the *entity-class* argument should be a LISP symbol which will serve as the name of the link-class. This name may be used in conjunction with the **defattribute** and **defconstraint** macros to associate attributes and constraints with the link-class (see Sections 2.2.2 and 2.2.3). The *superclasses* argument should be a list of symbols identifying the link-classes which are to serve as direct superclasses of the new *entity-class* link-class. The construction of class precedence lists for link-classes follows the same algorithm as for component-classes (see Section 2.2.4); thus, as with **defcomponent**, the ordering of classes within the list of superclasses is significant.

As with **defcomponent**, a number of options may be specified as the final arguments to the **deflink** macro. A listing of the recognized options is presented in Table 2.5; note that all supported **deflink** options provide analogous features to the corresponding **defcomponent** options. As with **defcomponent**, specification of **deflink** options is not required, though specification of the **:category** option is required for interactive instantiation of link-classes using the *Rubber Airplane* user-interface.

Option	Description
(:documentation *string*)	Specifies *string* as the documentation-string for the link-class.
(:comment *string*)	Specifies *string* as an ignored comment. Provided for annotation of the **deflink** code itself.
(:category *cat1 cat2 ... catn*)	Declares a hierarchical path of library categories for locating the link-class. Category *cat1* will be a top-level category, and *cat2* through *catn* will be successive sub-categories. Categories not already present are automatically created.
:abstract-link	Declares that the link-class is an abstract class, which may not be instantiated, but may serve as a superclass for another link-class.
(:required-links *link1 link2 ...*)	Declares *link1*, *link2*, ... as link-classes which must be present as (indirect) superclasses of the link-class, upon instantiation. The arguments *link1*, *link2*, ... should be symbols naming defined link-classes. Only valid in conjunction with the :abstract-link option.
(:required-attributes (*att1 dims1*) (*att2 dims2*) ...)	Declares *att1*, *att2*, ... as attributes, with corresponding dimensions *dims1*, *dims2*, ..., which must be present as (inherited) attributes of the link-class, upon instantiation. Arguments *att1*, *att2*, ... should be symbols or strings, and arguments *dims1*, *dims2*, ... should be dimension-strings. Only valid in conjunction with the :abstract-link option.

Table 2.5: Supported options for the **deflink** macro.

```
(deflink design-link ()
        () ()
  :abstract-link
  (:documentation
      "Basic link type.  Has no attributes or constraints of its own."))
```

Figure 2.4: LISP definition for the base link-class, **design-link**.

There is, of course, no **defcomponent** analog to the remaining argument to **deflink**, the *linkage-specifications* argument. This argument should be a list, each element of which is a linkage specification, as described in the preceding section, Section 2.3.2. As indicated there, these linkage specifications will be used to identify the design entities to which the design link may be applied when the link-class is instantiated. In addition, note that all linkage specifications are, by necessity, inherited. Furthermore, specifying a linkage with the same name as an inherited linkage does not override the inherited specification; rather, all linkage specifications provided by the link-classes in the class precedence list which share the same name are combined using the **:and** linkage predicate. In this way, none of the requirements of any of the superclasses are lost, but specialization still remains possible.

A primary purpose of design links is the implementation of constraints which simultaneously reference the attributes of multiple design entities. As indicated above, the constraints of a link-class may access the attributes of its linkages by means of the linkage-names which appear as the first element of each linkage specification. Specifically, linkage attributes are accessed by appending the linkage-name to the end of the attribute's name, inserting an "@" character between them to serve as a delimiter. (This is the reason why, as indicated above, and in Section 2.2.2, names of both link-class linkages and design-entity attributes may not themselves contain this character.) Thus, a link-class with a linkage named "**forward-wing**" could reference the corresponding wing instance's **span** attribute as either an input or output parameter of one of its constraints using the name "**span@forward-wing**".

The Base Link-Class

As with components, there is a base link-class, class **design-link**, which must be present as a superclass of all instantiable link-classes. Unlike the **design-component** component-class, **design-link** provides no attributes or constraints of its own; its use is required simply to introduce a standard means of identifying design links within the class hierarchy. For instance, class **design-link** may be used as an argument to the **:class** linkage predicate (see Section 2.3.2), or as an argument to the **:owner-class** collection predicate (see below, Section 2.3.4). The actual **deflink** definition for the **design-link** link-class is presented

in Figure 2.4.

Example Link Definitions

As an example of a link-class definition, consider the following possible definition for an abstract link which supports the modeling of the aerodynamic interference between two airfoils:

```
(deflink airfoil-interference-mixin
      ((forward-airfoil :class airfoil)
       (aft-airfoil :class airfoil))
      ((separation-x
          :documentation "Lengthwise separation of the airfoils."
          :low-value 10 :order-of-magnitude 25 :high-value 100
          :dimensions "l" :units "ft")
      (separation-z
          :documentation "Heightwise separation of the airfoils."
          :low-value -5 :order-of-magnitude 5 :high-value 25
          :dimensions "l" :units "ft")
      (downwash
          :documentation "Downwash on the aft airfoil."
          :order-of-magnitude 5 :dimensions "A" :units "deg"))
      ()
   :abstract-link
   (:category aerodynamics interference)
   (:documentation
      "Provides basic attributes for computing interference effects
       between two (horizontally-oriented) airfoils."))
```

In addition to specifying three basic attributes, two linkages are specified, one for each of the two airfoil components which, logically enough, are both required to be instances of the **airfoil** component-class. Based on these linkages, the following constraints can be defined for computing the **separation-x** and **separation-z** attributes of link-class **airfoil-interference-mixin**:

```
(defconstraint (airfoil-interference-mixin "Lengthwise Separation")
       (separation-x "m")
       ((position-x@forward-airfoil "m") (position-x@aft-airfoil "m"))
   "Computes the lengthwise separation between the two airfoils."
   (- position-x@forward-airfoil position-x@aft-airfoil))
```

```
(defconstraint (airfoil-interference-mixin "Heightwise Separation")
      (separation-z "m")
      ((position-z@forward-airfoil "m") (position-z@aft-airfoil "m"))
   "Computes the heightwise separation between the two airfoils."
   (- position-z@forward-airfoil position-z@aft-airfoil))
```

Note the use of the "@" character to reference the attributes of linkages, as described above. In these two examples, the positions of the two airfoils are accessed in order to compute the x- and z-components of the distance vector which separates the two airfoils.

To extend this example, consider the use of this abstract class as a superclass of an instantiable aerodynamic interference link-class:

```
(deflink canard-wing-interference
      ((forward-airfoil :class canard)
       (aft-airfoil :class wing))
      ()
      (airfoil-interference-mixin design-link)
   (:category aerodynamics interference)
   (:documentation
      "Provides basic attributes for computing interference effects
       between two (horizontally-oriented) airfoils.")
   (:comment "Need canard-specific constraint on downwash."))
```

Here, the linkages inherited from **airfoil-interference-mixin** have been specialized, such that the **forward-airfoil** linkage is required to be an instance of component-class **canard**, and the **aft-airfoil** linkage is required to be an instance of component-class **wing**. As indicated in Section 2.3.3, however, these new linkage specifications do not supersede the inherited specifications; rather, all linkage specifications with a common name are merged via the :and linkage predicate. Thus, the effective linkage specifications for link-class **canard-wing-interference** are

```
((forward-airfoil :and (:class canard) (:class airfoil))
 (aft-airfoil :and (:class wing) (:class airfoil)))
```

Since component-classes **canard** and **wing** are likely to be subclasses of component-class **airfoil**, the merging of linkage specifications is not strictly necessary in this case. More complicated cases (e.g., when combining more diverse link-classes than those employed in this example), however, necessitate the merging of linkage specifications in order to ensure satisfaction of all linkage requirements of the link-classes present in the class precedence list.

2.3.4 Collector Constraints

Overview of Collector Constraints

Collector constraints provide design links with a second means for constraining attributes owned by other design entities, independent of their linkages. Specifically, collector constraints may be used to implement summations or products of related attributes; they may also be used to find the minimum or maximum value associated with a collection of attributes. In this way, collector constraints are completely independent of the specific components, links, and states present in a given design, but they still have access to the attributes of these design entities.

Attributes to be related via a collector constraint are selected by means of **collection predicates**. Collection predicates are similar to linkage predicates, but whereas linkage predicates are used to identify suitable design entities, collection predicates are used to select attributes. Given a collector constraint and its collection predicate, all attributes a_i which satisfy the predicate are grouped together as the constraint's summation, Π-product, minimum-value, or maximum-value terms. The collector constraint must also specify an output parameter A; thus, the resulting constraint will assume one of four possible forms:

$$A = \sum_i a_i$$

$$A = \prod_i a_i$$

$$A = \min_i a_i$$

$$A = \max_i a_i$$

Only scalar attributes may serve as input and output parameters of collector constraints. Note that, for dimensional reasons, all attributes to be summed or examined for a minimum or maximum value must have the same dimensions as the constraint's output variable. Additionally, since the number of terms to be combined by a collector constraint is indeterminate, it is further required that both the output parameter (A) and all input parameters (a_i) for a Π-product collector constraint be dimensionless: this is the only way to ensure the dimensional validity of such constraints.

Finally, it should be mentioned that summation and product collector constraints, once instantiated, are treated as normal equality constraints by the *Rubber Airplane* program. Thus, they are subject to inversion during the course of constraint propagation: if a value is made available for the collector constraint's output parameter by some other means, the collector constraint may be used to compute a value for one of its input parameters. Due to the difficulties associated with inverting the operators used to find least and greatest values (see Footnote fn:min-max on page 27), minimum-value and maximum-value collector

constraints are treated as uninvertible.

Collector Constraint Specification

The **defcollector** macro is provided for defining collector constraints. The basic format for **defcollector** is

(defcollector (*link-class collector-name collector-type***)**
output-parameter
collection-predicate
*documentation-string***)**

The *link-class* argument should be a symbol identifying the link-class with which the collector constraint is to be associated. The *collector-name* argument should be a string or symbol which names the constraint. As with other constraints, collector constraints are subject to inheritance and specialization; inherited collector constraints are superseded by the presence of a constraint with the same name in association with a class which appears higher in the inheriting link-class's class precedence list. The third argument, *collector-type*, must be one of four keyword symbols, either **:sum**, **:product**, **:min**, or **:max**; the *collector-type* serves to indicate whether a summation, Π-product, minimum-value, or maximum-value collector constraint is desired.

The *output-parameter* argument should be a symbol naming an attribute which is to serve as the left-hand-side of the summation or product relationship represented by the collector constraint. This *output-parameter* attribute may be an attribute defined by the collector constraint's own link-class, or an attribute inherited by the link class; it may also be an attribute of one of the link-class's linkages, accessed using the "@" syntax introduced above. The *documentation* argument should be a string describing the intended role of the collector constraint, and is optional.

The remaining argument to **defcollector** is the *collection-predicate* argument. As indicated above, the collection predicate is used to identify those attributes which are to be grouped together for summation or multiplication. The form in which collection predicates are specified is as follows:

(predicate-keyword predicate-argument predicate-argument ...)

The first element of the specification, the *predicate-keyword*, should be a keyword, which names one of the recognized linkage predicates. A listing of the linkage predicates defined for *Rubber Airplane* appears in Tables 2.6–2.7. As can be seen in this table, many of the predicate keywords available for defining collection predicates are very similar in nature to those provided for defining linkage specifications (see Section 2.3.2). One unique addition

Predicate	Description
(:named *att-name*)	Selects attributes whose names are the same as *att-name*. The *att-name* argument should be either a string or a symbol.
(:name-contains *att-name-substring*)	Selects attributes whose names contain the sequence of characters specified by *att-name-substring* as a substring. The *att-name-substring* argument should be either a string or a symbol.
(:owner-class *class-name*)	Selects attributes whose owners are design entities which are instances of class *class-name* or one of its subclasses. The argument *class-name* should name a defined component-, link-, or state-class.
(:owner-superior-class *class-name*)	Selects attributes of component-instances whose superiors (i.e., the component to which they are attached, see Section 2.2.5) are instances of class *class-name* or one of its subclasses. The argument *class-name* should name a defined component-class.
(:owner-inferior-of *linkage-name*)	Selects attributes whose owners are inferior components to the component represented by the *linkage-name* linkage associated with the link-class instance for which the collector constraint has been defined.
(:query)	Selects attributes based upon direct querying of the program user. This collection predicate causes the user to be prompted for final determination of whether or not a given attribute meets the collection criteria.

Table 2.6: Collection predicates provided for defining collector constraints.

is the **:owner-inferior-of** predicate keyword, which allows collection to be based on the inferiors of one of a link-class's linkages. The remaining elements of the specification represent the arguments appropriate to the predicate which appears as the *predicate-keyword* argument. Tables 2.6 and 2.7 include descriptions of the appropriate arguments for each recognized predicate keyword. Note that all string comparisons are performed using the Common Lisp function **string-equal**, which is *not* case-sensitive; note also that symbols are coerced into strings prior to name comparison.

Finally, note that four "synonyms" for **defcollector** macro are also provided: **defsum**, **defproduct**, **defmin**, and **defmax**. The syntax for these macros is

Predicate	Description
(:and *pred pred ...*)	Selects attributes which satisfy all of the collection predicates specified by the *pred* arguments. The predicates are applied in the order in which they appear in the specification; predicate application continues until one of the predicates proves false, or all are applied successfully. Each *pred* argument should itself be a complete and valid collection predicate specification.
(:or *pred pred ...*)	Selects attributes which satisfy at least one of the collection predicates specified by the *pred* arguments. The predicates are applied in the order in which they appear in the specification; predicate application continues until one of the predicates proves true, or all are determined to be false. Each *pred* argument should itself be a complete and valid collection predicate specification.
(:not *pred*)	Selects attributes which *do not* satisfy the collection predicate specified by the *pred* argument. The *pred* argument should itself be a complete and valid collection predicate specification.
(:predicate *LISP-pred*)	Selects attributes which satisfy the LISP predicate indicated by the *LISP-pred* argument. The *LISP-pred* argument should be a symbol or **lambda**-expression identifying a LISP function of one argument, which should accept a *Rubber Airplane* attribute-instance as its argument, and return a boolean value indicating predicate satisfaction.

Table 2.7: Collection predicates provided for defining collector constraints (continued).

```
(defsum (link-class collector-name)
        output-parameter
        collection-predicate
    documentation-string)

(defproduct (link-class collector-name)
        output-parameter
        collection-predicate
    documentation-string)

(defmin (link-class collector-name)
        output-parameter
        collection-predicate
    documentation-string)

(defmax (link-class collector-name)
        output-parameter
        collection-predicate
    documentation-string)
```

where the various arguments to **defsum**, **defproduct**, **defmin**, and **defmax** play the same role as the corresponding arguments to **defcollector**. In fact, the only difference between these macros and the **defcollector** macro is the absence of the *collector-type* argument; for **defsum**, it is assumed to be **:sum**, for **defproduct** it is assumed to be **:product**, and so on. These additional macros simply provide a shorthand notation for defining the various types of collector constraints.

Collector Constraint Examples

Before presenting an example collector constraint, first consider the following link-class definition:

```
(deflink aircraft-gross-properties
      ()
      ((overall-height
          :documentation "Overall height of the vehicle."
          :order-of-magnitude 10 :dimensions "l" :units "m")
       (gross-weight
          :documentation "Net vehicle weight."
```

```
      :order-of-magnitude 100000 :dimensions "f" :units "lbf")
   (net-lift
      :documentation "Net vehicle lift."
      :order-of-magnitude 100000 :dimensions "f" :units "lbf")
   (net-thrust
      :documentation "Net thrust of all engines."
      :order-of-magnitude 10000 :dimensions "f" :units "lbf")
   (net-drag
      :documentation "Net vehicle drag."
      :order-of-magnitude 10000 :dimensions "f" :units "lbf"))
  (design-link)
(:category mission-performance gross-properties)
(:documentation
   "Provides attributes and collector constraints for computing
    vehicle gross properties."))
```

Note that this link has no linkages. It instead relies on collector constraints for determining the net height, weight, lift, thrust, and drag for the vehicle. Using this approach, there is no need to know the exact component breakdown of the aircraft—as would be required in order to specify linkages for each of the components—in order to be able to compose the appropriate maximum-value and summation constraints.

Based on this link-class definition, then, the collector constraint for determining the net lift for the vehicle might take the form:

```
(defcollector (aircraft-gross-properties "Vehicle Net Lift" :sum)
      net-lift
      (:name-contains "lift")
   "Collects lift attributes to compute overall vehicle lift.")
```

Note however, that this definition is somewhat flawed: the attribute which serves as its output parameter, `net-lift`, also satisfies the collection predicate, since its name, `"net-lift"`, contains the string `"lift"` as a substring. Having an attribute appear as variable on both sides of a summation or Π-product equation is mathematically nonsensical; the constraint definition is in error. Fortunately, the *Rubber Airplane* program explicitly checks for this condition, so that the above definition would, in fact, be acceptable. The following specification, however is better formed and explicitly avoids this problem:

```
(defcollector (aircraft-gross-properties "Vehicle Net Lift" :sum)
      net-lift
      (:and (:owner-class design-component) (:name-contains "lift"))
```

```
    "Collects lift attributes to compute overall vehicle lift.")
```

By restricting the class of design entities to be examined for the desired attributes, the link-class's own **net-lift** attribute becomes ineligible for collection, since its owner, which will be an instance of link-class **aircraft-gross-properties**, does not include the **design-component** component-class as a superclass. This form of the constraint is also somewhat clearer in conveying the intention of the constraint, since it explicitly indicates that the summation is to be applied over the components of the design. In addition, it is also more efficient than the original constraint definition, since the name comparison is only applied to attributes whose owners first satisfy the class restriction.

The other three summation collector constraints for this link class, corresponding to its three remaining attributes, may be defined using the **defsum** macro, as follows:

```
(defsum (aircraft-gross-properties "Vehicle Net Drag") net-drag
        (:and (:owner-class design-component) (:name-contains "drag"))
    "Collects drag attributes to compute overall vehicle drag.")

(defsum (aircraft-gross-properties "Vehicle Net Thrust") net-thrust
        (:and (:owner-class design-component) (:name-contains "thrust"))
    "Collects thrust attributes to compute overall vehicle thrust.")

(defsum (aircraft-gross-properties "Vehicle Gross Weight")
        gross-weight
        (:and (:owner-class design-component) (:named "weight"))
    "Collects weight attributes to compute overall vehicle weight.")
```

Note that the collection predicate for the third constraint, **"Vehicle Gross Weight"**, employs the **:named** predicate, rather than the **:name-contains** predicate. This is because, as indicated in Section 2.2.5, the built-in **design-component** component-class includes specification of a **weight** attribute. Thus, the less-specific **:name-contains** predicate is not needed, since the exact name of the **weight** attribute is known in advance.

Finally, the **defmax** macro may be used to define a collector constraint which computes the height of the vehicle (in order to check hangar clearance, for example). Assuming that each component specifies a **z-max** attribute which represents the maximum z-coordinate of its geometry, the overall height of the vehicle may be computed using a constraint of the form

```
(defmax (aircraft-gross-properties "Vehicle Height") overall-height
        (:and (:owner-class design-component) (:named z-max))
    "Finds maximum z-coordinate among all component-attributes.")
```

where, as was the case with the "Vehicle Gross Weight" constraint, the :named collector predicate can be used since z-max has been assumed to be a universal attribute, common to every component.

2.4 Design State Representation

2.4.1 Overview of Design States

It is the role of design states, the third and final type of design entity implemented for *Rubber Airplane*, to aid in the representation of time-varying phenomenon. This is accomplished by automatically creating new instances of pre-selected attributes and constraints as new design states are added. These attributes and constraints are referred to as being **state-dependent**, and each state-specific copy of such an attribute or constraint is referred to as a **state-instance** of that attribute or constraint. Thus, while designing an aircraft, attributes for vehicle lift, drag, weight, and thrust might be designated as state-dependent attributes; by instantiating a new design state for each flight leg of the aircraft's mission profile, a new state-instance for each of these attributes is created, one for each flight segment. Design states might also be used to model varying orbital conditions for a spacecraft, or the mode-specific behaviors of some mechanical device.

Design states, like components and links, are defined as classes, with their own attributes and constraints. As discussed below, however, this feature turns out to be of little practical value: programming of design knowledge is simplified if design states do not specify their own attributes or constraints, but merely serve as "placeholders" for the state-dependent attributes and constraints defined by a design's components and links. Since, nevertheless, means have been provided for defining state-classes in *Rubber Airplane*, the method for doing so is presented below. Following this discussion, Section 2.4.3 will describe *Rubber Airplane*'s provisions for declaring the state-dependency of component- and link-class attributes and constraints.

2.4.2 Definition of State-Classes

State Definition Syntax

State-classes are defined in much the same fashion as component- and link-classes. Specifically, the **defstate** macro is provided for this purpose, the syntax of which is as follows

```
(defstate entity-class
        attribute-specifications
        superclasses
    option
```

option
...)

As with **defcomponent** and **deflink** the *entity-class* argument should be a LISP symbol which names the state-class. This symbol will serve as the name of the state-class, and is used when defining attributes and constraints for the class by means of the **defattribute** and **defconstraint** macros (see Sections 2.2.2 and 2.2.3), as well as when referring to the class in a linkage specification or collection predicate, or as the direct superclass of another state-class. The *superclasses* argument is a list of symbols which name the state-classes which will serve as the direct superclasses of the current state-class. The construction of class precedence lists for state-classes follows the same algorithm as for component- and link-classes; as indicated in Section 2.2.4, the ordering of classes within the list of superclasses is significant.

The final set of arguments to the **defstate** macro allows for the selection of various class-specific options, much like those provided for the **defcomponent** and **deflink** macros. A listing of the available options appears in Table 2.8; as can be seen in Table 2.8 these options provide analogous features to the corresponding **defcomponent** and **deflink** options. Similarly, specification of **defstate** options is not mandatory, though declaration of the **:category** option is required for interactive instantiation of state-classes using the *Rubber Airplane* program's user-interface.

The Base State-Class

In addition to the base component- and link-classes, **design-component** and **design-link**, a base state-class, **design-state** is also provided. All instantiable state-classes (i.e., those which do not include specification of the **:abstract-state** option as part of their **defstate** definition) are required by the program to include class **design-state** as a superclass. Like the **design-link** link-class, **design-state** does not define any attributes or constraints of its own; its use provides a standard means for identifying design states within the *Rubber Airplane* class hierarchy, and when employing the various class-based linkage and collection predicates (see Sections 2.3.2 and 2.3.4). Furthermore, as discussed in greater detail below, it is usually unnecessary to directly associate attributes and constraints with state-classes. For most purposes, then, it is the case that **design-state** is the only state-class needed to support the desired design analyses. The **defstate** definition for the **design-state** state-class is given in Figure 2.4.

Option	Description
(:documentation *string*)	Specifies *string* as the documentation-string for the state-class.
(:comment *string*)	Specifies *string* as an ignored comment. Provided for annotation of the **defstate** code itself.
(:category *cat1 cat2 ... catn*)	Declares a hierarchical path of library categories for locating the state-class. Category *cat1* will be a top-level category, and *cat2* through *catn* will be successive sub-categories. Categories not already present are automatically created.
:abstract-state	Declares that the state-class is an abstract class, which may not be instantiated, but may serve as a superclass for another state-class.
(:required-states *state1 state2 ...*)	Declares *state1*, *state2*, ... as state-classes which must be present as (indirect) superclasses of the state-class, upon instantiation. The arguments *state1*, *state2*, ... should be symbols naming defined state-classes. Only valid in conjunction with the :abstract-state option.
(:required-attributes (*att1 dims1*) (*att2 dims2*) ...)	Declares *att1*, *att2*, ... as attributes, with corresponding dimensions *dims1*, *dims2*, ..., which must be present as (inherited) attributes of the state-class, upon instantiation. Arguments *att1*, *att2*, ... should be symbols or strings, and arguments *dims1*, *dims2*, ... should be dimension-strings. Only valid in conjunction with the :abstract-state option.

Table 2.8: Supported options for the **defstate** macro.

```
(defstate design-state
        () ()
  (:documentation
      "Basic state type.  Has no attributes or constraints of its own.
       Note that this state-class is instantiable; it is not
       an :ABSTRACT-STATE."))
```

Figure 2.5: LISP definition for the base state-class, **design-state**.

Example State Definitions

The first example state-class is an abstract class which can be instantiated to provide the design with access to the altitude-dependent properties of the atmosphere in which the vehicle is to be operating. The definition for this **atmosphere-mixin** state-class, omitting the relevant constraints, is

```
(defstate atmosphere-mixin
      ((altitude
          :documentation "Altitude for vehicle operation."
          :low-value -1000 :order-of-magnitude 30000
          :dimensions "l" :units "ft")
       (density
          :documentation "Ambient density at altitude."
          :low-value 0 :order-of-magnitude 1 :high-value 2
          :dimensions "m l-3" :units "kg m-3")
       (pressure
          :documentation "Ambient pressure at altitude."
          :low-value 0 :order-of-magnitude 1 :high-value 2
          :dimensions "p" :units "atm")
       (temperature
          :documentation "Ambient temperature at altitude."
          :low-value 0 :order-of-magnitude 15 :high-value 40
          :dimensions ""))
      ()
   :abstract-state
   (:category flight-conditions atmosphere mixins)
   (:documentation
      "Provides attributes and constraints for representing
       standard Earth atmosphere."))
```

This state-class specifies attributes for the atmospheric properties (density, temperature, and pressure), as well as an attribute for the altitude at which the vehicle is to operate. It is assumed that appropriate constraints for relating these properties to altitude are also associated with the class.

Based upon the definition of this abstract state-class, an instantiable subclass which represents an aircraft flight condition may be defined:

```
(defstate flight-condition
      ((alpha
```

```
      :documentation "Vehicle angle of attack."
      :low-value -10 :order-of-magnitude 0 :high-value 15
      :dimensions "A" :units "deg")
   (speed
      :documentation "Vehicle flight speed."
      :low-value 75 :order-of-magnitude 500 :high-value 700
      :dimensions "l t-1" :units "kt"))
  (atmosphere-mixin design-state)
(:category flight-conditions)
(:documentation
  "Provides attributes for representing a single aircraft
   flight condition."))
```

By creating one instance of this **flight-condition** state-class for each of the flight-legs associated with the design's mission-profile, individual attributes for the atmospheric and operational properties for each segment are made available to the constraints which model the relevant design relationships.

2.4.3 State-Dependent Attributes and Constraints

Specification of State-Dependency

As discussed above, state-dependent attributes and constraints are those attributes and constraints for which multiple copies are to be created, one for each design state present in the design. Of course, most attributes and constraints—such as those describing component geometry (e.g., fuselage length, landing gear dimensions) or those modeling fixed or single-occurrence behaviors (e.g., operating costs, takeoff distance)—need not be treated as state-dependent.[7] A number of design parameters, however—such as gear deployment, flight conditions, and vehicle weight—do represent time-varying phenomenon, and are best implementing using state-dependent attributes and constraints.

To take advantage of the presence of design states, then, some means is needed for designating those attributes and constraints which are to be considered state-dependent. To implement annotation of state-dependent attributes, an additional keyword argument, **:state-dependent?**, has been added to those supported for attribute specifications via **defattribute**, **defcomponent**, and **deflink**. The argument for this keyword should be either **t** or **nil**, which are the LISP symbols used to denote boolean truth values of "true" and "false", respectively. Note that if the **:state-dependent?** specification is not explicitly

[7] For convenience, such attributes and constraints will be referred to as state-independent.

declared in the definition of an attribute, the attribute is by default assumed to be state-independent.

As an example, consider an alternative implementation of the **atmosphere-mixin** state-class, presented above, as a design link. Each of its attributes would then have to be annotated as being state-dependent, in order to achieve the same functionality as the original state-class. Thus, the link-class and its attributes would be defined as follows:

```
(deflink atmosphere-mixin
      ()
      ((altitude
          :documentation "Altitude for vehicle operation."
          :state-dependent?  t
          :low-value -1000 :order-of-magnitude 30000
          :dimensions "l" :units "ft")
       (density
          :documentation "Ambient density at altitude."
          :state-dependent?  t
          :low-value 0 :order-of-magnitude 1 :high-value 2
          :dimensions "m l-3" :units "kg m-3")
       (pressure
          :documentation "Ambient pressure at altitude."
          :state-dependent?  t
          :low-value 0 :order-of-magnitude 1 :high-value 2
          :dimensions "p" :units "atm"))
      ()
   :abstract-link
   (:category flight-conditions atmosphere mixins)
   (:documentation
      "Provides attributes and constraints for representing
       standard Earth atmosphere."))

(defattribute (atmosphere-mixin temperature)
   :documentation "Ambient temperature at altitude."
   :state-dependent?  t
   :low-value 0 :order-of-magnitude 15 :high-value 40
   :dimensions "")
```

where the definition of the **temperature** has been moved to a **defattribute** statement in order to show the use of the state-dependent annotation with both syntax forms. Because

Rubber Airplane does not permit mixing of design-entity types when specifying superclasses, the **flight-conditions** state-class would also have to be redefined as a link-class:

```
(deflink flight-conditions
      ((alpha
          :documentation "Vehicle angle of attack."
          :state-dependent?  t
          :low-value -10 :order-of-magnitude 0 :high-value 15
          :dimensions "A" :units "deg")
       (speed
          :documentation "Vehicle flight speed."
          :state-dependent?  t
          :low-value 75 :order-of-magnitude 500 :high-value 700
          :dimensions "l t-1" :units "kt"))
      (atmosphere-mixin design-link)
   (:category flight-conditions)
   (:documentation
      "Provides attributes for representing a single aircraft
       flight condition."))
```

As with the attributes of the **atmosphere-mixin** link-class, the attributes of this **flight-conditions** link-class must also be declared state-dependent.

Note, however, that no additional syntax is required for defining state-dependent constraints. Any constraint associated with a component- or link-class which employs a state-dependent attribute as either an input or output parameter is automatically recognized as being a state-dependent constraint. For example, one might define for the **atmosphere-mixin** link-class introduced above a constraint representing the state equation of a gas:

```
(defconstraint (atmosphere-mixin "State Equation")
       (pressure "N") ((density "kg m-3") (temperature "K"))
   "Implements the state equation of a gas as an equality constraint."
   (let ((R 287.0)) ;; Units are "m2 s-2 K-1"
     (* density R temperature)))
```

Since all three of the parameters for this constraint are state-dependent, the constraint itself must be state-dependent.

In practice, it is observed that every state-dependent constraint has at least one state-dependent input parameter, and *all* of its output parameters are state-dependent. A state-dependent constraint which does not follow this pattern is pathological:

- If a constraint specifies state-dependent output parameters but no state-dependent input parameters, then it must be the case that all copies of these state-dependent attributes will have the same values. If this is so, then state-dependent attributes are not needed.

- If a constraint specifies one state-dependent input parameter and no state-dependent output parameters, then it must be the case that all copies of this state-dependent attribute will have the same value. Again, if this is so, a state-dependent is not needed.

- If a constraint specifies multiple state-dependent input parameters but all output parameters are state-independent, then it is mathematically consistent, but it is not likely that the constraint has been specified in its clearest form. This case is not strictly erroneous, but it is very likely that the constraint definition could be improved.

- If a constraint specifies multiple output parameters, some of which are state-dependent, but some are not, then it will be the case that, since the constraint must necessarily be a `:one-way` constraint and therefore not invertible, the multiple copies of this state-dependent constraint will compute multiple and potentially contradictory values for the state-independent attributes. For this reason, such a constraint is not well-formed.

These observations allow *Rubber Airplane* to perform additional error-checking on constraints involving state-dependent attributes. Constraints which do not fit the pattern described above are flagged as erroneous, and suggestions for repairing or improving the constraint definition may be offered.

Finally, note that association of state-dependent attributes and constraints with state-classes is not permitted. Such a specification would not be logically consistent: given three instances of a state-class which specifies a single state-dependent attribute, should there be nine copies of this attribute (i.e., three copies—corresponding to the three design states—for each of the three design states), or just three (i.e., exactly one for each design state)? *Rubber Airplane* signals an error when the user attempts to define a state-dependent attribute or constraint for a state-class.

Design Links and State-Dependency

As mentioned in the preceding section, design constraints need not be explicitly annotated as being state-dependent. This is true even of constraints associated with link-classes, though identification of state-dependent constraints must sometimes be delayed until instantiation of the constraint specifications. For components, state-dependent constraints may be identified when they are defined. This is because constraints associated with a component-class

may only reference the class's own attributes; determination of which attributes are state-dependent, if any, is readily performed. Because constraints associated with a link-class may also reference the attributes of its linkages, such determination is not always possible until the link-class is instantiated and the linkages' attributes become available. For this reason, it is also the case that errors in constraints which reference state-dependent linkage-attributes—such as those described in the preceding section—cannot always be recognized prior to constraint instantiation. Unfortunately, it becomes the responsibility of the user defining such constraints to ensure their consistency and correctness, if instantiation errors are to be avoided.

An additional complication involving link-class constraints and state-dependency concerns those constraints which need to access only particular state-instances of a state-dependent attribute. In the case where the state-dependent attribute is owned by the same link-class as such a constraint, then it is only necessary to specify the desired design states as linkages of the link-class, in which case the normal "@" syntax may be used to access the desired attributes. For instance, if the **atmosphere-mixin** link-class defined above wished to constrain the state-instance of the angle-of-attack attribute, **alpha**, corresponding to a design state named **"Cruise"**, it would first be necessary to allocate a linkage for this design state. The specification for such a linkage might be:

```
(cruise-state :class design-state)
```

When the link-class is instantiated, design state **"Cruise"** should be selected as the design entity for this linkage-specification; then, any constraint which references the attribute **alpha@cruise-state** will access the desired state-instance.

It is more often the case, however, that a link-class would be required to constrain a particular state-instance of one of its linkage's state-dependant attributes. In this case, the conventional "@" syntax is insufficient. For this reason, a double-"@" form is introduced to augment the original syntax. Once again, it is necessary to specify linkages for the design states corresponding to the desired state-instances. Using this new syntax, then, the format for referencing state-dependent attributes of linkages is

att-name@att-linkage@state-linkage

where *att-name* is the name of the state-dependent attribute associated with the linkage whose name appears in the *att-linkage* position, and *state-linkage* is the name of the linkage which references the appropriate design state. Thus, if a link-class named **takeoff-analysis** needs to access the state-instance of the state-dependent **density** attribute of the **flight-conditions** link-class associated with a **"Takeoff"** design state, it would be necessary for the **takeoff-analysis** link-class to specify two linkages, one for the **flight-conditions** link-class, and one for the **design-state** instance, as follows:

```
((flight-conditions :class flight-conditions)
 (takeoff-state :class design-state))
```

Assuming the "Takeoff" design state is selected as the **takeoff-state** linkage, then the name "**den y@flight-conditions@takeoff-state**" may be used to access the desired attribute.

2.4.4 Impracticality of design-state Subclasses

In Section 2.4.2, above, an example **flight-conditions** class is introduced to demonstrate the use of the **defstate** macro for defining state-classes. In Section 2.4.3, however, this class is redefined as a link-class with state-dependent attributes. At first, it might appear that, in actual use, the state-class representation is preferred, since the parameters associated with the class—flight speed, air density, etc.—directly describe a given design point, and, after all, that is the intended role of design states. Furthermore, implementation as a link-class introduces the need for the rather awkward double-"@" syntax, discussed above.

On the other hand, it is certainly the case that these parameters are, indeed, state-dependent, so perhaps the link-class formulation has certain advantages, as well. Indeed, as was indicated earlier, it was observed during the course of implementing and using design states, that direct association of attributes and constraints with state-classes is undesirable; the benefits of associating state-dependent attributes with component- and link-classes far outweigh the aesthetic advantages of implementing them as state-independent attributes of state-classes. These advantages are due primarily to the ability of *Rubber Airplane* to transparently define state-dependent constraints. If a state-dependent parameter, such as air density, is implemented as a state-independent attribute of a state-class, it becomes impossible to define a state-dependent constraint on this parameter. Individual constraints, which reference separate linkages for each of the state-class instance which provide this attribute, must be defined.

As a simple example of such a constraint, consider the definition of the lift coefficient of an airfoil:

$$C_L = \frac{L}{\frac{1}{2}\rho V^2 S}$$

where C_L is the lift coefficient, L is the airfoil's actual lift, ρ is the atmospheric density, V is the flight speed, and S is the airfoil's wing area. Whereas the lift, lift coefficient, and wing area might be implemented as attributes of an **airfoil** component-class, the flight speed and air density would likely be implemented either as attributes of a design state, or as state-dependent attributes of a design link. In either event, the constraint which implements this mathematical relationship would have to be associated with a link-class, such as, say, **airfoil-aerodynamics**, since it simultaneously references attributes from

multiple design entities. If the latter approach is taken, then only two linkages are required for the hypothetical **airfoil-aerodynamics** link-class:

```
((flight-conditions :class flight-conditions)
 (wing :class airfoil))
```

Assuming that component-class **airfoil** specifies a **wing-area** attribute, as well as state-dependent **lift** and **lift-coefficient** attributes, and that **density** and **speed** are state-dependent attributes of link-class **flight-conditions**, the lift-coefficient constraint may be implemented as follows:

```
(defconstraint (airfoil-aerodynamics "Lift Coefficient")
       (lift-coefficient@wing "")
       ((lift@wing "N" L) (density@flight-conditions "kg m-3" rho)
        (speed@flight-conditions "m s-1" V) (wing-area@wing "m2" S))
   "Implements the definition of airfoil lift coefficient."
   (/ L (* 1/2 rho V V S)))
```

This single constraint specification will yield multiple state-instances, one for each design state present in the design.

If, on the other hand, the flight-condition parameters are associated with a state-class, instead of a link-class, multiple constraint definitions and multiple linkage-specifications, one for each expected design state, will be required. This approach is not modular, since the link must be modified if the set of design states is increased or decreased, and negates many of the benefits originally derived from the introduction of design states (see Section 2.1.3). For this reason, the association of attribute and constraint specifications with state-classes, while supported by *Rubber Airplane*, is not advised. Thus, there should never be any need to define additional state-classes beyond the built-in **design-state** class.

2.5 Additional Representational Issues

2.5.1 Design Entity Instantiation

As mentioned briefly in Section 2.1.3, component-, link-, and state-classes are represented by **defstruct** data structures. These data structure have slots for storing the names, options, names of superclasses, and class precedence lists associated with design-entity classes, as well as for lists of their attribute and constraint specifications. Component-class structures also have a slot for storing component-geometry descriptions, which relate the values of a component's attributes to its three-dimensional geometry (see Appendix B); link-class structures are provided with a slot for storing linkage specifications. Attribute and constraint specifications are themselves stored as **defstruct** data structures. Different

defstruct-types are provided for scalar and discrete attributes, and well as for equality, inequality, and collector constraints, in order to accommodate the varying storage needs for these specifications.

When instantiating design entities, Flavor instances are created. There are three basic Flavor-types, one for each of the three types of design entities. These Flavor structures include instance variables for storing a design entity's name, the class structure upon which it is based, and lists of its attributes and constraints. Design-component instances also include instance variables for interfacing between its attributes and geometry; design-link instances include an instance variable which stores an association list[8] which groups linkage names with the design entities which serve as the actual linkages.

Instantiation of a design-entity class also results in instantiation of the attribute and constraint specifications associated with the class, including any inherited specifications. Attributes and constraints are also represented by Flavor instances. Instance variables are provided for the various properties included in the specifications. In addition, attribute instances include instance variables which list those constraints for which the attribute is an input parameter, and those for which it is an output parameter. Similarly, constraint instances include instance variables which list the attributes which serve as their input and output parameters. The presence of these lists is intended to facilitate constraint propagation, as discussed in Chapter 3.

State-dependent attributes and constraints are also represented by Flavor instances. These instances maintain association lists between their state-instances and the design states for which they have been created. They are also responsible for producing the appropriate state-instances when new design states are added, and for destroying state-instances as unwanted design states are removed. Note that, for each design (see Section 2.5.2), one design state is designated as the current or **focus state**. When a state-dependent attribute or constraint receives a request for state-specific data or to display itself, the request is automatically forwarded to the state-instance corresponding to the design's focus state.

2.5.2 Auxiliary Data Structures

The *Rubber Airplane* program allows users to work on more than one design project at the same time. Each such project is represented by a *Rubber Airplane* **design** object, each of which maintains its own independent set of components, links, and states. Individual designs may be saved into text-files, which stores both the contents of a design (i.e., its set

[8] An association list is a LISP data structure comprised of a list of pairs. The first element of each pair is considered to be a "key", based upon which the second element of the pair—the "data"—may be retrieved. Common Lisp includes built-in functions for creating and manipulating association lists, including both a function for looking up the "data" item associated with a given "key", and a function for retrieving the "key" associated with a given piece of "data".

of design entities), as well as its status (i.e., all user-assigned values for its design-entities' attributes). These design objects are themselves controlled by *Rubber Airplane*'s so-called **top-level object**, which keeps a list of all the designs. This top-level object also controls the program's user-interface (see Appendix C), which allows the user to display and manipulate both textual and graphical representations of any one of the current designs.

Chapter 3

Constraint Propagation

3.1 Introduction

Component-modeling, as discussed in Chapter 2, is introduced as a means for adding flexibility to the process of modeling a design problem. By taking advantage of object-oriented programming techniques, component-modeling allows a description of the design task to be developed incrementally, by adding, removing, and modifying the components, links, and states which are used to represent the design.

Constraint propagation is similarly employed to add flexibility to the mathematical analyses associated with the design process. As indicated in Chapter 1, constraint propagation allows a single declarative statement of a mathematical equation—such as $a = b + c$—to be used to infer multiple imperative forms—in this case, $b = a - c$ and $c = a - b$. Thus, the mathematical relationships which govern a design problem may be treated as data to be manipulated by the program, rather than as explicit instructions for calculating a specific design parameter (i.e., the variable appearing on the left-hand side of the equation). Design relationships may be used to compute any of their associated parameters, based on known values for the remaining parameters. Furthermore, the order in which the relationships are applied may be varied; constraints are applied as the supporting parameter-values become available. Use of constraint propagation thus frees both the programmer and the user from concern over the direction and sequencing of program calculations, relegating these decisions to the computer.

In this chapter, the algorithms employed by *Rubber Airplane* to implement constraint propagation are presented. Section 3.2 discusses the algorithm responsible for local propagation of *Rubber Airplane* constraints. Section 3.4 introduces modifications to this algorithm which support the detection and solution of simultaneous equations within the constraint network. Section 3.5 briefly describes an experimental integration of the local-propagation algorithm with a Fortran-based optimization program.

Finally, note that, unlike its predecessor *Paper Airplane*, *Rubber Airplane* attempts

to perform all constraint propagation—both local and loop-based—interactively. As fixed values are assigned to the base variables by the user, computations are performed automatically; there is no need for the user to request that constraint propagation take place. This alternative approach is possible because of the superior performance of the Lisp Machine hardware for which *Rubber Airplane* has been written in comparision to the more conventional computer architecture for which *Paper Airplane* was developed. Thus, while there are some similarities in the algorithms used by the two programs (cf. Reference [20]), the differences which are present are primarily motivated by the desire to improve the utility of constraint propagation by allowing it to occur interactively.

3.2 Local Propagation

3.2.1 Overview

Local propagation is the process whereby *individual* constraints are propagated, and the results made available for further propagation. Whenever a value is supplied for one of a constraint's parameters, this constraint is examined for possible propagation. In the case of single-output constraints, if values are available for all but one of a constraint's parameters, the constraint may be used to calculate a value for the remaining, unknown parameter. The availability of this computed value may enable further constraint propagations; the process thus repeats itself, recursively. Multiple-output constraints—which are not invertible—may be propagated only when values are available for all of the constraint's input parameters.

Note that, during the course of local propagation, constraints are examined singly. No global analysis is performed; local propagation does not, therefore, enable either the identification or processing of cycles within a network of constraints. Although such cycles may represent a mathematically solvable set of simultaneous equations, local propagation can only be used to propagate constraints individually. While it is often the case that simultaneous equations may be solved via local propagation through appropriate sequencing of the relevant constraints, there are also circumstances in which this is not possible (e.g., two equations in two unknowns). Local propagation must be augmented if the solution of constraint cycles is required.

3.2.2 Data Structures

As indicated in Chapter 2, attributes and constraints are represented as Flavor instances. Several of the instance variables associated with these data structures are used to support constraint propagation.

Values of both scalar and discrete attributes may be computed via local propagation (though, as indicated in Chapter 2, only uni-directional constraints may reference discrete

attributes). Instance variables associated with both types of attributes which are accessed during constraint propagation are listed in Table 3.1.

Note that *Rubber Airplane* permits the user to supply new values for the *order-of-magnitude*, *low-value*, and *high-value* instance variables of scalar attributes, in order to override those provided by the corresponding attribute specifications. In this way, unusual design circumstances need not require modification of class definitions; individual attribute-instances may themselves be modified. The value declared in the corresponding attribute specification for the *value-list* instance variable of a discrete attribute, however, may not be overridden.

Constraint propagation is applied to both omni-directional and uni-directional (i.e., **:one-way**) equality constraints. In the latter case, although constraint inversion is prohibited, constraint propagation may still be employed to order constraint application. The constraint instance variables which are used during propagation are listed in Table 3.2. Note that the additional instance variable required for invertible constraints, *compiled-normal-function*, acts as an error term for the constraint, indicating the degree to which the current values of the input and output parameters satisfy the constraint.

3.2.3 Algorithm for Local Propagation

The implementation of local constraint propagation employed by *Rubber Airplane* is based upon monitoring of the **value-supplier** associated with each attribute, which is stored by its *value-supplier* instance variable. Whenever a value is assigned to an attribute, specification of a value-supplier for that value is also required. At any given time, each attribute will have one of three possible values for its value-supplier:

- a constraint, implying that the attribute's current value was computed using that constraint;
- the keyword "**:user**", indicating that the current value has been explicitly assigned to the attribute by the user; or
- the keyword "**:guess**", representing the lack of a reliable value for the attribute.

Values supplied by the user, as well as those computed by constraints, are assumed fixed. Only when an attribute's value-supplier is **:guess** may constraint propagation be employed to calculate a new value for it. Such attributes are therefore referred to as **free parameters** of the constraint.

Initially, all attributes have a default value-supplier of **:guess**, assigned during attribute instantiation. Whenever an attribute's value becomes fixed (i.e., its value-supplier is either **:user** or a constraint), the attribute polls all of its constraints, stored in its *forward-constraints* and *reverse-constraints* instance variables, to see if any of them are **perfectly**

Object Type	Instance Variable	Description
Attribute	*value*	Stores the value of the attribute instance, in *Rubber Airplane* internal units.
	value-supplier	Indicates the source for the attribute's current value. Should be either a constraint (indicated that the value was computed using the constraint), the keyword **:user** (indicating that the value was explicitly assigned to the attribute by the program operator), or the keyword **:guess** (indicating that, lacking any other source, a default value was assigned by the program).
	time-tag	Stores the time at which the current value was assigned, represented in Common Lisp "Universal Time" format (see Reference [35]). The value of this instance variable is updated whenever a new value is assigned by the user, or computed by a constraint.
	forward-constraints	Maintains a list of the constraints for which this attribute instance is an output parameter.
	reverse-constraints	Maintains a list of the constraints for which this attribute instance is an input parameter.
Scalar Attribute	*order-of-magnitude*	Stores the order-of-magnitude specification for the attribute instance, in *Rubber Airplane* internal units.
	low-value	Stores the low-value specification for the attribute, in *Rubber Airplane* internal units.
	high-value	Stores the high-value specification for the attribute, in *Rubber Airplane* internal units.
Discrete Attribute	*value-list*	Stores a list of the possible values for the attribute, or a symbol which evaluates to such a list.

Table 3.1: Attribute instance variables used in constraint propagation.

Object Type	Instance Variable	Description
Equality Constraint	*output-parameters*	Stores a list of the attributes which serve as output parameters for the constraint.
	input-parameters	Stores a list of the attributes which serve as input parameters for the constraint.
	parameters	Stores a combined list of the input and output parameters for the constraint.
	computed-parameters	Maintains a list of the attributes whose current values were computed using the constraint.
	compiled-function	Stores a compiled LISP function which performs any required unit conversions and executes the operations listed in the body of the corresponding constraint specification.
Invertible Constraint	*compiled-normal-function*	Stores a compiled LISP function which calls the function stored by the *compiled-function* instance variable, and subtracts the result from the current value of the constraint's input parameter. (Note that all invertible constraints have but a single output parameter).

Table 3.2: Constraint instance variables used in constraint propagation.

constrained. Defining the **degree** of a constraint as the count of its output parameters, a constraint is considered to be "perfectly constrained" whenever the number of the constraint's attributes whose value-suppliers are **:guess** is equal to its degree. The degree of a constraint indicates the number of parameters which may be computed using the constraint, whether it is used "as is", or is inverted; thus, a constraint only becomes perfectly constrained when the number of free parameters of the constraint matches its degree.

If any such perfectly constrained constraints are found, they are immediately used to compute values for their free parameters. The constraint then serves as the value-supplier for these free parameters, whose values thus become fixed, causing the constraint propagation process to be called recursively. Note that, once a constraint has been used to compute a value for one or more of its attributes in this manner, it will no longer have any free parameters, and thus there is no danger of it being propagated again.

Actually, in the case of uni-directional (i.e., **:one-way**) constraints, propagation may only take place when the set of free parameters exactly matches the constraint's set of output parameters. Since it is the case that all multiple-output constraints are uni-directional, it turns out that some simplification in the algorithm is possible. If the degree of a constraint is unity, then propagation may be used to compute any free parameter. If the degree is greater than one, then propagation can only occur when the free parameters match the output parameters.

Value-suppliers are also used to implement constraint retractions. The values and value-suppliers of computed attributes may not be changed. User-supplied values, however, can be retracted. This is done by changing the value-supplier of the appropriate attribute from **:user** to **:guess**. When this takes place, each of the attribute's constraints, as listed in its *forward-constraints* and *reverse-constraints* instance variables, are examined. If any of these constraints has one or more other attributes listed in its *computed-parameters* instance variables, these attributes are removed from the list and their value-suppliers are revised: rather than listing the constraint as their value-suppliers, the value-supplier of each will be changed to **:guess**. This change may itself yield further retractions; like propagation, constraint retraction can call itself recursively.

Note that, when retracting a calculation, only the value-supplier of an attribute is changed—its value is unaffected, and remains set to the value previously computed by the constraint. This is done for reasons of both simplicity and practicality. With respect to practicality, it is observed that the numerical inversion and loop-solving techniques discussed below are most reliable when applied to computing the results of small, incremental changes in attribute values. Maintaining computed values beyond constraint retraction makes them available to the numerical algorithms, thereby improving program performance.

Similarly, if the user wishes to assign a new user-supplied value to an attribute, the old user-supplied value must first be retracted. This is done to simplify maintenance of the proper dependency relationships. Note that it is not actually necessary for the user to explicitly perform these steps; if an attribute's current value-supplier is **:user** and the user wishes to change its value without changing the supplier, the program interface automatically carries out a temporary change in the attribute's value-supplier from **:user** to **:guess**—causing the appropriate retractions to occur—before the new value is assigned, after which the original value-supplier, **:user**, is restored.

To summarize, local propagation and retraction of constraints is implemented as follows:

1. Initially, the *value-supplier* instance variable of all attributes is **:guess**.

2. All changes in the *value-supplier* of an attribute are monitored. Five types of changes are possible:

   (a) From **:guess** to **:guess**.

   (b) From **:guess** to **:user**.

   (c) From **:guess** to a constraint.

   (d) From **:user** to **:guess**.

   (e) From a constraint to **:guess**.

   Note that only Cases 2a, 2b, and 2d may be instigated by the user.

3. In Case 2a, nothing happens.

4. In Cases 2b and 2c, constraint propagation takes place:

   (a) Examine each of the constraints listed in the *forward-constraints* and *reverse-constraints* instance variables of the attribute, in order to identify those which are perfectly constrained. A constraint is perfectly constrained when it has the same number of output parameters as free parameters (i.e., attributes whose value-suppliers are **:guess**).

   (b) For each perfectly-constrained constraint:

   - If the constraint is omni-directional, it must be the case that there is exactly one free parameter. The constraint is used to compute the value of this free parameter, and may be inverted, if necessary. The value-supplier for this computed value is the constraint itself. The free parameter is added to the list maintained by the constraint's *computed-parameters* instance variable.

- If the constraint is uni-directional and the constraint's free parameters match its output parameters, use the constraint to compute values for its output parameters. The computed values are assigned to the appropriate attributes; the value-supplier for these new values is the constraint itself. The output parameters are added to the list maintained by the constraint's *computed-parameters* instance variable.
- All other cases would require the inversion of a uni-directional constraint; therefore, no action is taken.

5. In Cases 2d and 2e, constraint retraction takes place:

   (a) Examine each of the constraints listed in the *forward-constraints* and *reverse-constraints* instance variables of the attribute, in order to identify those whose *computed-parameters* instance variables are non-empty lists.

   (b) For each such constraint, iterate through the list of attributes which appear as the constraint's *computed-parameters*. The value-supplier of each of these attributes, which should be the constraint itself, is changed to **:guess**; their values remain unchanged. Each of the attributes is removed from the list maintained by the constraint's *computed-parameters* instance-variable.

As mentioned above and is apparent from this enumeration, since both propagation and retraction cause additional changes in the value-suppliers of the affected attributes, both of the two processes are recursive.

3.2.4 An Example

To illustrate the use of the algorithm detailed in the preceding section, consider a simple constraint network created by instantiating a hypothetical **airfoil** component-class, for which the following two constraints have been defined:

```
(defconstraint (airfoil "Definition of Aspect Ratio")
       (aspect-ratio "")
       ((span "m") (wing-area "m2" S))
   "Implements the definition of aspect ratio as an equality constraint."
   (/ (* span span) S)))

(defconstraint (airfoil "Finite-Wing Lift Slope")
       (cl-alpha "")
       ((cl-alpha_inf "") (oswald "") (aspect-ratio ""))
   "Approximates the effect of finite wingspan on lift-curve slope."
```

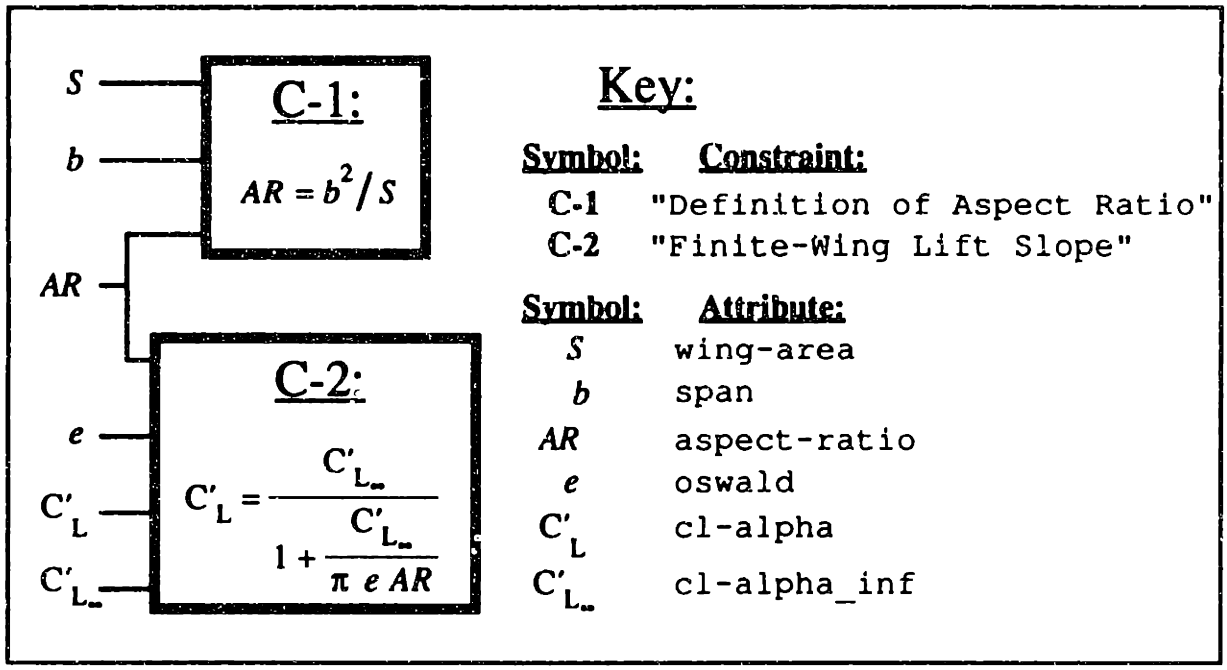

Figure 3.1: Constraint network for local propagation example.

```
(/ cl-alpha-inf (+ 1 (/ cl-alpha_inf pi oswald aspect-ratio))))
```

where it has been assumed that component-class **airfoil** also provides definitions for the required attributes. The **span**, **aspect-ratio**, and **wing-area** attributes referenced here represent the parameters which describe the planform geometry. Attribute **oswald** denotes the Oswald's efficiency factor for the airfoil, which is a measure of the ellipticity of the airfoil's lift distribution. The remaining two attributes, **cl-alpha** and **cl-alpha_inf**, represent the slope of the airfoil's lift-curve, with respect to angle of attack, for the finite and infinite aspect-ratio cases, respectively. The resulting constraint network is depicted graphically in Figure 3.1, where the attributes and constraints are written using mathematical, rather than LISP, notation. The correlation between the two notations is also given in Figure 3.1.

For this example, it is assumed that the user has provided values for the **wing-area**, **span**, **oswald**, and **cl-alpha** attributes (i.e., the planform and required performance have been specified, but the airfoil cross section has not been chosen). The value-supplier for each of these attributes is, therefore, **:user**, while **:guess** is the value-supplier for both the **aspect-ratio** and **cl-alpha_inf** attributes. These initial conditions are depicted in Figure 3.2(a), where the superscripts "U" and "G" are used to designate value suppliers of **:user** and **:guess**, respectively.

Initially (Figure 3.2(a)), the aspect-ratio constraint, C–1, has one free parameter, while the lift-slope constraint, C–2, has two. Constraint C–1 is thus perfectly constrained, and

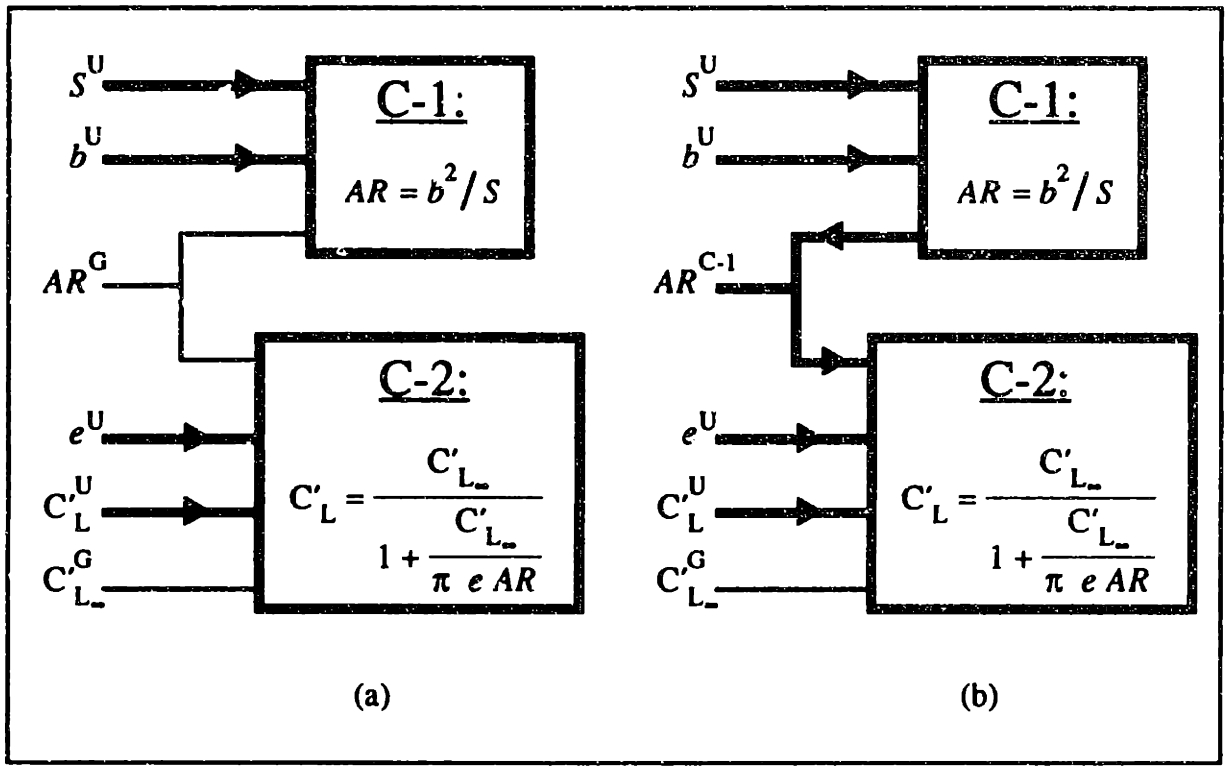

Figure 3.2: Constraint propagation of constraints C–1 and C–2 (a) before propagation of C–1, and (b) after propagation of C–1.

may be used to compute a value for **aspect-ratio** (AR). As indicated in Section 3.2.3, constraint C–1 will be examined whenever the value-supplier of any of its attributes is modified. Thus, once the value-supplier of both the **span** (b) and **wing-area** (S) attributes have been set, the program will immediately determine that constraint C–1 is perfectly constrained and ready for propagation. At this point, the appropriate calculation is performed, the new value is assigned, and the value-supplier of the **aspect-ratio** attribute becomes constraint C–1, as denoted by the superscript "C–1" in Figure 3.2(b). In addition, the **aspect-ratio** attribute becomes a **computed parameter** of constraint C–1, and is added to the list maintained by its *computed-parameters* instance variable, which was previously empty. Furthermore, since the value-supplier of **aspect-ratio** is no longer **:guess**, constraint C–2 has become perfectly constrained (Figure 3.3(a)). Its only remaining free parameter is **cl-alpha_inf** ($C_{L'_\infty}$), which may then be computed by inverting the mathematical relationship (see Figure 3.3(b)), and the value-supplier of **cl-alpha_inf** becomes constraint C–2, as indicated by the superscript "C–2" in Figure 3.3(b). Finally, the **cl-alpha_inf** attribute is added to constraint C–2's list of *computed-parameters* (which should previously have been empty). At this stage, since none of the constraints involving **cl-alpha_inf** are perfectly constrained, constraint propagation is terminated.

Thus, both constraints have been propagated in order to compute values for the **aspect-**

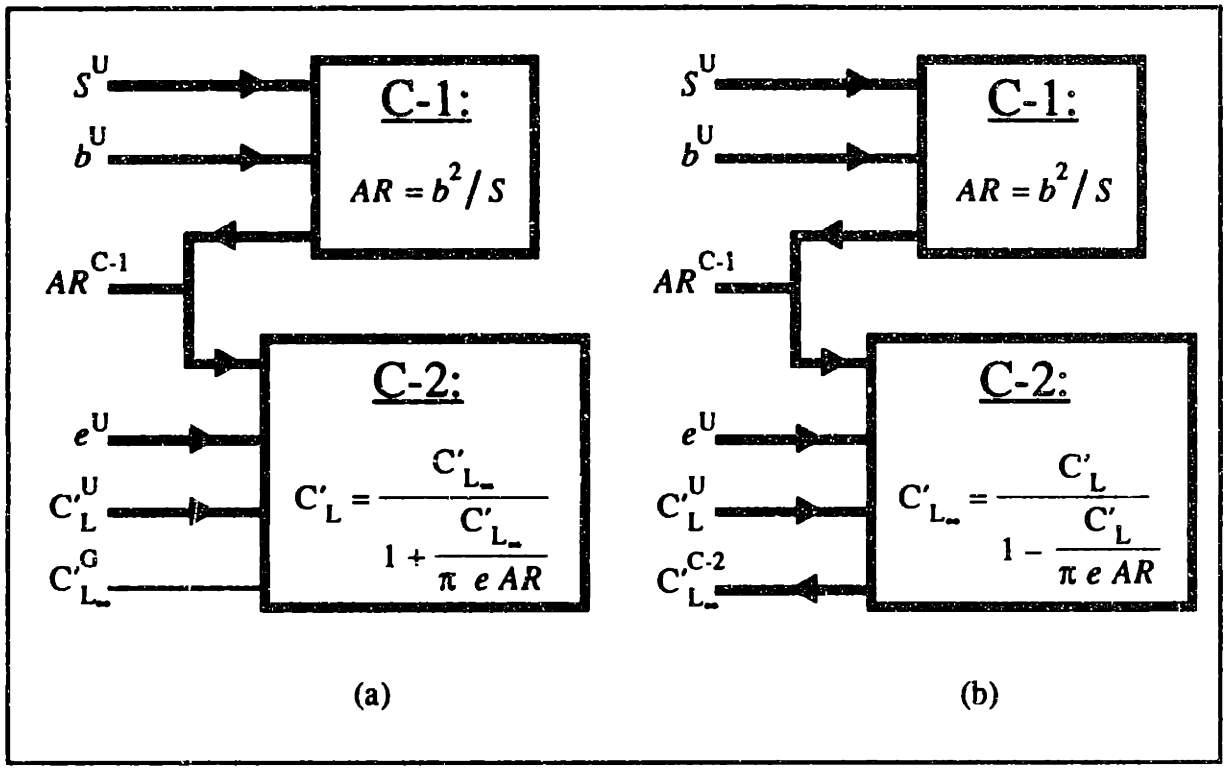

Figure 3.3: Constraint propagation of constraints C–1 and C–2 (a) before propagation of C–2, and (b) after propagation of C–2.

ratio and **cl-alpha_inf** attributes, based upon user-supplied values for the other attributes. If the user wishes to change the value of one of the latter attributes, however, the propagation must first be revoked, so that the new value may be propagated. (As mentioned in Section 3.2.3, the program interface automatically invokes constraint retraction whenever a user-supplied value is altered; the user need not initiate retraction as a separate operation.) Once retraction takes place, the new user-supplied value is assigned to the appropriate attribute, its value-supplier becomes **:user** once more, and constraint propagation proceeds as before.

To illustrate the retraction process, consider continuing the example introduced above, and retracting the user-supplied value for the **span** attribute. As indicated in Figure 3.4(a), the first step is to change the value-supplier of this attribute from **:user** to **:guess**. Next, any constraints which refer to the attribute are examined to see if they have any computed parameters. In this case, only constraint C–1 accesses **span**, and it does, indeed, have a single computed parameter, **aspect-ratio**. This computation is retracted (Figure 3.4(b)), by setting the value-supplier of **aspect-ratio** to **:guess**; recall that the attribute's value is unaffected. The attribute is also removed from constraint C–1's list of computed parameters. Furthermore, because the value-supplier of **aspect-ratio** has changed from constraint C–1 to **:guess**, however, additional constraint retraction is required.

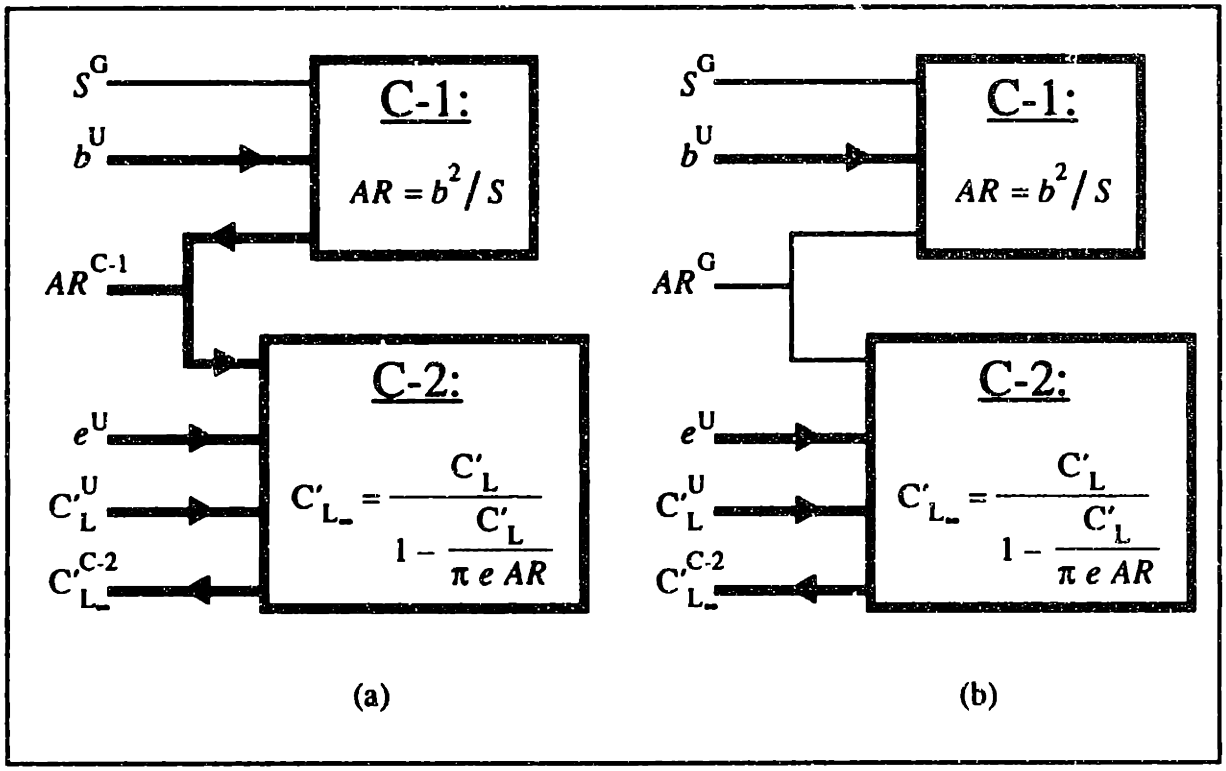

Figure 3.4: Constraint propagation of constraints C–1 and C–2 (a) before rectraction of C–1, and (b) after retraction of C–1.

Attribute **aspect-ratio** is a parameter of both constraints. At this point, however, only one of them, constraint C–2, has any computed parameters. Examination of its *computed-parameters* instance variable reveals that the value-supplier of attribute **cl-alpha_inf** must be retracted. Its value-supplier is therefore set to **:guess** (see Figure 3.5(b)), and it is removed from C–2's list of computed parameters (which will now be empty). Since none of the constraints involving attribute **cl-alpha_inf** have any computed parameters, constraint retraction has been completed.

3.3 Constraint Inversion

3.3.1 Computation of Attribute Values

In the preceding section, little discussion was devoted to the means by which constraints are actually used to compute values for the attributes whose values they relate. In some forms of constraint propagation, such as substitution and transformation systems (see Chapter 1), the process by which constraints are propagated and the process by which parameters are computed are interwoven; indeed, they are the exact same process. Alternatively, however, the approach adopted here—local propagation based upon value-suppliers—has the advantage that these two processes are completely independent. The means by which attribute

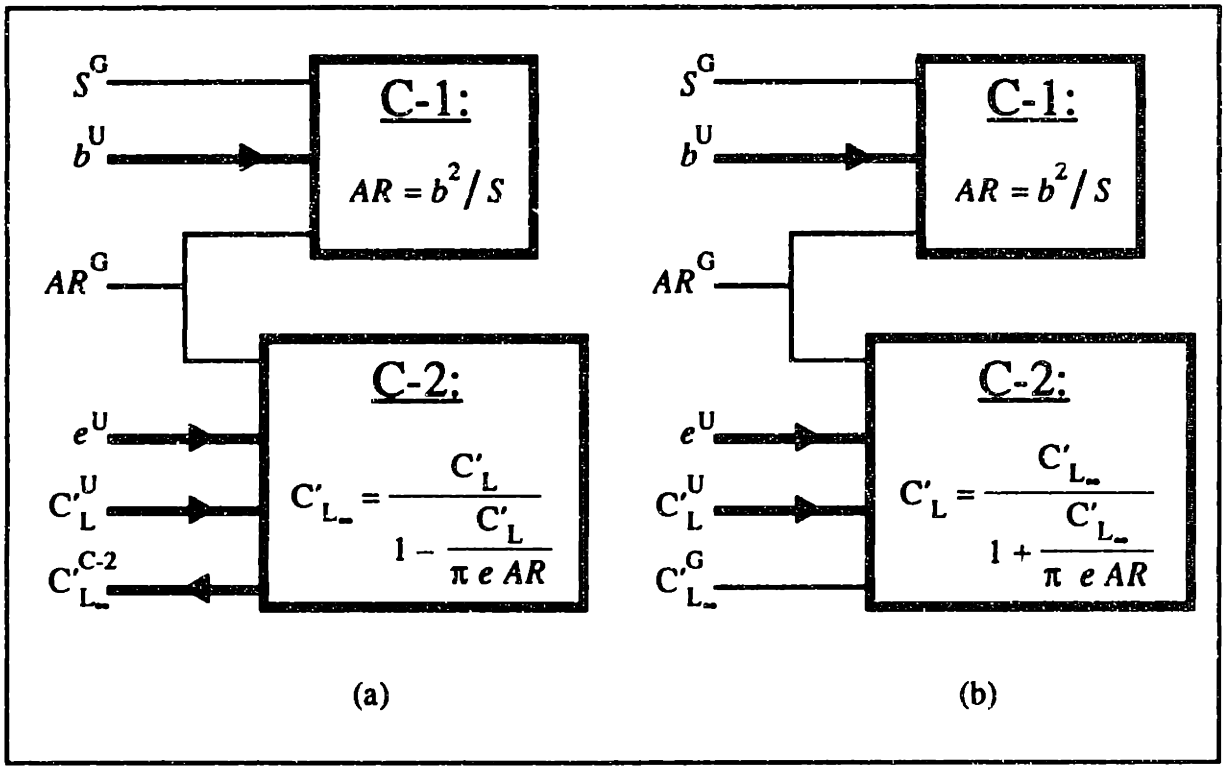

Figure 3.5: Constraint propagation of constraints C–1 and C–2 (a) before retraction of C–2, and (b) after retraction of C–2.

values are calculated using individual constraints has no bearing on the constraint propagation mechanism, which is therefore only responsible for selecting which attributes are to be computed by the constraints, and the order in which these calculations should occur.

When constraint propagation has determined that a constraint is to be used to compute a value for its output parameters, it is a straightforward process to use the LISP function stored as the *compiled-function* instance variable of the constraint to compute the desired values, using the current values of its input parameters as the arguments to this function. As indicated in Table 3.2, this function is based on the *body* specification provided by the constraint's **defconstraint** definition (see Chapter 2). It also performs the appropriate unit conversions on the input and output values.[1]

If it is decided that a constraint is to be used to compute a value for one of its input parameters, then it is necessary to invert the constraint.[2] As indicated above, since the means by which parameters are computed is divorced from the actual propagation mechanism, a

[1]Note that the values of the input parameters must be converted from *Rubber Airplane* internal units to the constraint's units; the output value(s) must be converted to *Rubber Airplane* internal units from constraint units.

[2]Recall that *Rubber Airplane* requires all invertible constraints to be of degree unity. Since all multiple-output constraints must therefore be uninvertible, it is never necessary to solve for multiple input parameters simultaneously. Note that numerical techniques for inverting multiple-output constraints have been implemented for *Paper Airplane*; see Reference [21].

multitude of approaches are possible. Transformation of individual constraint bodies, or other symbolic algebra techniques, could be employed. Based on simplicity of implementation and past experience with the *Paper Airplane* program, however, numerical inversion based upon the Newton-Raphson method was selected.

3.3.2 Numerical Inversion

The advantage of numerical techniques is their generality. Using numerical techniques, the set of mathematical operations performed by a constraint may be treated as a "black box". Rule-based and symbolic algebra techniques may limit the types of operations which may be included in the body of a constraint, based upon the completeness of the available transformation rules. Furthermore, it is not unusual in engineering design that constraints include such operations as numerical integration or table-lookup which cannot be represented analytically; symbolic inversion techniques are inadequate in such cases. Since numerical techniques rely solely upon observation of the output values of a constraint corresponding to a set of input values, such restrictions do not apply.[3]

The specific numerical method adopted for *Rubber Airplane* is the Newton-Raphson technique, used for finding the zeroes of a function. Given a function, $f(x)$, the Newton-Raphson technique may be used to find those values of x for which for which $|f(x)| < \epsilon$ (i.e., $f(x) \approx 0$), where ϵ is some arbitrarily small number indicating the desired precision for the numerical analysis. The process by which such values for x are computed is as follows:

1. Select an initial **seed value** for x, referred to as x_i, which is presumed to be close in value to one of the zeroes of $f(x)$.

2. Compute $f(x_i)$ and $f'(x_i)$. The derivative is typically calculated numerically, using a formula such as:
$$f'(x_i) \approx \frac{f(x_i + \Delta x) - f(x_i - \Delta x)}{2\Delta x}$$

3. If $|f(x_i)| < \epsilon$, then x_i is the desired zero.

4. If $|f(x_i)| \geq \epsilon$, then a new value for x is chosen, x_{i+1}, corresponding to the point where the tangent to $f(x)$ at x_i intersects the x-axis, i.e.,
$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}$$

5. Return to Step 2, repeating the process, substituting x_{i+1} for x_i.

This procedure is illustrated graphically in Figure 3.6.

[3]Of course, numerical techniques do not perform well when constraint parameters may take on only discrete values. It is for this reason that *Rubber Airplane* requires that all constraints involving discrete attributes be uninvertible.

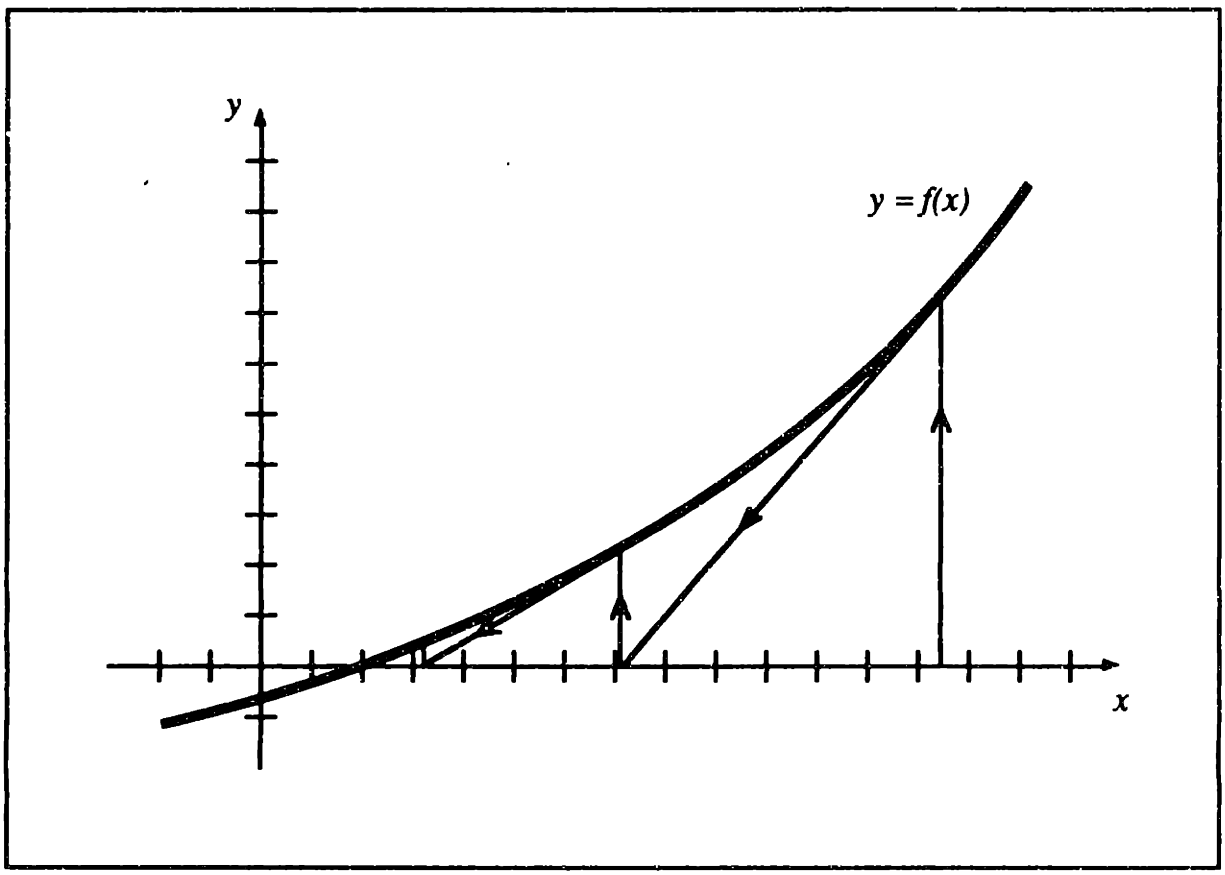

Figure 3.6: The Newton-Raphson technique for numerically locating the zeroes of a function.

To actually use the Newton-Raphson method to simulate constraint inversion, it is necessary to transform constraints into normalized form. *Rubber Airplane* constraints are defined using the **defconstraint** macro (see Chapter 2) in the form $y = f(x_1, x_2, \ldots)$, where y represents the output parameter and $x_1, x_2, \ldots$ represent the input parameters. The corresponding normalized form is

$$F(y, x_1, x_2, \ldots) = y - f(x_1, x_2, \ldots)$$

It is rather straightforward to modify the LISP code given in the *body* of a **defconstraint** specification in order to produce a LISP function which implements the normalized form of the corresponding constraint. This function, when compiled, is stored as the *compiled-normal-function* of each instance of the constraint; when constraint propagation requires solution of the constraint for one of its input variables, the Newton-Raphson method is applied to this function, holding the values of all other attributes constants. Thus, to calculate a value for one of the input variables, $x_k \in x_1, x_2, \ldots$, the Newton-Raphson method is employed to find roots of the function $g(x_k) = F(y, x_1, x_2, \ldots)$

As mentioned above, however, use of the Newton-Raphson method requires selection of an initial seed value, for the first iteration of the algorithm. In order to determine an appropriate seed value, *Rubber Airplane* first generates a list of candidate seed values for the desired input parameter. The normal form of the constraint is evaluated for each of the candidate seed values; the value which yields the result of least magnitude (i.e., the result closest to zero) is selected as the initial seed value for the numerical analysis. The list of candidate seed values consists of the following:

- the current value of the attribute which is to be computed;
- its order-of-magnitude value: and
- a logarithmic distribution (see Section 3.3.3) between the low-value and high-value specifications associated with the attribute.

Note that, while examining this list, should it be determined that one of the candidate seed values is itself a zero (i.e., should the corresponding value of the normal form satisfy the appropriate convergence criterion), it is immediately returned as the computed value for the attribute; Newton-Raphson iteration is not required for such cases.

3.3.3 Logarithmic Distribution between Attribute Bounds

As indicated in the preceding section, selection of the initial seed value for the Newton-Raphson analysis requires generation of a list of candidates. Included among these candidates is a logarithmic distribution of values based upon the low-value and high-value bounds associated with the attribute whose value is to be computed.

To logarithmically divide an interval between two limits, a_l and a_u, into n subdivisions, the lower limit, a_l, is successively multiplied by the factor,

$$e^{\frac{(\log a_u - \log a_l)}{n}}$$

until the value of the upper limit, a_u, is reached (i.e., n times). Thus, the logarithmic distribution between these two limits may be written

$$ld_n(a_l, a_u) = \left\{ a_l e^{\frac{(\log a_u - \log a_l)}{n}}, a_l e^{\frac{2(\log a_u - \log a_l)}{n}}, a_l e^{\frac{3(\log a_u - \log a_l)}{n}}, \ldots, a_l e^{\frac{(n-1)(\log a_u - \log a_l)}{n}}, a_u \right\}$$

where the ith element in the logarithmic distribution is given by

$$a_{i,n} = a_l e^{\frac{i(\log a_u - \log a_l)}{n}}$$

The reason logarithmic distribution was chosen for generating candidate Newton-Raphson seed values is because, for large intervals, a logarithmic distribution divides the interval according to the orders of magnitude included in the interval (i.e., *logarithmically*), while for small intervals, the interval is divided approximately linearly. Thus, the four-interval logarithmic distribution between 1 and 10,000 is

$$\{1,\ 10,\ 100,\ 1{,}000,\ 10{,}000\}$$

while the four-interval logarithmic distribution between 1 and 10 is

$$\{1,\ 1.7783,\ 3.1623,\ 5.6234,\ 10\}$$

and the four-interval logarithmic distribution between 1 and 2 is

$$\{1,\ 1.1892,\ 1.4142,\ 1.6818,\ 2\}$$

This feature has the advantage that, when there is a large disparity between the upper and lower bounds, a more suitable set of test values is generated. In such cases, a linear distribution would be likely to result in a set of candidate values which all have the same order of magnitude; for example, the four-interval linear distribution between 1 and 10,000 is

$$\{1,\ 2500.75,\ 5000.5,\ 7500.25,\ 10{,}000\}$$

Note that all the intermediate values are of order of magnitude $O(10^3)$. If the actual zero is near one of the two bounds, it may then be the case that none of the generated candidates is a suitable seed value. The logarithmic distribution provides a more diverse set of candidates

under such circumstances, and still yields a reasonable range of values when the bounds are close to one another in value.

Furthermore, note that the mathematical description of the elements of a logarithmic distribution presented above provides for the extension of the logarithmic distribution beyond the given upper and lower limits. As indicated above, the elements of an n-interval logarithmic distribution are computed by evaluating the expression

$$a_{i,n} = a_l e^{\frac{i(\log a_u - \log a_l)}{n}}$$

for integral values of i between zero and n (inclusive). By choosing integral values of i which are either less than zero or greater than n, the distribution may be extended beyond the prescribed upper and lower limits, a_l and a_u. Thus, for a six-interval distribution which is extended once beyond each of its bounds, $n = 4$ and i varies from -1 to 5. If the lower and upper bounds are 1 and 10,000, respectively, the values in such an extended distribution would be

$$\{0.1,\ 1,\ 10,\ 100,\ 1,000,\ 10,000,\ 100,000\}$$

The same distribution with lower and upper bounds of 1 and 10, respectively, is

$$\{0.5623,\ 1,\ 1.7783,\ 3.1623,\ 5.6234,\ 10,\ 17.783\}$$

The actual distribution chosen for the generation of candidate seed values is a ten-interval logarithmic distribution which is twice-extended beyond the upper and lower bounds specified for the corresponding attribute. For such a distribution, $n = 6$ and i varies from -2 to 8.

3.4 Computational Loops

3.4.1 Overview

As discussed above, local propagation is inadequate for solving constraint networks which involve cycles. For this reason, some means of recognizing constraints which must be solved simultaneously is required. Local propagation must be augmented by some form of global analysis for the detection and solution of simultaneous equations. At the same time, this global analysis must be compatible with the local propagation techniques already implemented. This means that individual constraints must be treated as indivisible "black boxes"; analytical approaches such as symbolic algebra and network transformation are not possible. Global analysis should also be performed interactively, and its operation should be transparent to the user.

The first step taken in attempting to meet these requirements was to adopt a heuristic first employed by *Paper Airplane*, which assumes that, in engineering design, simultaneous

equations almost exclusively appear in the form of **computational loops**. This term is here meant to imply that such simultaneous equations can, without resort to any other form of algebraic manipulation besides substitution, be reduced to at most two equations in two unknowns.[4] It could alternatively be asserted that such systems of equations are characterized by the property that, by positing an assumed value for just one of the unknowns, values for all of the remaining unknowns are readily calculable, without needing to solve an additional set of simultaneous equations.

This heuristic was inspired by the observation that most conventional computer programs for aircraft design include a primary iteration loop which computes the vehicle's gross weight by first *assuming* a value for it. This assumed value is then used to compute the weights of the various components of the vehicle, which are then summed in order to determine a revised gross-weight estimate. This estimate is used to recompute the component weights, which are once more summed to calculate a new value for the gross weight. The process is repeated until convergence upon a stable value for the gross weight is observed. In this manner, the set of simultaneous equations governing the component and gross weights is solved by constructing a computation loop based on an assumed value for the gross weight.

Thus, rather than search for all possible cycles within a constraint network, attention is focused on identifying the computational loops. If the assumption above is valid, this approach will be adequate for handling the majority of simultaneous equations encountered in solving engineering design problems. The means by which such computational loops are detected, and the techniques used for solving them, are discussed in the following sections. Note that, due to reliance on numerical algorithms for solving constraint loops, only invertible constraints, involving only scalar attributes, are eligible for use as elements of a computational loop.

3.4.2 Loop Detection

Loop Construction

The first step in detecting computational loops is recognition of the individual constraints which may be combined to form such loops. Once such constraints have been identified, actual loop construction may be attempted. As discussed above, however, most of the simultaneous equations present in a constraint network can be solved using local propagation only. In order to avoid conflict with the local propagation algorithm, then, it is necessary to delay all loop construction attempts until local propagation has been completed.

For this reason, rather than immediately attempting loop construction once a candidate

[4] Note that if a system can be reduced in this manner to but a single equation in one unknown, then such a system can be solved using the local propagation technique presented above.

Object Type	Instance Variable	Description
Scalar Attribute	*loop-supplier*	Indicates the assumed source for the attribute's value in the loop currently under construction. Should be either a constraint (indicating that the value may be computed using the constraint), the keyword `:assumed` (indicating that a known value for the attribute has been assumed in order to construct the loop), or the symbol `nil` (indicating that, lacking any other source, the attribute's value-supplier should also serve as its loop-supplier).
Invertible Constraint	*in-loop?*	Stores a boolean flag, either `t` ("true") or `nil` ("false"), indicating whether or not the constraint has been used as a loop-supplier in the loop currently under construction.

Table 3.3: Additional attribute and constraint instance variables used in loop propagation.

loop constraint has been found, the constraint is instead added to queue of candidate loop constraints, to be processed only after local propagation has concluded. At this point, **loop propagation** may be attempted, by examining each of the queued constraints in turn. In attempting loop construction, the first step is re-examination of the candidate constraint. This is because, between the queuing of the constraint and the actual attempt at loop construction, local propagation may have eventually caused this constraint to become perfectly constrained, and applied it to compute the value of one of its attributes. Thus, if the first constraint in the queue is observed to have any attributes listed in its *computed-parameters* instance variable, the constraint is removed from the queue, and the next candidate constraint is examined for computed parameters. It may be the case that all the constraints in the queue will be rejected in this manner. If, however, a candidate constraint with no computed parameters is found, the next step is selection of a suitable **loop variable**. This loop variable will be an attribute whose value is to be assumed known. Thus, the loop variable plays the same role as the gross-weight variable in the iteration loop discussed in the preceding section. Based on this assumed value, calculation of values for other attributes may become possible, which in turn enable calculation of a new value for the loop variable, thus closing the loop. Note that, to support loop propagation, two more attribute and constraint instance variables are required, in addition to those indicated in Tables 3.1 and 3.2. These additional instance variables are listed in Table 3.3.

As with local propagation, it is not necessary to deal with actual attribute values while attempting loop propagation; instead, it is once more possible to divorce loop calculation from loop construction through the use of **loop-suppliers**, which play an analogous role

to the value-suppliers employed in local propagation. The loop-supplier of an attribute can have one of three possible values:

- the keyword "**:assumed**", indicating that the attribute has been selected as the loop variable;
- a constraint, implying that the attribute's value can be computed using that constraint, assuming a known value for the loop variable; or
- the LISP symbol **nil** indicating that no loop-supplier has otherwise been assigned to the attribute, and that the current value of its value-supplier (which may be either a constraint, **:user**, or **:guess**) should be used as the attribute's loop-supplier.

Use of the loop-supplier enables loop construction to be performed using the algorithm originally developed for local propagation. Before loop construction begins, all attributes' loop-suppliers are **nil**, as are the *in-loop?* instance variables of all constraints. A loop variable is then chosen, by selecting the free parameter of the candidate loop constraint which has the most **loop-eligible** constraints. A constraint is said to be **loop-eligible** as long as its *computed-parameters* instance variable is the empty list (i.e., the constraint has not already been used during local propagation) and its *in-loop?* instance variable is **nil** (i.e., the constraint has not already been used during the current attempt at loop construction). By choosing the attribute with the largest number of loop-eligible constraints, loop construction is accelerated, since the largest number of constraints become immediate candidates for addition to the loop about to be built (i.e., the largest number of constraints will have their degrees of freedom—with respect to loop construction—reduced by one).

Consider, for example, the following set of constraints representing a simplified aircraft gross weight calculation:

$$\begin{array}{lrcl} \text{C–0:} & W_1 & = & f_1(W_T) \\ \text{C–1:} & W_2 & = & f_2(W_T) \\ \text{C–2:} & W_3 & = & f_3(W_T) \\ \text{C–3:} & W_T & = & W_1 + W_2 + W_3 \end{array}$$

where W_T represents the gross weight, and W_1, W_2, and W_3 represent three component weights. The first three equations relate individual component weights to the total vehicle weight, while the fourth relationship, C–3, computes gross weight as the sum of the component weights. Assume that the four parameters have yet to be calculated by other means (i.e., all have a value-supplier of **:guess**), and that each of the four equations represents a loop-eligible constraint. If the first constraint, C–0, is queued as a candidate loop constraint, the corresponding candidate loop variables will be W_1 and W_T. Parameter W_1 has two loop-eligible constraints (C–0 and C–3), but parameter W_T, being a free parameter of

all four of the indicated constraints, has four. Thus, W_T is the best choice for loop variable for this set of simultaneous equations, since constraints C–0, C–1, and C–2 all become immediately available for local propagation of loop-suppliers once the loop-supplier of W_T is set to `:assumed`. Had W_1 been selected as the loop variable, only constraint C–0 would be available for local propagation of loop-suppliers.

Note, though that the success of this loop-detection algorithm does not depend upon the choice of loop variable: if a computational loop[5] may be constructed which includes the candidate loop constraint, the algorithm will find this loop, regardless of the choice of loop variable. The criterion presented above has been selected only to accelerate loop propagation. Furthermore, note that this algorithm for loop detection is also insensitive to the choice of loop constraint. If a computational loop exists, this algorithm would succeed in constructing it given any of the included constraints as an initial loop constraint.

Once a loop variable is chosen, it is assigned a loop-supplier of `:assumed`, and local propagation of loop-suppliers is performed. Each of the loop variable's constraints is examined to identify those which are loop-eligible. If any such loop-eligible constraint is then observed to be perfectly constrained *according to the loop-suppliers of its attributes*, the constraint is added to the loop currently under construction: its *in-loop?* instance variable is set to `t` (i.e., "true"), and the constraint becomes the loop-supplier of its free parameter. (Since only invertible constraints are eligible for loop propagation, and all invertible constraints are of degree unity, such constraints will always have exactly one free parameter when they are perfectly constrained.) The loop-eligible constraints of this free parameter are then examined, and, if any of these are perfectly constrained, loop propagation continues recursively.

There are two circumstances under which local propagation of loop-suppliers may terminate. First, like local propagation of value-suppliers, the process may simply run out of constraints which are, or have become, perfectly constrained. In this case, loop construction has failed, because no constraint has been found to close the loop by enabling computation of the original loop variable, which was assumed known. (In the gross weight analogy introduced above, the iteration loop is closed by the constraint which calculates the gross weight by summing the component weights, which were themselves calculated based on the assumed gross weight.) Loop construction may then be attempted on the next candidate in the queue of potential loop constraints; this process is repeated until either loop construction succeeds, or the queue is exhausted.

Alternatively, however, local propagation of loop-suppliers may encounter a constraint whose *in-loop?* instance variable is `nil`, but has zero free parameters (i.e., based on the loop-

[5]Recall that, in this document, the phrase "computational loop" strictly refers to set of simultaneous equations which is of the form anticipated by the heuristic presented in Section 3.4.1.

suppliers of its attributes). Such a constraint is referred to as a **closing constraint**, because it may be used to close the loop. A constraint which has not been used to compute a value for any of its parameters, but nevertheless has zero free parameters, is overconstrained. Recall, though, that the loop variable was only *assumed* known; this assumption artificially reduced the number of degrees of freedom of the system of equations by one. The resulting discovery of an overconstrained constraint thus indicates the successful discovery of a computational loop within the constraint network. This overconstrained constraint will necessarily have among its parameters either the loop variable, or one or more attributes whose loop-supplier is another constraint (otherwise, it should have earlier been eligible for local propagation based upon value-suppliers).

In the example introduced above, if parameter W_T is selected as the loop variable, local propagation of loop-suppliers—using the first three of the four constraints—may be employed to assign loop suppliers to each of the three component weights. Thus, the loop-supplier for parameter W_1 would be constraint C–0, the loop-supplier for W_2 would be constraint C–1, and the loop-supplier for W_3 would be constraint C–2. Under these circumstances, however, loop-suppliers would have been assigned to all four of the parameters associated with constraint C–3, though the constraint itself remains loop-eligible. Constraint C–3 has thus become overconstrained, and may therefore serve as the closing constraint for this loop, to be used in computing the value of the loop variable, W_T, based on the values of the other three loop parameters.

As might be expected, under most circumstances the original loop variable will indeed be a parameter of the closing constraint. However, because the loop variable is selected from the parameters of but a single constraint (i.e., the constraint found at the head of the queue of candidate loop constraints), it may be the case that the original choice of the loop variable was, in some way, non-optimal. In such cases, the loop variable may not itself be a parameter of the loop's closing constraint. A valid computational loop has indeed been constructed, but—for calculation and propagation purposes—some reorganization is required to correct such "blemished" loops. This is a rather straightforward process, however, once the loop has been constructed.

Once a closing constraint has been identified, then, the required constraints must be collected together to form a computational loop. Loops are themselves represented as lists of pairs, each pair consisting of an attribute and the constraint which will be used to compute it within the loop. The first member of the computational loop will be a pair consisting of the loop variable and the closing constraint, representing the fact that the closing constraint enables the calculation of the original, assumed parameter. Next, all of the parameters of the closing constraint are examined. If any of them have a loop-supplier which is a constraint, a pair consisting of the attribute and its loop-supplier are added to

the beginning of the loop. The parameters of all such loop-supplier constraints are similarly examined, and any additional attribute/loop-supplier pairs are added onto the beginning of the loop. This process continues recursively until all of the dependencies which support the closing constraint have been collected. Note that, because new pairs are always added at the beginning of the loop, the loop's original pair, containing the loop variable and the closing constraint, will actually be the last element of the completed loop structure.

It might seem more efficient to simply keep a record of all the loop-supplier assignments made during loop construction, and use this record to represent the computational loop. As a matter of fact, it is, indeed, necessary to maintain such a record, so that all *loop-supplier* and *in-loop?* instance variables can be reset after loop construction has been attempted, in preparation for subsequent loop construction efforts. However, because the local propagation algorithm presented here performs a depth-first search through the constraint network, it turns out that there may be some search branches which do not actually contribute to the closing of the loop. While performing the calculations needed to solve the loop, these branches may therefore be ignored. By explicitly collecting only those attribute/loop-supplier pairs upon which the closing constraint is dependent, these "dead-end" branches are pruned from the final computational loop. Computational loops are thus reduced to minimum size, improving both the performance and the stability of the numerical algorithms used to solve them (see Section 3.4.3). Note that, since loop construction employs the same search mechanism as normal local propagation of constraints, it will be the case that, once the corresponding loop is solved, these extraneous branches may be solved using local propagation alone.

As indicated above, after the computational loop has been constructed, it becomes possible to repair those loops for which the loop variable is not a parameter of the closing constraint. If such cases, it is a straightforward matter to examine the resulting loop to determine which of its parameters *should* have been chosen as the loop variable, simply by determining which of the attributes appears most frequently as a parameter of all of the loop's constraints. According to the heuristic which forms the basis of this loop detection scheme, this is the attribute which should be chosen as the loop variable. The loop can then be reconstructed based on the new loop variable. In practice, it has been observed that this new loop variable will always be a parameter of the new loop's closing constraint.

Note that since this computation can be executed very efficiently; for convenience the implementation has chosen to perform it on all newly constructed loops, regardless of whether or not the loop variable is a parameter of the closing constraint. If the loop variable selected by this procedure does not match the original loop variable, a new computational loop is constructed. Under certain circumstances, this new loop can actually have fewer elements than the original, which, as indicated above, improves the performance of the numerical

algorithms employed to actually solve computational loops.[6]

After construction of an acceptable computational loop has been completed, the appropriate loop solution algorithm (see Section 3.4.3) may then applied to compute values for its attributes, using the corresponding constraints. Numerical techniques are employed, and thus iteration is required. Once these values have been computed, it is necessary to assign the appropriate values and value-suppliers to the attributes. For convenience, a two-step process is employed. First, all of the values are assigned, without modifying the attributes' value-suppliers, which will all be **:guess**. Then, in order to forestall future attempts to propagate the closing constraint, the loop variable is added to the list of computed parameters maintained by the *computed-parameters* instance variable of the closing constraint, which previously should have been empty. Next, the closing constraint is assigned as the value-supplier for the loop variable (this is why it is necessary that the loop variable be an actual parameter of the closing constraint). This change in the value-supplier of the loop variable will automatically invoke local propagation. As a result, the appropriate constraints will be assigned as the value-suppliers of the attributes with which they were associated within the computational loop. (This is true because the algorithm used for local propagation of value-suppliers is identical to the algorithm used for local propagation of loop-suppliers while constructing the loop.) Also, because the correct values have already been assigned to these attributes, and current values are always examined while performing local-propagation calculations (see Section 3.2), the computations resulting from this local propagation will incur only a small performance penalty. Furthermore, because the loop variable is a parameter of the closing constraint and local propagation is used to assign the other value-suppliers, this approach has the additional advantage that there is no need to alter the process by which constraint propagations are retracted: the retraction process detailed in Section 3.2 applies equally well to loop-based constraint calculations.

It will also be the case that any secondary calculations made possible by the solution of the loop will also be performed during the course of this local propagation. Then, once local propagation has been completed, the queue of candidate loop constraints will once more be examined, allowing further loop construction to be attempted. Note that, while this approach does not allow for the nesting of computational loops, the fact that the queue is re-examined after each successful loop propagation means that later computational loops can be based upon the calculations performed by earlier computational loops.

The process by which computational loops are detected and constructed may thus be summarized as follows:

1. Initially, the *loop-supplier* instance variable of all attributes is `nil`, as is the *in-loop?*

[6]Specifically, this will be the case if the original loop constraint was actually on an extraneous or "dead-end" branch, as described above.

instance variable of all constraints.

2. During local propagation, identify candidate loop constraints (see Section 3.4.2) and add them to a queue of such constraints. Candidate constraints should only be added to the queue once; there is no need for duplicate appearances of constraints within the queue. After local propagation has been completed, proceed to Step 3.

3. Remove the first constraint from the queue. If this constraint is loop-eligible (i.e., the value of its *computed-parameters* instance variable is the empty list and the value of its *in-loop?* instance variable is `nil`), proceed to Step 4; if not, repeat this step.

4. Count the loop-eligible constraints associated with each of the constraint's free parameters. The free parameter with the most loop-eligible constraints is selected as the loop variable.

5. Set the loop-supplier of the loop variable to `:assumed`. Add the attribute and its loop-supplier to an on-going record of loop-supplier assignments.

6. Examine all of invertible equality constraints associated with this attribute. For each such constraint, based upon the loop-suppliers associated with its parameters,

   - If the constraint is loop-eligible but has no free parameters, proceed to Step 9.
   - If the constraint is loop-eligible and has exactly one free parameter, then
     (a) Set the *in-loop?* instance variable of the constraint to `t`.
     (b) Set the loop supplier of the attribute which is the constraint's free parameter to be the constraint. Add the attribute and the constraint to the record of loop-supplier assignments.
     (c) Repeat Step 6 for the attribute identified as the constraint's free parameter.

   Note that, due to this presence of Step 6c, this process is potentially recursive.

7. No computational loop can be constructed based on the selected loop variable. For each attribute mentioned in the record of loop-supplier assignments, set its *loop-supplier* instance variable to `nil`. For each constraint mentioned in the record of loop-supplier assignments, set its *in-loop?* instance variable to `nil`.

8. Return to Step 3.

9. This loop-eligible constraint with zero free parameters is the closing constraint. Construct a computational loop as follows:

   (a) The first pair placed in the loop consists of the loop variable and the closing constraint.

(b) For this constraint, examine the loop supplier of each of its parameters. If its loop-supplier is a constraint,

i. Construct a pair consisting of the parameter (an attribute) and its loop supplier (a constraint).

ii. Add this pair to the beginning of the current computational loop structure.

iii. Repeat Step 9b for the constraint which serves as the parameter's loop-supplier.

Note that this, too, is a recursive process.

10. For each attribute mentioned in the record of loop-supplier assignments, set its *loop-supplier* instance variable to `nil`. For each constraint mentioned in the record of loop-supplier assignments, set its *in-loop?* instance variable to `nil`.

11. Examine each of the attributes in the computational loop, determining the frequency with which each appears as a parameter of the loop's constraints. If the attribute which appears most frequently is not the loop variable for the current loop, return to Step 5, using this attribute as the loop variable for a second round of loop construction. Note that if a new loop is deemed necessary, this test does not have to be repeated for the new loop.

12. Solve the computational loop in order to compute values for the attributes in the loop.

13. Assign these values to the attributes, without changing their value-suppliers.

14. Add the loop variable to the closing constraint's list of computed parameters (which previously should have been empty).

15. Assign the closing constraint as the value-supplier for the loop variable, thus triggering local propagation and returning to Step 2.

As indicated above, there is no need to modify or supplement the algorithm presented in Section 3.2 for performing constraint retractions to account for the presence and solution of computational loops. In Section 3.4.3 of this chapter, however, certain minor modifications to this loop-propagation algorithm are suggested, which enable more effective use of iteration loops in solving for the attributes of computational loops; see Section 3.4.3 for details.

Selection of Candidate Loop Constraints

In the description of the loop detection algorithm presented above, the means by which candidate loop constraints are selected for addition to the queue was not described. Dur-

ing the course of this research, a number of different approaches were investigated before deciding upon the current selection strategy.

One common factor among these approaches, however, was the modification of the original local propagation algorithm to incorporate the selection process. Note in the algorithm presented in Section 3.2 that local propagation is initiated whenever the value-supplier of an attribute is changed. The next step is to count the number of free parameters associated with each of the attribute's constraints, to determine which, if any, have become perfectly constrained. Because the count of a constraint's free parameters is an important factor in assessing a constraint's utility as a loop constraint, it is at this point that the selection process takes place. Whenever an attribute's constraints are examined for possible local propagation, those which are not found to be perfectly constrained are also examined for possible loop propagation. Any of an attribute's invertible equality constraints which pass the selection criterion are added to the queue of candidate loop constraints, to be processed after local propagation has concluded. Note that this approach is particularly efficient because propagation, both local and loop-based, is only enabled by changes in attribute value-suppliers. Testing for loop constraints as part of the local propagation process ensures that the minimum subset of constraints is examined.

Recall that the loop detection algorithm is based on assuming a loop-supplier for the chosen loop variable. Local propagation of loop-suppliers is then performed, in order to find a closing constraint. In order for loop propagation to succeed at all, it is thus necessary that at least one of the loop variable's constraints have exactly two free parameters—the loop variable itself, as well as one other attribute. Otherwise, local propagation of loop-suppliers will fail immediately, since none of the loop variable's constraints will become perfectly constrained upon assignment of the loop variable's loop-supplier.

For this reason, the first approach taken was to queue only those constraints which were observed to have exactly two free parameters. By selecting a loop variable from the parameters of such constraints, at least the first stage of loop propagation is guaranteed to succeed. Of course, due to the nature of constraint propagation, the loop may ultimately include constraints which have more than two free parameters. Nevertheless, as explained above, the loop must include at least one constraint which, after local propagation, had exactly two free parameters.

For most situations, this strategy proved adequate. Two mitigating factors must be recalled, however:

- When attempting to construct a computational loop, if no loop can be developed from the first constraint on the queue, it is removed from the queue.

- Loops can include constraints which, after local propagation, still had more than two

free parameters.

Consider the case in which, after local propagation has been completed, a particular constraint has four free parameters. Loop propagation is then attempted, based on some candidate loop constraint, which, based on the above selection criterion, must have had two free parameters after local propagation was completed. Assume that, during the course of loop propagation, propagation of loop-suppliers encounters this constraint which has four free parameters, but loop-suppliers have been assigned for only two of them, so that loop propagation cannot use this constraint. Furthermore, assume that this attempt at loop propagation fails, but, had the constraint in question had only three free parameters, the loop could have included this constraint and would have succeeded. Finally, assume also that the fourth free parameter of this constraint has no other constraints.

Thus, if the user were to assign a value to this fourth parameter, its value-supplier (and, effectively, its loop-supplier) would be set to `:user`, and the loop propagation which previously failed would now succeed. Because it failed, however, the original loop constraint has been removed from the queue. And, even if the required user-supplied value were to be assigned, loop propagation would still not succeed, because no loop constraint would be available. The only constraint associated with this attribute, the constraint which previously had four free parameters, still has three free parameters. It would not pass the selection criterion. Since there are no other constraints associated with this attribute, no new constraints would be added to the queue. No means are available for detecting the computational loop which has now been created; the algorithm has failed.

The first attempt at correcting this deficiency involved a slight alteration to the loop propagation algorithm: instead of using a single queue for storing candidate loop constraints, two were provided. The original selection criterion was retained; candidates detected during local propagation were added to the first queue. After termination of local propagation, the loop propagation algorithm proceeded as above, by attempting to construct loops using successive elements of the first constraint queue. Rather than discard those constraints for which loops could not be constructed, however, such constraints were added to the second queue. After loop propagation was completed, the two queues were switched. In this manner, the above problem was avoided by effectively maintaining a permanent queue of all those constraints with exactly two free parameters.

This solution was quickly deemed unworkable, however, because the queue rapidly grew to a prohibitively large size. Loop detection thus became very time-consuming, since it required repeated processing of this large queue. The advantages of linking candidate selection with local propagation were effectively lost, since constraints were being re-examined without regard for changes—or lack thereof—in the value-suppliers of their attributes. If loop propagation were to continue to be performed interactively, an alternative solution was

required.

For this reason, it was decided to modify the selection criterion, rather than the propagation algorithm. Instead of selecting only constraints with exactly two free parameters, all constraints with *two or more* free parameters were deemed eligible as candidate loop constraints. The second queue was removed; loop constraints which failed the first attempt at loop propagation were removed from the queue until changes in their attribute's value-suppliers caused them to be enqueued once more. Because constraints with more than two free parameters could be enqueued, the difficulty associated with such constraints, as described above, is avoided.

Of course, this means that a larger number of constraints will be enqueued during the course of local propagation, which might suggest longer processing times, as was the case with the previous modification. In practice, however, it is observed that the additional processing time is not prohibitive: most of these constraints cannot actually be used to construct a computational loop and, furthermore, the amount of computation required to confirm this, using the algorithm presented in Section 3.4.2, is minimal. The efficiency gained from not continually re-examining constraints—as was required by the original algorithm modification—more than compensates for this small performance penalty.

In addition, this alternative selection criterion enables the use of another means for improving algorithm performance. Observe that all computational loops must necessarily consist only of constraints which have two or more free parameters: a constraint with only one free parameter is eligible for local propagation. Note also that a constraint may only be used in one computational loop, since it may only be used to compute a single parameter. As indicated in Section 3.4.2, any constraint of an actual computational loop could be used as its initial loop constraint. As a corollary to this statement, it may also be observed that if a constraint is used during an unsuccessful attempt at loop propagation, any subsequent attempt to use the constraint during loop construction will also fail, unless some intermediate change in attribute value-suppliers has occurred.

For this reason, when using the new selection criterion, whenever a constraint is used as the loop-supplier of an attribute during loop propagation based on some other loop constraint, this constraint, if present, may be removed from the queue of candidate loop constraints. This is possible because, regardless of the success or failure of the current loop propagation attempt, any subsequent attempts to use this constraint would result in exactly the same loop. There is thus no need to attempt to use this constraint as a loop constraint. If the current loop construction effort succeeds, the constraint will no longer be available. If it fails, then later use of this constraint as a loop constraint will also fail. The only way such an attempt could succeed is if some change in attribute value-suppliers has taken place between the two attempts, which enables the construction of a true computational

loop. This change in value-suppliers, however, must necessarily enqueue at least one of the constraints present in this loop. For this reason, the removal from the queue of those constraints which meet the above description is justified.

Specifically, then, whenever the *in-loop?* instance variable of a constraint is set to `t`, any occurrence of the constraint within the queue of candidate loop constraints may be removed. Not surprisingly, this modification can substantially reduce the number of candidate loop constraints which must be examined. The resulting performance benefits make this approach even more preferable than the original modification to the algorithm; this final revised approach is quite acceptable for use in interactive loop detection.

In summary then, during the course of local propagation, while examining the constraints of an attribute to find those which are perfectly constrained, any invertible equality constraint which has two or more free parameters is enqueued as a candidate loop constraint. Furthermore, Step 6 of the loop propagation algorithm presented in Section 3.4.2 (see page 119) may be altered such that, whenever the *in-loop?* instance variable of a constraint is set to `t`, all occurrences of the constraint within the queue of candidate loop constraints (of which there should be at most one) may be removed.

Example Loop Detection

To illustrate the above process, consider the following three mathematical constraints, expressed in normalized form:

$$\begin{array}{lrcl} \text{C–0:} & E(v,x) & = & 0 \\ \text{C–1:} & F(w,x,y) & = & 0 \\ \text{C–2:} & G(y,z) & = & 0 \\ \text{C–3:} & H(x,z) & = & 0 \end{array}$$

Assume that the value of parameter w has been supplied by the user. Its value-supplier is, therefore, `:user`. All other parameters have a value-supplier of `:guess`. The loop-suppliers of all the parameters is `nil`, indicating that the respective value-suppliers should serve as loop-suppliers, as well. These initial conditions are depicted in Figure 3.7(a), where superscripts are used to denote both value-suppliers and loop-suppliers. The first superscript indicates the value-supplier of a parameter, the second its loop-supplier.

Setting the value-supplier of w to `:user` will trigger local propagation. This parameter has only one constraint, C–1, and it is not perfectly constrained. Since C–1 has multiple free parameters, it is instead added to the queue of possible loop constraints. At this point, local propagation has been completed; loop propagation may be attempted. The first (and only) constraint on the queue is C–1, the parameters of which are examined for selection of a loop variable.

Constraint C–1 has two free parameters, x and y. Parameter x has three loop-eligible constraints, while y has only two, so x is chosen as the loop variable. Its loop-supplier

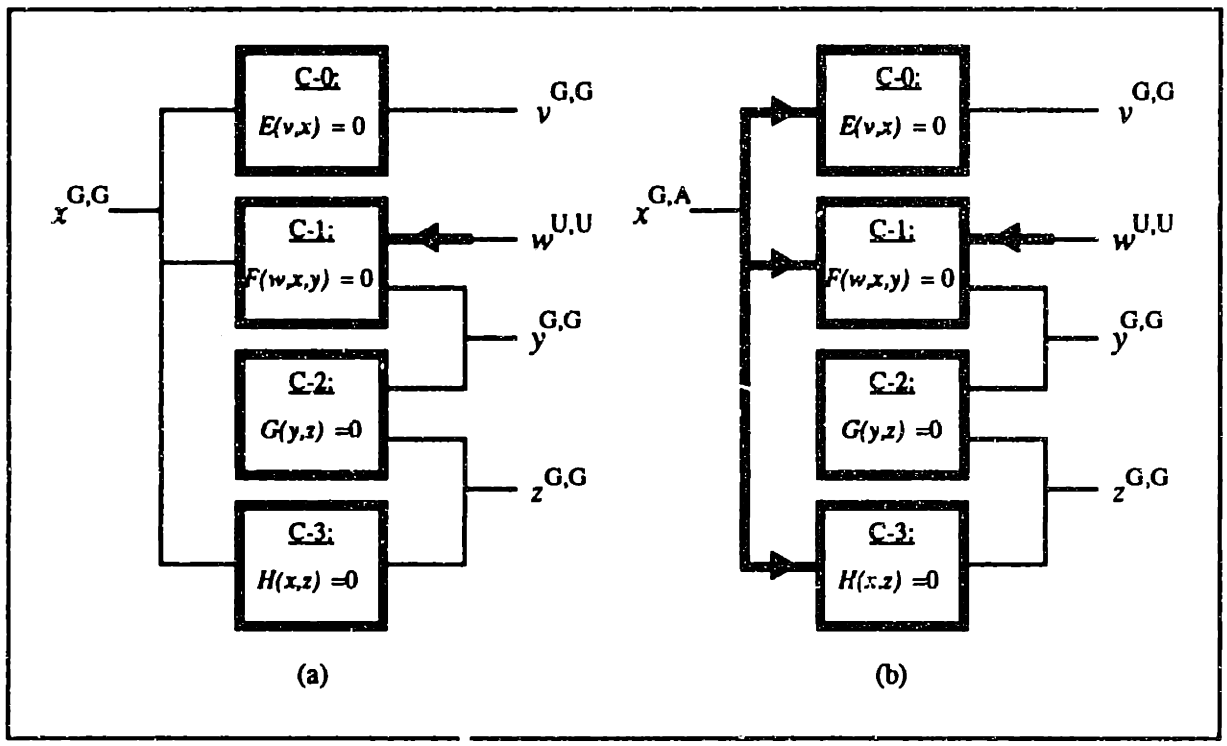

Figure 3.7: Loop propagation of constraints C–0 through C–4 (a) before loop propagation, and (b) after selection of the loop variable.

is set to **:assumed**, as indicated by the superscript "A" in Figure 3.7(a), and this loop-supplier assignment is recorded for later retraction. The next step is to examine each of x's loop-eligible constraints, in turn, for possible propagation of loop-suppliers. The first such constraint is C–0. Only one of its parameters, v, has a loop-supplier of **:guess** (actually, its loop-supplier is **nil**, and its value-supplier is **:guess**), indicating that, for loop propagation purposes, C–0 is perfectly constrained. Its *in-loop?* instance variable is set to **t**; note that, at this point, if C–0 were also in the queue of candidate loop constraints, it would be removed. Also, C–0 is made the loop-supplier of v, and a record is made of this assignment. Since v no longer has any loop-eligible constraints, control is returned to parameter x, for examination of its second loop-eligible constraint, C–1.

Constraint C–1 is also perfectly constrained according to the loop-suppliers of its attributes, and therefore may be used in the loop to compute y. Its *in-loop?* instance variable is set to **t** (constraint C–1 is not present in the constraint queue and therefore need not be removed), constraint C–1 is made the loop-supplier of parameter y, and the loop-supplier assignment is noted in the ongoing record. This is the stage of loop propagation depicted in Figure 3.8(a).

At this point, the other constraints associated with y are examined. Parameter y has only one remaining loop-eligible constraint, C–2. Examination of the loop-suppliers associ-

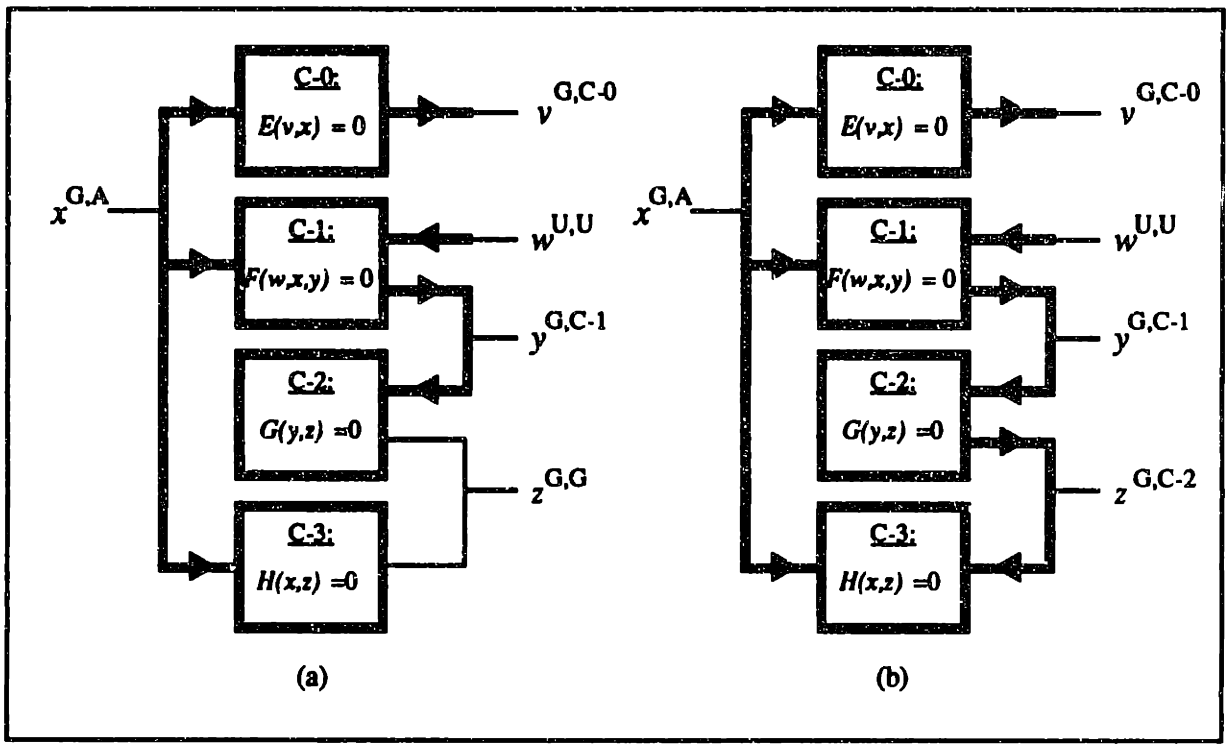

Figure 3.8: Loop propagation of constraints C–0 through C–4 (a) after initial propagation of the loop variable, and (b) after detection of the closing constraint.

ated with the parameters of C–2 reveals that it has one remaining free parameter, z. The *in-loop?* instance variable of constraint C–2 is set to `t`, and C–2 becomes the loop-supplier for parameter z, as indicated in Figure 3.8(b). A record is made of this fourth loop-supplier assignment.

The constraints of parameter z are now examined, and it is observed that z now has one loop-eligible constraint, C–3. Constraint C–3 has two parameters, x and z. The loop-supplier of x is `:assumed`, and the loop-supplier of z is constraint C–2. For the purposes of loop propagation, then, constraint C–3 has no free parameters. It may therefore be used as the closing constraint for the current loop, and construction of the computational loop structure may begin. The first pair added to the computational loop consists of the loop variable and the closing constraint:

$$\{(x, \text{C–3})\}$$

Constraint C–3 has one other parameter, and because its loop-supplier is a constraint, another pair is added to the computational loop:

$$\{(z, \text{C–2}), (x, \text{C–3})\}$$

Next, the parameters of C–2 are examined. Parameter z is already present in the computational loop, but y, whose loop-supplier is the constraint C–1, is not. A pair for y is therefore

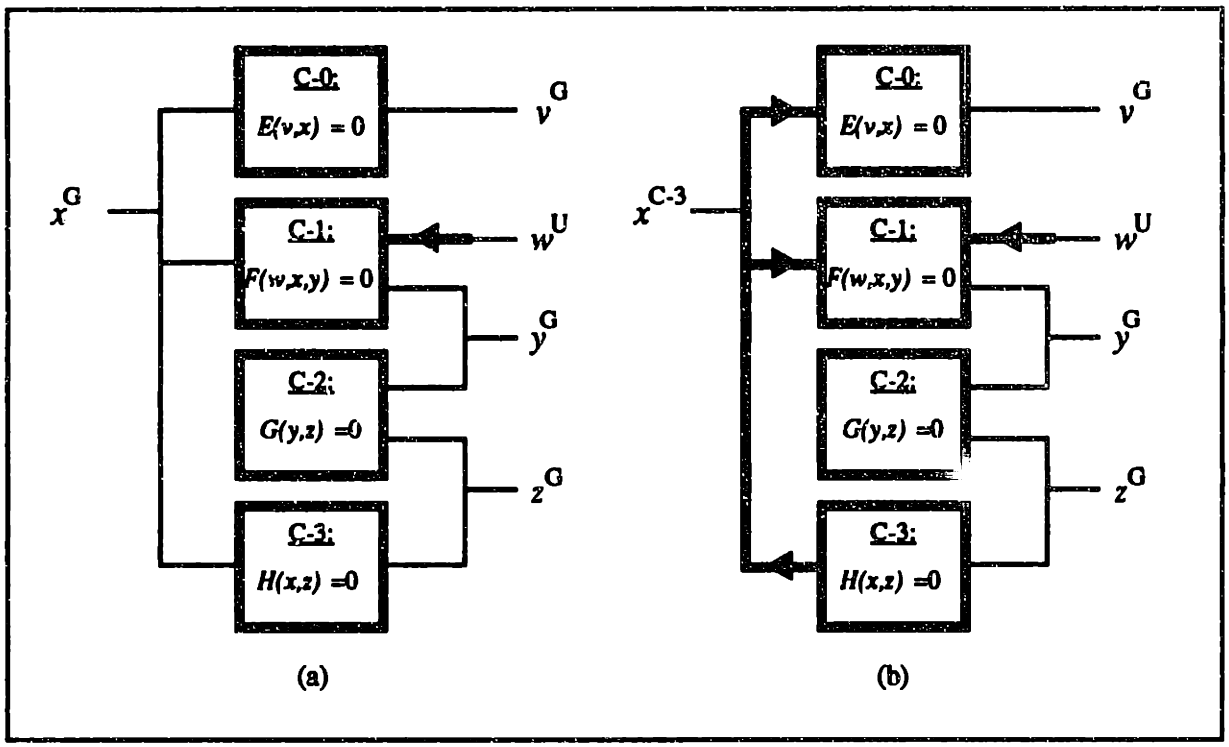

Figure 3.9: Loop propagation of constraints C–0 through C–4 (a) before all local propagation, and (b) after propagation of the closing constraint.

added:

$$\{(y, \text{C–1}), (z, \text{C–2}), (x, \text{C–3})\}$$

The next step is to examine the parameters associated with the newly added constraint, C–1. The loop-supplier of y is a constraint, but y is already present in the computational loop. The loop-supplier of x is `:assumed`, and the loop-supplier of w is `:user`, so no computational loop entries are required for these parameters. The computational loop has been completed. Examination of this loop shows that each of the parameters in the loop is a parameter of two of the loop's constraints. Thus, any of the loop's parameters is a good choice for loop variable, including x. Furthermore, x is, indeed, a parameter of the closing constraint; the computational loop need not be recomputed.

At this point, all loop-supplier assignments may be retracted, by consulting the record which was kept during the loop propagation process. Note that constraint C–0, though it served as a loop-supplier during propagation, was not included in the final computational loop. This is why it was necessary to keep the record of loop-supplier assignments: not all of the necessary retractions can be inferred from the final loop structure.

Next, the computational loop is passed on to the appropriate loop-solving algorithm so that values for the loop's attributes may be computed. These computed values are assigned to the attributes, without modifying their values suppliers, and the loop variable,

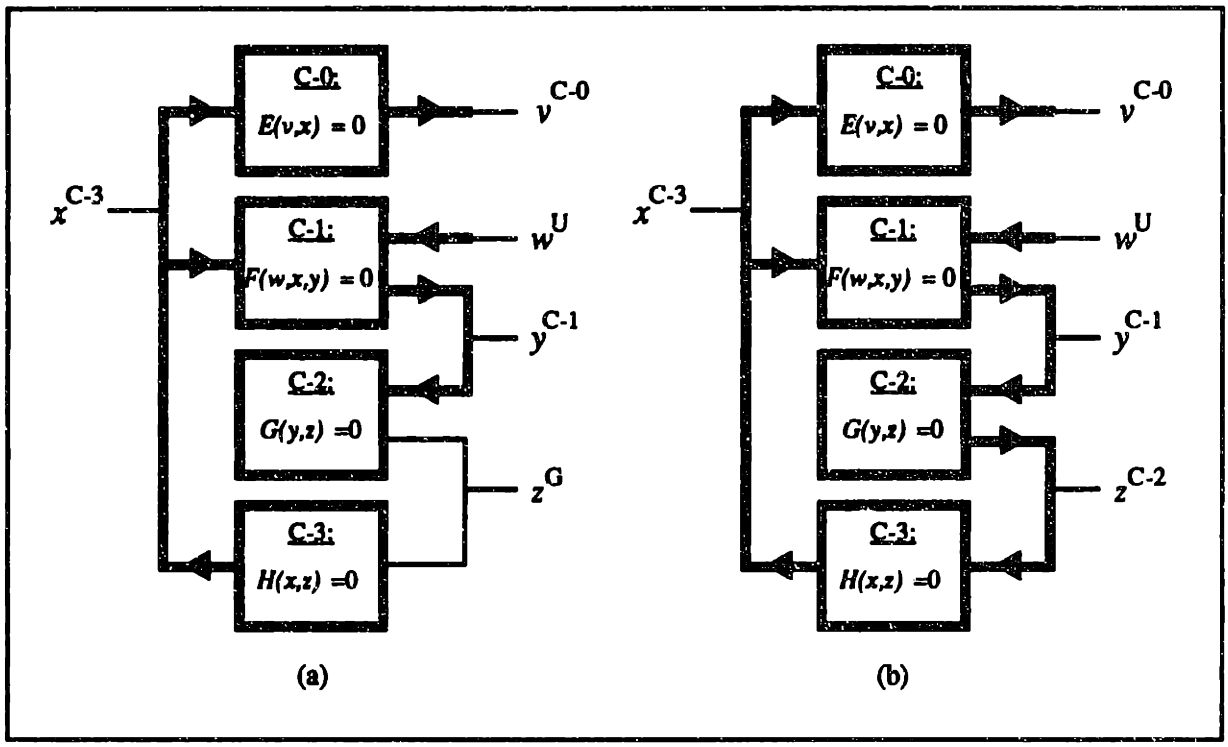

Figure 3.10: Loop propagation of constraints C–0 through C–4 (a) after propagation of constraints C–0 and C–1, and (b) after propagation of constraint C–2.

x, is assigned as the computed parameter of the closing constraint, C–3. Ignoring loop-suppliers now, the current status of the constraint network is as depicted in Figure 3.9(a). Next, constraint C–3 is assigned as the value-supplier for x, as indicated in Figure 3.9(b). This value-supplier assignment causes both C–0 and C–1 to become perfectly constrained, allowing the "computation" of both v and y. Recall that the appropriate value for y, consistent with the parameters and constraints present in the computational loop, has already been assigned to it. Only the value for v need actually be computed, since v was not included in the computational loop. The appropriate value-suppliers are assigned to these two parameters, as indicated in Figure 3.10(a). The *computed-parameters* instance variables of the two constraints are also modified.

At this point, constraint C–2 is observed to be perfectly constrained, and may be used to "compute" z. As depicted in Figure 3.10(b), constraint C–2 is assigned as the value supplier for z. Similarly, z is made a computed parameter of constraint C–2. After this step has been taken, none of the constraints associated with z are observed to be perfectly constrained, and local propagation is concluded. In this manner, the effects of the solution of the computational loop are propagated through the constraint network.

3.4.3 Loop Computation

Overview

Two techniques are available for the solution of computational loops in *Rubber Airplane*: simultaneous Newton-Raphson and a simple iteration loop method. Both methods are numerical and iterative in nature. Initially, only the Newton-Raphson method was supported. Unfortunately, problems with the stability of this method were encountered when attempting to solve particularly large computational loops (e.g., the weight loop of an aircraft design, see Chapter 4). For this reason, a second method, based on construction of a simple iteration loop, was added. The program chooses which method to apply based on the size of the computational loop. Small loops are solved using the simultaneous Newton-Raphson method, and large loops are solved as iteration loops. A threshold parameter is provided, the value of which represents the maximum loop size for which the Newton-Raphson method may be applied. Based upon experimentation with the various test cases (see Chapter 4), this parameter has been set at five. Computational loops involving more than five pairs of attributes and constraints are solved using iteration loops.

Simultaneous Newton-Raphson

The one-dimensional Newton-Raphson method, presented in Section 3.3.2, is a technique for locating the zeroes of a function of one variable. The simultaneous Newton-Raphson method allows for the determination of the points in a multi-dimensional space which represent the zeroes of a set of simultaneous equations.

To describe the method, then, consider a vector of n parameters,

$$\vec{x} = \left\{ \begin{array}{c} x_1 \\ x_2 \\ \vdots \\ x_n \end{array} \right\}$$

and a set of n simultaneous equations governing these parameters:

$$\vec{F} = \left\{ \begin{array}{c} f_1(x_1, x_2, \ldots, x_n) \\ f_2(x_1, x_2, \ldots, x_n) \\ \vdots \\ f_n(x_1, x_2, \ldots, x_n) \end{array} \right\}$$

The simultaneous Newton-Raphson method provides a means for numerically computing a value for $\vec{x}$ for which $\vec{F}(\vec{x}) = \vec{0}$, i.e., a set of values for the parameters $x_1, x_2, \ldots, x_n$ for which the values of all of the governing functions $f_1, f_2, \ldots, f_n$ are simultaneously zero:

$$\begin{aligned} f_1(x_1, x_2, \ldots, x_n) &= 0 \\ f_2(x_1, x_2, \ldots, x_n) &= 0 \\ &\vdots \\ f_n(x_1, x_2, \ldots, x_n) &= 0 \end{aligned}$$

A numerical search through the n-dimensional parameter space, based on partial derivatives and starting from the initial values of the parameters, is employed to find the desired zero. Actually, since this is a numerical method, precise zeroes are not found. Rather, an arbitrarily small convergence parameter, ϵ must be chosen; the search will return values of $\vec{x}$ for which the magnitude of $\vec{F}(\vec{x})$ is smaller than ϵ, i.e., $|\vec{F}(\vec{x})| < \epsilon$.

The steps in this search are as follows:

1. Select an initial set of values for the elements of $\vec{x}$, referred to as $\vec{x}_i$, which are presumed to be close in value to one of the zeroes of $\vec{F}(\vec{x})$.

2. Compute $\vec{F}(\vec{x}_i)$ and the Jacobian matrix of partial derivatives at $\vec{x}_i$,

$$\mathbf{J}_i = \left.\frac{\partial \vec{F}}{\partial \vec{x}}\right|_{\vec{x}=\vec{x}_i} = \begin{bmatrix} \left.\frac{\partial f_1}{\partial x_1}\right|_{\vec{x}=\vec{x}_i} & \left.\frac{\partial f_1}{\partial x_2}\right|_{\vec{x}=\vec{x}_i} & \cdots & \left.\frac{\partial f_1}{\partial x_n}\right|_{\vec{x}=\vec{x}_i} \\ \left.\frac{\partial f_2}{\partial x_1}\right|_{\vec{x}=\vec{x}_i} & \left.\frac{\partial f_2}{\partial x_2}\right|_{\vec{x}=\vec{x}_i} & \cdots & \left.\frac{\partial f_2}{\partial x_n}\right|_{\vec{x}=\vec{x}_i} \\ \vdots & \vdots & \ddots & \vdots \\ \left.\frac{\partial f_n}{\partial x_1}\right|_{\vec{x}=\vec{x}_i} & \left.\frac{\partial f_n}{\partial x_2}\right|_{\vec{x}=\vec{x}_i} & \cdots & \left.\frac{\partial f_n}{\partial x_n}\right|_{\vec{x}=\vec{x}_i} \end{bmatrix}$$

   The partial derivative is typically calculated numerically, using a difference formula such as

$$\left.\frac{\partial f_j}{\partial x_k}\right|_{\vec{x}=\vec{x}_i} \approx \frac{f_j(x_1, x_2, \ldots, x_k + \Delta x_k, \ldots x_n) - f_j(x_1, x_2, \ldots, x_k - \Delta x_k, \ldots x_n)}{2\Delta x_k}$$

   where the values of all the parameters (except x_k) are held constant at the values specified by $\vec{x}_i$.

3. If $|\vec{F}(\vec{x}_i)| < \epsilon$, then $\vec{x}_i$ is the desired zero.

4. If $|\vec{F}(\vec{x}_i)| \geq \epsilon$, then a new value for $\vec{x}$ is chosen, $\vec{x}_{i+1}$, corresponding to the point where the gradient vector at $\vec{x}_i$ intersects the $\vec{F}(\vec{x}) = \vec{0}$ plane, i.e.,

$$\vec{x}_{i+1} = \vec{x}_i - (\mathbf{J}_i)^{-1} \vec{F}(\vec{x}_i)$$

   Note that this step requires the inversion of the Jacobian matrix.

5. Return to Step 2, repeating the process, substituting $\vec{x}_{i+1}$ for $\vec{x}_i$.

As can be seen, this algorithm is basically an extension into multiple dimensions of the original (single-dimension) Newton-Raphson method.

In applying this technique to the solution of computational loops, the parameter vector, $\vec{x}$, will be a vector whose elements are the values of the loop's attributes. The elements of the function vector, $\vec{F}(\vec{x})$, are calculated by applying the LISP functions stored as the *compiled-normal-function* instance variables of the loop's constraints to the attribute-values stored in $\vec{x}$ or, for those parameters which are not being solved for in the loop, the corresponding attributes' current values. The final result for $\vec{x}$ will yield the solution-values for the loop attributes.

As mentioned above, however, stability problems were encountered when this technique was applied to very large computational loops (e.g., a twenty-element loop for computing the gross weight and various component weights of an aircraft design, see Chapter 4): the algorithm would not converge. The reason for this instability is the need for good initial conditions when employing the Newton-Raphson iteration. In the case of one-dimensional Newton-Raphson, employed for numerical constraint inversion (see Section 3.3.2), a simple search strategy has been implemented to provide suitable initial conditions. Because simultaneous Newton-Raphson involves multiple parameters, searching for appropriate initial conditions within the corresponding multi-dimensional space is too costly. Instead, the current (pre-iteration) values of the loop's attributes are used to provide the initial values for the parameter vector. While some of these values may provide a good starting point for Newton-Raphson iteration, it is always possible that one or more of these attribute-values will lead to divergence rather than convergence. As the size of the computational loop grows, the likelihood of such poor initial conditions grows, since more attribute-values are involved. To overcome these difficulties associated with large computational loops, then, a second solution method has been adopted.

Iteration Loops

During the course of this research, it has been observed that, of the computational loops encountered in the course of implementing the various test cases (see Chapter 4), all were either very small, involving only two or three attribute/constraint pairs, or very large, involving twenty or more such pairs. As indicated in the preceding sections, the simultaneous Newton-Raphson method is well-suited to solving the smaller computational loops, computing the required solutions both quickly and reliably, but encounters difficulty when presented with larger loops.

It has also been observed that these larger loops are typically based on calculation of vehicle weight characteristics. Recalling the heuristic basis on which computational loops are constructed (see Section 3.4.1), it was therefore deemed appropriate to solve such loops

in the same manner as they are solved in conventional computer programs for vehicle design, by constructing a simple iteration loop which assumes a value for the loop variable, applies the loop's constraints to compute values for the remaining parameters, and then, based on these computed values, uses the closing constraint to compute a new value for the loop variable, repeating this process until convergence upon a stable value for the loop variable is observed.

To implement this technique, it is a simple matter to iterate through the computational loop structure, as it has been described in Section 3.4.2, applying the constraint from each pair in the loop to compute a value for the corresponding attribute, holding the values of all of its other parameters constant; recall that the final pair in the computational loop consists of the loop variable and the closing constraint. Convergence is detected by monitoring the value of the loop variable from one pass through the loop to the next. As is the case when employing the simultaneous Newton-Raphson method, the current pre-iteration values of the loop's attributes are used as the initial conditions for this analysis.

The only difficulty encountered in applying this approach has been a strong sensitivity to the choice of closing constraint. As mentioned above, most of the large loops encountered to date have involved calculation of vehicle and component weights. For such loops, it is important to the stability of this method that the summation constraint which collects all of the component weights in order to compute the total vehicle weight serve as the closing constraint. In the loop-detection algorithm described in Section 3.4.2, however, there is no guarantee that this will always be the case.

For example, consider a set of constraints for computing the component and gross weights of an aircraft, where the component weights are computed as some fraction of the gross weight. Assume that the vehicle gross weight has been chosen as the loop variable, and that loop detection has proceeded to the point where two loop-eligible constraints remain, one which computes the fuselage weight as a fraction of the gross weight, and another which computes the gross weight as the sum of all component weights, including the fuselage weight. One of these two constraints must be chosen for examination, if loop propagation is to proceed. If the fuselage-weight constraint is chosen next, it may serve as the loop supplier for the fuselage weight, based on the assumed value for the gross weight. The summation constraint will then serve as the closing constraint, which is the desired outcome from the point of view of iteration-loop stability. If the weight-summation constraint is chosen next, however, it will serve as the loop-supplier for the fuselage weight parameter, again based on the assumed value for the gross weight. It will be the fuselage-weight constraint, which also has vehicle gross weight as one of its parameters, which is selected as the closing constraint. When this path is taken, it has been observed that the iteration-loop technique is less likely to converge.

For this reason, a slight modification to the loop detection algorithm is required. Note that, in Section 3.4.2, no criterion for ordering the propagation of loop-eligible constraints is indicated: loop-eligible constraints are propagated as soon as they are observed to be perfectly constrained. Unfortunately, this approach enables non-optimal choices for loop closing constraints to be made, as described above. To overcome this problem, it is necessary to introduce a means for ranking loop-eligible constraints prior to propagation. One approach, the approach employed in *Rubber Airplane*, is to maintain an ordered queue of all perfectly constrained loop-eligible constraints; loop propagation then proceeds by propagating the first element of the queue, rather than propagating such constraints as soon as they are encountered while examining the *forward-constraints* and *reverse-constraints* instance variables of the corresponding attribute instances.

A criterion for ordering these constraints is also required. Recall that the immediate objective of this ordering is to ensure that, for weight-like loops, the summation constraint be selected as the closing constraint. It is therefore desirable that propagation of this constraint be delayed. The overall objective, however, is improving the convergence characteristics of iteration loops. In the example presented above, a choice had to be made between propagating the fuselage-weight constraint and the weight-summation constraint. Note that the fuselage-weight constraint basically relates only two of the loop's parameters, the weight of the fuselage and the net weight of the entire vehicle. The summation constraint, however, relates not only these two parameters, but all of the other component-weight parameters, a majority of which are also likely to be parameters of the computational loop, as well. The summation constraint is therefore much more dependent upon the calculations performed by the constraints which precede it in the iteration loop (i.e., those which compute the various component weights), than is the fuselage-weight constraint. It is this lack of dependency of the fuselage-weight constraint which makes it a poor choice for closing constraint, and inhibits iteration-loop convergence.

Thus, although the argument has been presented in terms of a specific example, there is an underlying general principle at work here, which suggests that, for stability, the choice of closing constraint for an iteration loop should be based upon constraint interdependencies. Fortunately, the implementations of local and loop-based propagation employed here provide a ready means for measuring such interdependences. Note that loop propagation is based upon loop-suppliers, which in turn get their initial values from the attributes' value-suppliers. Thus, while loop propagation is taking place, these value-suppliers are always available as a means for gauging the progress of loop propagation. Specifically, recall that a constraint becomes ready for loop propagation when the loop-suppliers of its parameters indicate that the constraint is perfectly constrained. If, however, this constraint is observed to have a large number of free parameters *according to the value-suppliers of its attributes*,

then it must be the case that the constraint is highly dependent upon the loop propagations which have already been performed. If the constraint has a relatively small number of free parameters—again, based upon value-suppliers rather than loop-suppliers—then the constraint is correspondingly less dependent upon previous loop propagations.

This observation suggests, then, that an appropriate criterion for ordering constraints for loop propagation is to rank the constraints whose parameters' loop-suppliers indicate that they are perfectly constrained according to the constraint's number of free parameters, as indicated by those parameters' value-suppliers. Constraints with fewer such free parameters should be located earlier in the queue; those with more—such as the weight-summation constraint in the example above—are placed later in the queue, thereby delaying their propagation. One means for implementing this ranking is by maintaining an ordered queue of the constraints which are determined to be ready for loop propagation. Thus, Step 6 of the original loop propagation algorithm (see Section 3.4.2) is subdivided into three separate steps, as follows:

6.A. Examine all of invertible equality constraints associated with this attribute (i.e., the attribute which has just been assigned a new loop-supplier). For each such constraint, based upon the loop-suppliers associated with its parameters,

- If the constraint is loop-eligible but has no free parameters, proceed to Step 9.
- If the constraint is loop-eligible and has exactly one free parameter, then add this constraint to queue of constraints ready for loop propagation.

6.B. Sort the queue of loop-ready constraints according to the number of free parameters associated with each constraint, based on value-suppliers. Those with fewer free parameters are ranked ahead of those with larger numbers of free parameters.[7]

6.C. Remove the first constraint from the queue, and perform the following sequence of operations:

1. Set the *in-loop?* instance variable of the constraint to t.[8]
2. Set the loop supplier of the attribute which is the constraint's free parameter to be the constraint. Add the attribute and the constraint to the record of loop-supplier assignments.

[7]Actually, it is not necessary to repeatedly re-sort this queue. Since the count of a constraint's free parameters—based on value-suppliers—does not change during loop propagation, and because only one constraint is added to the queue at a time, it is most efficient to perform sorting as constraints are enqueued. In this way, it is only necessary to perform enough comparisons to determine the location in the queue at which the newly added constraint should be inserted.

[8]Recall that, as indicated in Section 3.4.2, at this point, any occurrence of this constraint in the queue of candidate loop constraints may be removed.

3. Return to Step 6.A, applying it to the attribute identified as the constraint's free parameter.

By modifying the loop propagation algorithm in this manner, improved performance of the iteration loop method is observed, due to the corresponding changes in structure of the resulting computational loops.

In closing, note that while it is not a particularly efficient approach, the use of iteration loops has proved a reliable method for solving the types of large computational loops encountered in the test problems implemented thus far. When combined with the more versatile simultaneous Newton-Raphson method, these two techniques have proven adequate for the solution of simultaneous non-linear equations in aerospace engineering conceptual design. The success of the approaches described here for both detecting and solving such systems provides good evidence for the general validity of the computational-loop heuristic in this problem domain.

3.5 Optimization

Before concluding this chapter on constraint propagation in *Rubber Airplane*, brief mention should also be made of some experiments performed at Lockheed-Georgia, aimed at integrating optimization techniques with *Rubber Airplane*. Using the Fortran compiler available on the Symbolics Lisp Machine, a Fortran-based numerical optimization program, OPT [12], was made accessible from within the *Rubber Airplane* program. Specifically, the optimization routines were used to augment the local propagation mechanisms presented in Section 3.2. (At the time, the loop detection and solution algorithms had not yet been implemented.)

The general problem of mathematical optimization may be described as follows. Consider a set of k parameters, $x_1, x_2, \ldots, x_k$, represented by the vector $\vec{x}$. The goal of optimization is to maximize the scalar function, $f(\vec{x})$, subject to the constraints,

$$\begin{aligned} \vec{G}(\vec{x}) &= \vec{0} \\ \vec{H}(\vec{x}) &\geq \vec{0} \end{aligned}$$

where $\vec{G}(\vec{x})$ is a vector of m functions of the k parameters, representing a set of m equality constraints which the solution must satisfy. Similarly, $\vec{H}(\vec{x})$ is a vector of n functions of these parameters, representing a set of n inequality constraints which the solution must satisfy. The OPT program uses numerical, gradient-based techniques for solving optimization problems of this form.

In integrating *Rubber Airplane* with OPT, local propagation was employed to compute values for a subset of those attributes of a given design whose values had not been assigned

by the user. At the user's request, the optimization program could then be called upon to solve for the remaining attributes. Note that, typically, local propagation is not adequate for solving all of the constraints associated with a design. In an optimization program, there are normally multiple degrees of freedom in the constraint network; under such circumstances, local propagation cannot be used to solve all of the constraints, because the system is necessarily underconstrained. The optimization routines may be used both for the solution of simultaneous equations within the constraint network (for which local propagation is inadequate) and as a means for utilizing the inequality constraints and the maximization function (i.e., $f(\vec{x})$) to solve systems which are underconstrained.

Within *Rubber Airplane*, the maximization function is specified by selecting a single scalar attribute whose value is to be maximized or minimized. Because *Rubber Airplane* permits the interactive definition of new attributes and constraints, if the user actually wishes to optimize some function of multiple attributes, it is a simple matter to define a problem-specific constraint representing this function, as well as a corresponding attribute to serve as this constraint's output parameter. Note that minimization is simulated by maximization of the attribute's opposite (i.e., to minimize z, maximize $-z$).

Once such an attribute has been selected, and any user-supplied attribute values have been assigned, the OPT program may be called upon to apply those constraints not used during local propagation to solve for any remaining attributes with a value-supplier of **:guess**. The parameter vector $\vec{x}$ will represent the values of these attributes. The vector of equality constraints, $\vec{G}(\vec{x})$, is constructed from those equality constraints which have not been used during local propagation (i.e., those constraints for which the values of their *computed-parameters* instance variables is the empty list). Because equality constraints are expected to be of the form $g_i(\vec{x}) = 0$, whenever evaluation of one of these constraints is required by the optimization program, the LISP function stored in the constraint's *compiled-normal-function* instance variable is applied. Similarly, the vector of inequality constraints, $\vec{H}(\vec{x})$, consists of all of the inequality constraints associated with the current design which have at least one of their parameters represented in $\vec{x}$. Inequality constraints are expected to be of the form $h_i(\vec{x}) \geq 0$; like the *compiled-normal-function* instance variable associated with equality constraints, inequality constraints are provided with a similar instance variable for storing a compiled LISP function which evaluates the inequality constraint in the required form. Once the analysis has been completed, the parameter values computed for $\vec{x}$ by the optimization program therefore represent those values for the corresponding attributes which result in an optimal value for the chosen maximization or minimization attribute, yet still satisfy all of the relevant equality and inequality constraints.

Note that it was for their utility in bounding the optimization search space that inequality constraints were first added to *Rubber Airplane*. In the current implementation

of *Rubber Airplane*, which does not support optimization, inequality constraints are used only to monitor constraint propagation: the user is automatically notified of any violations of a design's inequality constraints.

While time was not available for full-scale testing of these optimization features, these efforts have nevertheless demonstrated the feasibility of incorporating numerical optimization techniques into a symbolically-oriented design tool, such as *Rubber Airplane*. Considering the potential advantages resulting from the ability to incorporate optimization into conceptual and preliminary design analyses, further research is recommended to more thoroughly investigate the utility of such a hybrid approach.

Chapter 4

Test Cases

4.1 Motivation

In the preceding chapters, the description of a general-purpose system for implementing and solving problems in engineering conceptual design has been presented. To verify the utility of this approach, however, its application to realistic design tasks is required. For this reason, three sample problems have been implemented using the *Rubber Airplane* prototype design tool. The three vehicle designs which have been examined are:

- a long-endurance, manned surveillance aircraft;
- a subsonic commercial transport aircraft; and
- a small-payload launch vehicle.

Note that the goal of this exercise was to demonstrate the ability of component-modeling and constraint propagation to support the types of analyses required for such design problems. Time did not permit the development of complete analytical models for any of the three test cases; effort was therefore focused upon the implementation of representative analysis modules, as a proof-of-concept study. As such, the actual results of the underlying design efforts are relatively unimportant; it is the process whereby these results were made attainable that is of primary relevance.

In the sections which follow, the means by which the appropriate design-specific analyses were implemented will be discussed. The advantages and disadvantages associated with the use of constraint-based component-modeling in representing the required design knowledge will also be presented. In this manner, the practicality of this approach in supporting realistic conceptual design tasks may be evaluated.

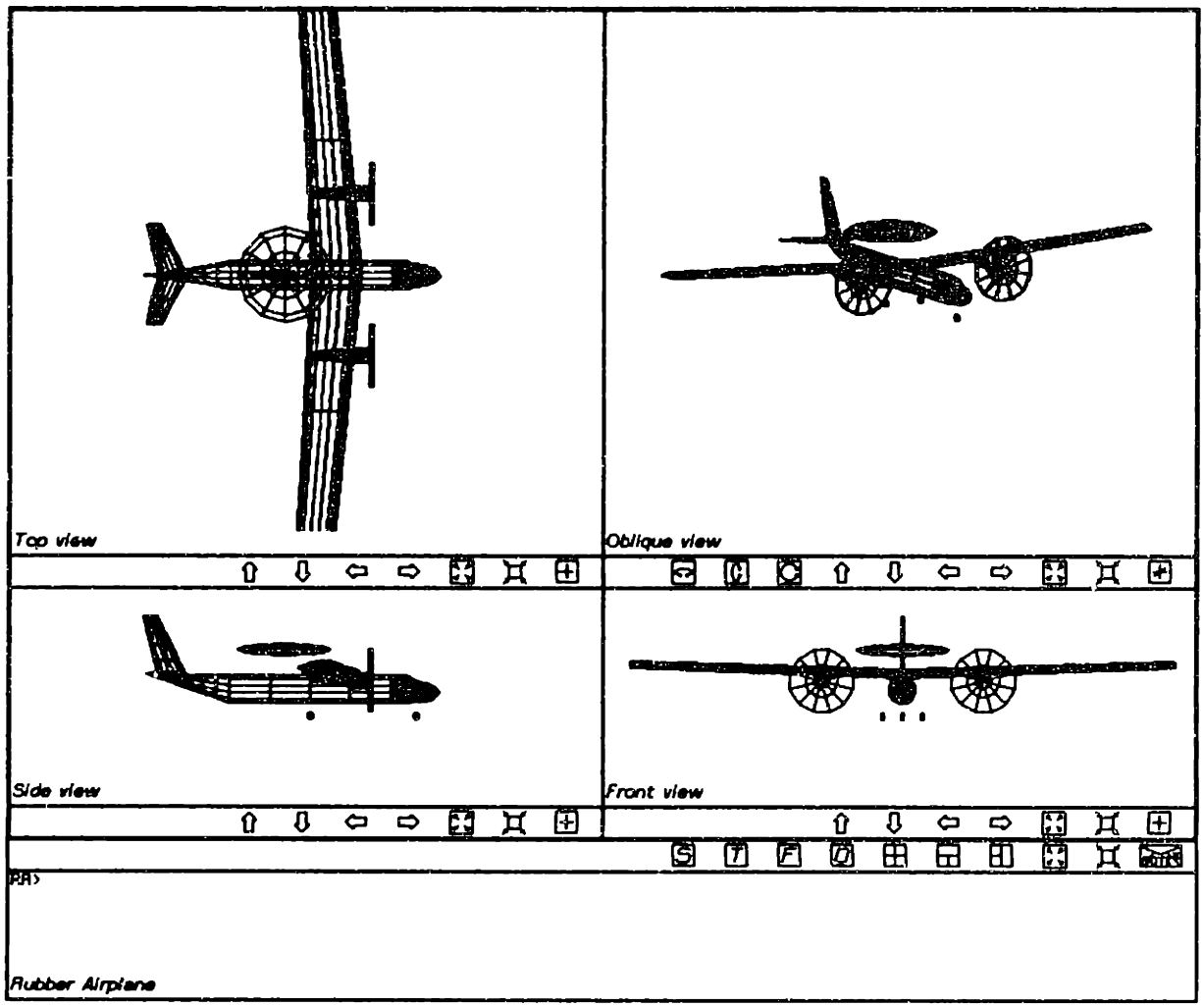

Figure 4.1: Screen image depicting the completed design for the surveillance aircraft test case.

4.2 Long-Endurance, Manned Surveillance Aircraft

4.2.1 Specifications

The first test case calls for the design of a manned, radome-equipped surveillance aircraft. The vehicle is intended to serve as a high-altitude observation platform and communications-relay station, and is required to remain on station for periods of up to 24 hours, without refueling. The dimensions and weight of the radome are fixed, and a crew of five is specified. Furthermore, because the vehicle is intended for use in maritime surveillance, a cruising range of 1000 nautical miles from base to loitering station (and back) must be supported.

Component Class	Description
basic-fuselage	Fuselage geometry for a transport-type aircraft.
basic-wing	Airfoil geometry for a swept, tapered wing, for which different cross-sections may be specified at the root and tip.
basic-vertical	Airfoil geometry for a vertical stabilizer.
basic-horizontal	Airfoil geometry for a horizontal stabilizer.
radome	Geometry for an external, fuselage-mounted radome.
turboprop-pair	Geometry for a pair of wing-mounted turboprop engines.
fuel-system	Component representation of fuel and fuel tanks
forward-landing-gear	A simple forward landing gear (tire only).
rear-landing-gear	A pair of rear landing gear tires.
controls	Component representation for aircraft controls.
systems	Component representation for aircraft support systems.
air-conditioning	Component representation for air-conditioning system.
avionics	Component representation for aircraft control avionics.
radar-equipment	Component representation for radome support systems.
crew	Component representation of aircraft crew.
simple-payload	Component representation for crew support systems.

Table 4.1: Instantiated component-classes for the surveillance aircraft test case.

4.2.2 Problem Representation

In implementing this test case, effort was focused on both mission performance and aerodynamics. Propulsion data was entered based on a paper study of the design requirements, and structural analysis was ignored. A total of 16 components, 10 design states, and 39 links were employed in representing the implemented analyses, with a total of 530 attributes and 377 constraints.[1] A screen image displaying the geometry of the completed design appears in Figure 4.1.

The component-classes used in representing the vehicle are listed in Table 4.1, along with descriptions of their various functions. As is evident from this table, there are components representing each of the major physical components (fuselage, wings, radome, etc.), as well as a number of components for representing various subsystems (avionics, controls, etc.). Note that these latter components do not have any geometry associated with them, since they actually represent groupings of multiple components; they do, however, enable the

[1]Note that, throughout this chapter, statistics regarding attributes and constraints will combine all the state-instances of a particular state-dependent attribute or constraint together, such that they count as but a single attribute or constraint of the owning design entity.

Component Class	Superclasses	Attributes	Constraints
basic-fuselage	4	25	14
basic-wing	11	51	28
basic-vertical	11	46	26
basic-horizontal	11	51	28
radome	5	18	10
turboprop-pair	2	23	8
fuel-system	2	19	11
forward-landing-gear	4	17	9
rear-landing-gear	4	18	9
controls	2	11	7
systems	2	17	10
air-conditioning	2	17	10
avionics	2	17	10
radar-equipment	2	17	10
crew	3	18	10
simple-payload	2	17	10
Total:		382	210

Table 4.2: Class statistics for the instantiated component-classes of the surveillance aircraft test case.

use of statistically-derived equations for the subsystem weights to be incorporated into the design.

Various statistics associated with all of the component-classes instantiated for this design are presented in Table 4.2. The number of attributes and constraints for each component-class gives some indication of the level of detail associated with the corresponding component. This table also lists the number of superclasses which are associated with each of the instantiated component-classes; these numbers indicate the degree to which the use of object-oriented programming techniques aided in the implementation of the component-class. Classes with a large number of superclasses were able to take advantage of inheritance and specialization to simplify component representation by means of decomposition: each superclass implements a different aspect—e.g., position, dimensions, gross properties—of the component. (Recall, however, that each instantiable component-class includes at least one superclass, the base **design-component** class, which itself specifies 11 attributes and 4 constraints.) Table 4.3 presents some of the details of this decomposition of component-class functionality, by listing and describing those component-classes which most often serve as superclasses of the instantiated component-classes.

As indicated above, ten design states have been employed to represent the mission

Component Class	Subclasses	Description
design-component	21	Base component-class.
basic-drag-mixin	13	Provides a general-purpose **drag** attribute.
point-mass-component	12	Equates component reference and center-of-gravity positions.
basic-lift-mixin	9	Provides a general-purpose **lift** attribute.
basic-aerodynamics-mixin	8	Combines **basic-lift-mixin** and **basic-drag-mixin**.
basic-section	7	Provides general airfoil cross-section attributes.
NACA-section-pair	6	Provides attributes to represent NACA-airfoil cross sections at both the root and tip of an airfoil.
airfoil-planform	6	Provides airfoil planform parameters.
induced-drag-mixin	6	Breaks down component drag into induced drag and profile drag components.
basic-airfoil	5	Combines **NACA-section-pair** and **airfoil-planform**.
horizontal-planform-mixin	3	Represents a horizontally-oriented airfoil planform.
vertical-planform-mixin	2	Represents a vertically-oriented airfoil planform.
basic-horizontal-airfoil	2	Combines **basic-airfoil** and **horizontal-planform-mixin**.
basic-landing-gear	2	Provides general landing-gear attributes.

Table 4.3: Most commonly occurring component-classes among the instantiated component-classes of the surveillance aircraft test case.

Design State Name	Description
"Takeoff"	End of takeoff run.
"Intermediate Climb"	Intermediate flight condition during climb to cruising altitude.
"Cruise Out"	Beginning of cruise out to station.
"Loiter"	Beginning of on-station loiter.
"Post-Loiter Climb"	Start of post-loiter climb to return-trip cruising altitude.
"Cruise In"	Beginning of cruise back to base.
"Begin Descent"	Start of descent for landing after cruise back to base.
"Begin Circle"	Start of loiter at base in preparation for landing.
"End Circle"	Beginning of final approach for landing.
"Landing"	Flight condition for vehicle landing.

Table 4.4: Instances of class **design-state** for the surveillance aircraft test case.

profile for this aircraft. A listing of these states appears in Table 4.4. In accordance with the observations presented in Section 2.4.4 of Chapter 2, all the state-instances listed in this table are instances of the base state-class, **design-state**.

The majority of the analyses associated with this design task are implemented via design links. A listing of the relevant link-classes, with brief descriptions of their intended roles, appears in Tables 4.5 and 4.6. The statistics for these classes are presented in Table 4.7; the significance of the first three columns of numbers is the same as for the data presented in Table 4.2 for the test case's component-classes. Note that, for the most part, the link-classes which have been employed have relatively fewer superclasses than the component-classes listed in Table 4.2; this is because the analyses implemented as design links vary widely in nature. Because they rely heavily on class-specific attributes and constraints, they are not amenable to the use of shared superclasses; those few link-classes which do appear as shared superclasses are listed in Table 4.8. On the other hand, however, the final column in Table 4.7 indicates that for a number of these link-classes multiple instances have been employed. Such occurrences suggest that, in these cases, a second means for exploiting object-oriented programming techniques is active, insofar as the corresponding analyses were defined only once, but applied several times.

4.2.3 Design Analysis

The primary analyses implemented for this test case fall into four basic categories: geometry, component weights, mission performance, and aerodynamics. The general characteristics of each of these analyses are discussed below.

Link Class	Description
flight-conditions	State-dependent description of the vehicle flight conditions.
gross-properties	Net weight, thrust, and aerodynamic forces on the vehicle.
wing-attachment	Sets the y- and z-coordinates of the wing based upon corresponding fuselage dimensions and coordinates.
tail-attachment	Attaches the horizontal and vertical stabilizers to the rear of the fuselage.
turboprop-wing-attachment	Attaches a symmetric pair of engines to the wings accounting for airfoil geometry and propeller clearance.
cockpit-attachment	Positions generic objects with respect to the cockpit portion of the fuselage.
cabin-attachment	Positions generic objects with respect to the cabin portion of the fuselage.
takeoff-distance	Computes the required takeoff distance for the vehicle.
landing-distance	Computes the required landing distance for the vehicle.
fuselage-weight-model	Computes the weight of the fuselage, based on dimensions and net vehicle weight.
wing-weight-model	Computes the weight of the wing, based on dimensions and net vehicle weight.
fuel-tank-weight-model	Computes the weight of the fuel tanks, based on total fuel weight.
controls-weight-model	Computes the weight of the aircraft control systems, based on net vehicle weight.
systems-weight-model	Computes the weight of the support systems, based on net vehicle weight.
forward-gear-weight-model	Computes the weight of the forward landing gear, based on net vehicle weight.
rear-gear-weight-model	Computes the weight of the rear landing gear, based on net vehicle weight.
person-weight-model	Computes the weight of a group of persons.
takeoff-fuel	Computes the fuel required for idle and takeoff.
climb-fuel	Computes the fuel required for a single climb segment.
propeller-cruise-fuel	Computes the fuel required for a single cruise segment
propeller-loiter-fuel	Computes the fuel required for a single loiter segment
propeller-circle-fuel	Computes the fuel required for a single circling segment
glide-fuel	Equates the vehicle weights for two design states.
total-fuel-weight	Sums the fuel weights required for each flight segment.

Table 4.5: Instantiated link-classes for the surveillance aircraft test case.

Link Class	Description
two-airfoil-vortex-lattice	Computes the lift and induced drag for a pair of airfoils.
wing-profile-drag	Computes wing profile drag, based on wetted area.
vertical-profile-drag	Computes vertical stabilizer profile drag, based on wetted area.
horizontal-profile-drag	Computes horizontal stabilizer profile drag, based on wetted area.
fuselage-profile-drag	Computes fuselage profile drag, based on wetted area.
radome-profile-drag	Computes radome profile drag, based on wetted area.
forward-gear-profile-drag	Computes profile drag for the forward landing gear, based on wetted area.
rear-gear-profile-drag	Computes profile drag for the rear landing gear, based on wetted area.

Table 4.6: Instantiated link-classes for the surveillance aircraft test case (continued).

Component Geometry

Parameters describing component geometries, such as position and dimensions, are associated with the component-classes themselves. Constraints which relate the geometric parameters of a single component, such as the definitions of aspect ratio and taper ratio for an airfoil, are also associated with the corresponding component-classes. Only parameters which represent relative geometric properties, such as the positions of two objects which are attached to one other, need be implemented using design links. Various examples of attachment link-classes appear in Table 4.5. As can be seen in Table 4.7, such link-classes, because they are component-specific, rarely have multiple superclasses. The one exception to this observation in Table 4.7 is the **tail-attachment** link-class, which is listed as having three superclasses; class **tail-attachment** is actually a combination of two separate attachment link-classes, one for attaching the horizontal stabilizer to the fuselage, and another for attaching the vertical stabilizer. (The base link-class, **design-link**, is the third superclass for class **tail-attachment**.) Note that, as indicated in Table 4.8, multiple instances of the **cabin-attachment** and **cockpit-attachment** link-classes have been employed to position various non-geometric components (as represented by, for example, the **controls**, **avionics**, and **crew** component-classes) at appropriate locations within the vehicle's fuselage.

Recall that the base component-class, **design-component**, also provides attributes for representing the coordinates of a component's center of gravity. Most of the component-classes listed in Table 4.1 include constraints for computing the center-of-gravity location. A

Link Class	Superclasses	Attributes	Constraints	Instances
flight-conditions	2	11	6	1
gross-properties	1	14	13	1
wing-attachment	1	1	2	1
tail-attachment	3	0	6	1
turboprop-wing-attachment	1	1	3	1
cockpit-attachment	1	0	3	3
cabin-attachment	1	0	3	3
takeoff-distance	1	14	10	1
landing-distance	1	12	6	1
fuselage-weight-model	1	0	1	1
wing-weight-model	1	1	2	1
fuel-tank-weight-model	1	0	1	1
controls-weight-model	1	0	1	1
systems-weight-model	1	0	1	1
forward-gear-weight-model	1	0	1	1
rear-gear-weight-model	1	0	1	1
person-weight-model	1	1	1	1
takeoff-fuel	2	2	2	1
climb-fuel	3	4	4	3
propeller-cruise-fuel	3	3	3	2
propeller-loiter-fuel	3	2	2	1
propeller-circle-fuel	3	2	2	1
glide-fuel	1	0	1	1
total-fuel-weight	1	3	3	1
two-airfoil-vortex-lattice	1	27	20	1
wing-profile-drag	3	5	6	1
vertical-profile-drag	3	5	6	1
horizontal-profile-drag	3	5	6	1
fuselage-profile-drag	3	6	7	1
radome-profile-drag	3	6	7	1
forward-gear-profile-drag	3	6	7	1
rear-gear-profile-drag	3	6	7	1
Total:		148	167	39

Table 4.7: Class statistics for the instantiated link-classes of the surveillance aircraft test case.

Link Class	Subclasses	Description
design-link	32	Base link-class.
basic-profile-drag-mixin	13	General-purpose induced drag model, based on wetted area and skin-friction coefficient.
fuel-consumption-mixin	6	Provides a general-purpose **fuel-weight** attribute.
transitional-fuel-consumption-mixin	4	Represents fuel weight as the change in vehicle weight between two design states.
body-profile-drag	4	Specializes **basic-profile-drag-mixin** for non-lifting bodies.
airfoil-profile-drag	3	Specializes **basic-profile-drag-mixin** for airfoils.

Table 4.8: Most commonly occurring link-classes among the instantiated link-classes of the surveillance aircraft test case.

number of component-classes incorporate the **point-mass-component** component-class as a superclass; the constraints of this class serve to equate the coordinates of a component's reference position (as indicated by its **position-x**, **position-y**, and **position-z** attributes) with those of its center of gravity (represented by the **cg-x**, **cg-y**, and **cg-z** attributes). This superclass is useful for both the non-geometric component-classes (as listed above), as well as for certain components which are represented as simple bodies of revolution (e.g., the landing gear and radome).

Component Weights

The weights of the various vehicle components are, for the most part, computed by means of statistically-derived equations based on component dimensions and total vehicle weight. An example link-class implementing such a model for the fuselage is presented in Figure 4.2. The weights of certain components—most notably the engines and the radome—have been input as constants. These component-weight constraints, in combination with the mission-performance constraints responsible for calculating net fuel weight, form the basis of the primary iteration loop for this test-case design, the closing constraint of which is a collector constraint for computing the vehicle's gross takeoff weight.

```
(deflink fuselage-weight-model ((fuselage :class basic-fuselage)
                                (vehicle :class gross-properties))
        ()
        (design-link)
   (:category weights)
   (:documentation "Empirical weight model for the fuselage of a
transport-type aircraft, based on fuselage dimensions, vehicle gross
weight, and vehicle load factor.")

(defconstraint (fuselage-weight-model "Fuselage Weight")
               (weight@fuselage "lbf")
               ((gross-weight@vehicle "") (load-factor@vehicle "")
                (total-length@fuselage "ft") (height@fuselage "ft")
                (width@fuselage "ft"))
  "Empirical weight model for aircraft fuselage weight."
  (let ((max-diameter (max height@fuselage width@fuselage)))
    (* 0.8 (expt total-length@fuselage 1.5)
       (expt max-diameter .25)
       (expt (* load-factor@vehicle gross-weight@vehicle) 0.15))))
```

Figure 4.2: LISP definition for a link-class which implements an empirical component-weight model, **fuselage-weight-model**.

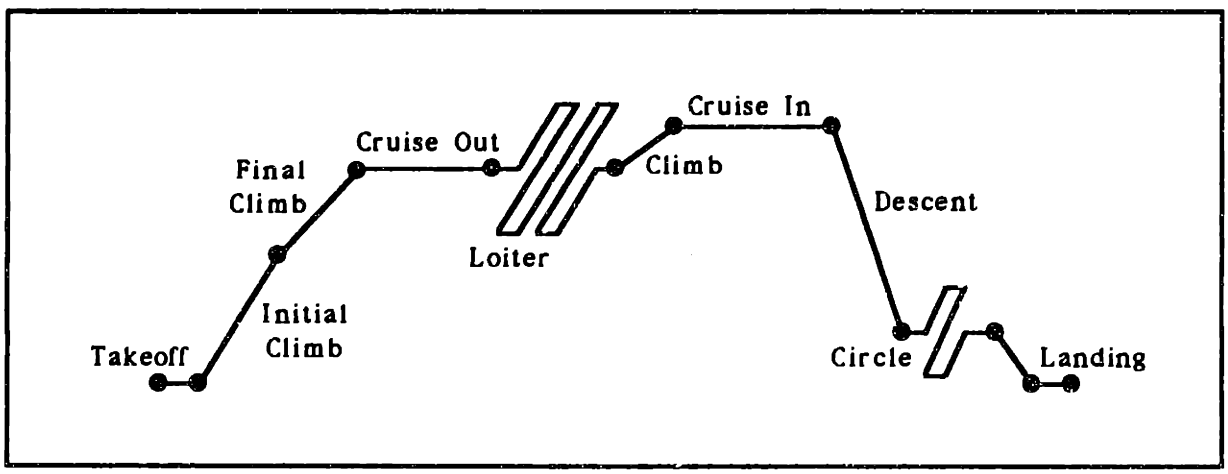

Figure 4.3: Mission profile for the surveillance aircraft test case.

Mission Performance

As indicated in Table 4.4, ten design states have been employed to represent the various flight segments which comprise the aircraft's mission profile, which is depicted graphically in Figure 4.3. These states represent specific design points within the mission profile, as described in Table 4.4. An instance of the `flight-condition` link-class provides state-dependent attributes for the atmospheric properties, as well as the vehicle's net weight, angle of attack, and flight speed.

Two aspects of mission performance analysis have been implemented: fuel consumption and runway performance. Various range- and endurance-based equations are employed to calculate the fuel requirements for the cruising, loitering, and circling flight segments. The takeoff and landing analyses are based on runway type, maximum lift coefficient for the vehicle, and—for the takeoff analysis—engine static thrust.

The fuel consumption equations require values for such parameters as engine specific fuel consumption, net vehicle lift-to-drag ratio, and vehicle gross weight for each flight leg. Noting from Figure 4.3 that there is a total of ten flight legs, ten independent values for each of these parameters are required. Prior to the introduction of design states, this could only be done by either:

- associating ten different attributes for each parameter with each relevant component- and link-class (e.g., have ten different `lift-to-drag` attributes for each airfoil component, so that ten different vehicle lift-to-drag ratios may be computed); or
- defining a design-link which incorporates all of the parameters required for the entire mission profile.

In the first case, each of the attributes added to the component-class must have a different name. Furthermore, if one wished to use a similar component description for a design problem whose mission profile required a different number of flight legs, implementation of a completely new component-class would be necessary. Clearly, such an approach does not lend itself to modular code development: the resulting component-classes are not reusable.

The second approach is somewhat more satisfactory, insofar as it does not lead to design-specific component-classes. Instead, it requires a design-specific link-class, which implements a particular mission profile. In addition, it will be the case that various component-specific attributes—for example, engine specific fuel consumption—must be implemented as link-class attributes, rather than component-class attributes, since in this case it is the link-class which is responsible for creating the parameters required for each flight leg. Furthermore, it will often be the case that such all-encompassing mission-profile link-classes must explicitly incorporate multiple versions of the same analyses. In this test case, for instance, the same set of calculations must be performed for the "loiter" flight leg as for

the "circle" flight leg; fuel calculations for the three climb phases, and for the two cruise conditions will also be similar.

The means for avoiding such design-specific component- and link-classes is through the use of design states; indeed, it was during the development of the mission profile analyses for this test case that the need for design states and state-dependent attributes and constraints was first recognized. By defining the various flight-leg parameters (e.g., specific fuel consumption, flight speed, vehicle lift-to-drag ratio) as state-dependent attributes, an infinite variety of mission profiles may be supported. Furthermore, a separate link may be used for each flight leg; multiple instances of the same link-class may be employed for similar flight legs (e.g., three instances of the `climb-fuel` link-class may be applied for analyzing the three climb phases of the mission profile. Thus, the various fuel calculations are implemented by providing each of the leg-specific link-classes with one linkage for the design-state which begins the flight leg, and another linkage for the design-state which completes the flight leg; an example of such a link-class—implementing the Bréguet range equation—appears in Figure 4.4. The fuel requirements are then totalled by means of a collector constraint defined by the `total-fuel-weight` link-class.

Aerodynamics

In view of the current practice of performing more complicated analyses earlier in the engineering design cycle, it was decided that some attempt at integrating more rigorous analytical methods into a *Rubber Airplane* test case should be made. Whereas very simple relationships have been employed for estimating vehicle weight and performance, a more demanding technique for computing airfoil aerodynamics has been implemented. Specifically, the vortex-lattice method [22,27] is used for computing airfoil lift and induced drag.[2]

The vortex-lattice method is based upon distributing discrete horseshoe vortices over the surface of one or more airfoils. These airfoils are subdivided into quadrilateral panels; the spanwise vorticity of each panel is represented by a discrete vortex along its quarter-chord line. Trailing vorticity is accounted for by extending both ends of each vortex as semi-infinite vortex filaments parallel to the free stream velocity. The result is a system of horseshoe vortices, one for each panel. The strengths of these vortices may then be computed by imposing flow tangency at the midpoint of the three-quarter chord of each panel. Once the vortex strengths have been computed, aerodynamic forces are calculated by use of the Kutta-Joukowski theorem, which relates force to the cross-product of the local velocity vector and the circulation vector. The force is computed at the midpoint of the quarter chord of each panel; note that local velocity will be a combination of the free

[2]Note that airfoil *profile* drag—as is the case for the profile drag of all exposed components—is computed by means of empirical expressions based on Reynolds number, skin-friction coefficient, and wetted area.

```
(deflink cruise-fuel ((engines :class basic-thrust-mixin)
                              (vehicle :class gross-properties)
                      (flight-conditions :class flight-conditions)
                      (before :class design-state)
                      (after :class design-state))
        ((fuel-weight
            :documentation "Fuel expended during cruise."
            :low-value 50 :order-of-magnitude 1000 :high-value 20000
            :dimensions "f" :units "lbf")
         (range
            :documentation "Range covered during cruise."
            :low-value 100 :order-of-magnitude 1000 :high-value 10000
            :dimensions "l" :units "NM")
         (endurance :documentation "Time spent during cruise."
                    :low-value 0.5 :order-of-magnitude 5 :high-value 24
                    :dimensions "t" :units "hr"))
        (design-link)
  (:category mission-profile)
  (:documentation "Computes fuel consumed during cruise using the
Breguet range equation."))

(defconstraint (cruise-fuel "Cruise Range") (range "m")
               ((speed@flight-conditions@before "m s-1")
                (sfc@engines@before "s-1")
                (lift-to-drag@vehicle@before "")
                (vehicle-weight@vehicle@before "N")
                (vehicle-weight@vehicle@after "N"))
  "The Breguet range equation."
  (* (/ speed@flight-conditions@before sfc@engines@before)
     lift-to-drag@vehicle@before
     (log vehicle-weight@vehicle@before vehicle-weight@vehicle@after)))

(defconstraint (cruise-fuel "Endurance") (endurance "s")
               ((range "m") (speed@flight-conditions@before "m s-1"))
  "Computes endurance from range and flight speed."
  (/ range speed@flight-conditions@before))

(defconstraint (cruise-fuel "Fuel Weight") (fuel-weight "N")
               ((vehicle-weight@vehicle@before "N")
                (vehicle-weight@vehicle@after "N"))
  "Computes fuel weight as the change in vehicle weight."
  (- vehicle-weight@vehicle@before vehicle-weight@vehicle@after))
```

Figure 4.4: LISP definition for a link-class which implements the Bréguet range equation to compute cruise fuel consumption, **cruise-fuel**.

stream velocity vector and the downwash induced by the horseshoe vortices of all the other panels. The forces on all panels are summed to determine the net aerodynamic forces. The component of force normal to the free stream represents lift; the tangential component yields the induced drag. The panel forces may also be used to compute aerodynamic moments.

The thin-airfoil assumption is applied when using the vortex-lattice method: the airfoils are assumed to have zero thickness, but camber is accounted for when imposing the tangency condition. Thus, the mean-line distributions for the airfoils are used to compute the required normal vectors, but tangency is actually enforced on the planes defined by the leading and trailing edges of the airfoils.

The actual execution of the vortex-lattice method can thus be divided into the following sequence of operations:

1. For each airfoil, subdivide it into panels. For each panel, compute the position of the endpoints of the horseshoe vortex (i.e., at the two ends of the panel's quarter-chord line), and the normal vector at the midpoint of the panel's three-quarter chord.

2. Given the set of airfoils whose aerodynamic forces are to be computed in tandem, for each panel of those airfoils, compute the downwash contributions due to the horseshoe vortices of all panels, at both

   - the midpoint of the panel's three-quarter chord, and
   - the midpoint of the panel's quarter chord.

   The first set of downwash velocities is needed to solve for the vortex strengths based on flow tangency. The second set is used to solve for the aerodynamics forces, once the vortex strengths are known. At this stage, unit strength is assumed for all vortices when performing these computations; because induced velocity is linear in vortex strength, it is possible to set up a system of linear equations for solving for the actual vortex strengths, based on these unit-strength downwash results. Note also that, if compressibility effects are to be considered, the flight Mach number is required to accurately compute the downwash velocities.

3. Using the first set of induced velocities, construct a square matrix, whose dimensions are determined by the total number of panels. Each element, $w_{i,j}$ of this matrix represents the normal component of the downwash on panel i due to the horseshoe vortex associated with panel j, as measured at the midpoint of the three-quarter chord of panel i. Thus, $w_{i,j}$ may be defined as follows:

$$w_{i,j} = \hat{n}_{i,\frac{3}{4}c} \cdot \vec{w}_{i,j}$$

where $\hat{n}_{i,\frac{3}{4}c}$ is the normal at the three-quarter chord of panel i, and $\vec{w}_{i,j}$ is the downwash induced at this point due to the horseshoe vortex on panel j.[3]

4. Given an angle of attack and a free-stream speed, this matrix is used to set up a system of linear equations representing the satisfaction of the flow-tangency conditions at the panel three-quarter chords. By factoring the matrix and solving the system based on the free-stream normal components, the required vortex strengths may be computed.

5. Based on free-stream density and the vortex strengths, flow forces on each panel are computed using the Kutta-Joukowski theorem,

$$\vec{F} = \rho \vec{U} \times \vec{\Gamma}$$

where ρ is fluid density, $\vec{\Gamma}$ is the panel's circulation vector (whose magnitude is equal to the strength of the horseshoe vortex associated with the panel), and $\vec{U}$ is the local velocity vector for the panel. This velocity vector is computed by summing the free-stream velocity vector with the downwash velocities induced by the vortices of all other panels, as measured at the midpoint of the panel's quarter-chord.

6. Panel forces are summed to determine the net lift and induced drag for each airfoil. Moments may also be computed, using the panel quarter-chord midpoints as the assumed center of pressure for each panel.

Note that, by assuming unit free-stream speed and density, and by correcting for airfoil reference area and chord, the vortex-lattice method may be used to directly compute lift, induced drag, and moment *coefficients*, rather than actual aerodynamic forces and moments. In implementing the vortex-lattice method for *Rubber Airplane*, it was this alternate approach which was taken.

Furthermore, note that Step 1 of this algorithm is dependent only on airfoil geometry. Furthermore, Steps 2 and 3 depend only on free stream Mach number and selection of the airfoils which are to be simultaneously solved. It is only upon reaching Step 4 that the free stream conditions are required. In fact, the inversion of the downwash matrix can take place prior to Step 4, since this operation is also independent of free stream conditions. This means that substantial computational efficiency may be gained by maintaining a record of previous computations. In cases where only the free stream conditions are modified, the algorithm may proceed directly to Step 4, and even bypass the computationally-expensive matrix inversion step, assuming that the results of previous calculations have been stored.

[3]Note that the downwash vector may be computed by means of the Biot-Savart law, which may be suitably modified—by means of the linear Prandtl-Glauert transformation—for the effects of subsonic compressibility. See, for example, Reference [22].

Even in the situation where one airfoil has been relocated with respect to the others, some small savings in computational effort is possible.

The *Rubber Airplane* implementation of the vortex-lattice method has therefore taken advantage of the re-usability of these computations. Various data structures for representing the panels of a single airfoil, and for storing the downwash values for a collection of airfoils, have been implemented. To illustrate the interface between *Rubber Airplane* and the LISP code which implements the vortex-lattice method, a design link which performs the method on a single airfoil is presented in Figures 4.5—4.7.

Figure 4.5 presents the definition of a link-class `airfoil-vortex-lattice`, which defines attributes for the various aerodynamic coefficients to be computed for the airfoil. The values for the `spanwise` and `chordwise` attributes are used when subdividing the airfoil into panels. By representing these parameters as attributes of the link-class, they come under the direct control of the user. Thus, in the early stages of a design, a relatively small number of panels can be used, in order to reduce computation time while attempting to size the airfoil. Later, the number of panels may be increased in order to improve the accuracy of the results. Additionally, the `Mach-threshold` attribute is provided to give the user control over when corrections for subsonic compressibility should be employed.

The actual constraint which performs the vortex-lattice analysis appears in Figure 4.6. This constraint has multiple output parameters, since the vortex-lattice method enables the simultaneous computation of three different aerodynamic coefficients.[4] Because it is a multiple-output constraint, it is also necessarily a `:one-way` constraint. It may at first appear that using an uninvertible constraint for computing airfoil aerodynamic forces is somewhat restrictive. It is the case, however, inverse airfoil design is still a matter of active research in aerodynamics. Furthermore, since most of the vortex-lattice results (induced drag, moments, pressure distribution, etc.) are rarely specified or imposed by other constraints, it is not necessary to invert the vortex-lattice calculation in practice, even if it were possible.

The input parameters for this constraint are the paneling parameters, the Mach number threshold, the flight conditions (specifically, angle of attack and Mach number), and a set of parameters which describe the airfoil's geometry: chord length, position of the leading edge, camber distribution, and incidence angle for both the root and tip cross-sections. In addition, various reference parameters, such as wing area and magnitude and position of the mean aerodynamic chord, are required. Based on these values, the first step is to convert the paneling parameters to integers: internally, the values of all scalar attributes are stored as single-precision floating-point numbers. The Common Lisp `floor` function

[4] Actually, the method may used to calculate other aerodynamic coefficients, such as lift-curve slope, as well. For brevity, discussion of the means by which these other coefficients may be computed has been omitted.

```
(deflink airfoil-vortex-lattice
                ((airfoil :class basic-horizontal-airfoil)
                 (flight-conditions :class flight-conditions))
        ((spanwise :documentation
             "Number of spanwise intervals for dividing the airfoil."
             :low-value 1 :order-of-magnitude 5 :high-value 10
             :dimensions "")
         (chordwise :documentation
             "Number of chordwise intervals for dividing the airfoil."
             :low-value 1 :order-of-magnitude 3 :high-value 6
             :dimensions "")
         (panels :documentation
             "Total number of panels for the Vortex-Lattice analysis."
             :low-value 1 :order-of-magnitude 15 :high-value 60
             :dimensions "")
         ("Mach-threshold" :documentation
             "Maximum Mach number for ignoring compressibility."
             :low-value 0 :order-of-magnitude 0.5 :high-value 0.7
             :dimensions "")
         ("C_L" :documentation "Lift coefficient for the airfoil."
                :state-dependent? t
                :low-value 0.01 :order-of-magnitude 0.5 :high-value 1.0
                :dimensions "")
         ("C_Di" :documentation "Drag coefficient for the airfoil."
                 :state-dependent? t
                 :low-value 0.001 :order-of-magnitude 0.01
                 :high-value 0.2 :dimensions "")
         ("C_M" :documentation "Moment coefficient for the airfoil."
                :state-dependent? t
                :low-value -1.0 :order-of-magnitude -0.001
                :high-value 1.0 :dimensions "")
         ("L-over-Di" :documentation "Lift to Drag ratio."
                      :state-dependent? t
                      :low-value 1 :order-of-magnitude 10
                      :high-value 100 :dimensions ""))
        (design-link)
  (:category aerodynamics vortex-lattice)
  (:documentation "Computes lift, drag, and moment coefficients
                   for an airfoil, using the Vortex-Lattice method."))
```

Figure 4.5: LISP definition for a link-class which implements the vortex-lattice method for computing the aerodynamic coefficients of a single airfoil, `airfoil-vortex-lattice`.

```
(defconstraint (airfoil-vortex-lattice "Vortex Lattice" :one-way)
               ((c_l "") (c_di "") (c_m "") (l-over-di ""))
               ((spanwise "") (chordwise "") (mach-threshold "")
                (alpha@flight-conditions "rad")
                (mach@flight-conditions "")
                (root-mean-line@airfoil "")
                (root-mean-line-scaling@airfoil "")
                (root-chord@airfoil "m") (root-incidence@airfoil "m")
                (tip-mean-line@airfoil "")
                (tip-mean-line-scaling@airfoil "")
                (tip-chord@airfoil "m") (tip-incidence@airfoil "m")
                (mac-x@airfoil "m" x-ref) (mac-y@airfoil "m" y-ref)
                (mac-z@airfoil "m" z-ref)
                (mac@airfoil "m" c-ref) (wing-area@airfoil "m2" S-ref)
                (root-x_le@airfoil "m") (tip-x_le@airfoil "m")
                (root-y_le@airfoil "m") (tip-y_le@airfoil "m")
                       (root-z_le@airfoil "m") (tip-z_le@airfoil "m"))
  "Computes the lift, drag, and moment coefficient using the
Vortex-Lattice method."
  (declare (special *mach-threshhold*))
  (setq spanwise (floor spanwise) chordwise (floor chordwise))
  (let* ((mach (if (> mach@flight-conditions mach-threshhold)
                   mach@flight-conditions
                   0f0))
         (airfoil (vl::make-airfoil root-x_le@airfoil root-y_le@airfoil
                                    root-z_le@airfoil root-chord@airfoil
                                    root-incidence@airfoil
                                    root-mean-line@airfoil
                                    root-mean-line-scaling@airfoil
                                    tip-x_le@airfoil tip-y_le@airfoil
                                    tip-z_le@airfoil tip-chord@airfoil
                                    tip-incidence@airfoil
                                    tip-mean-line@airfoil
                                    tip-mean-line-scaling@airfoil
                                    x-ref y-ref z-ref c-ref S-ref
                                    spanwise chordwise mach t))
         (problem (vl::make-problem airfoil)))
    (vl::vl-solve problem alpha-eff)
    (multiple-value-bind (cl cdi cm l-over-di)
                         (vl::aero-coeffs airfoil alpha@flight-conditions)
      (values cl cdi cm l-over-di))))
```

Figure 4.6: LISP definition for the constraint which implements the vortex-lattice method for link-class **airfoil-vortex-lattice**.

is used for this purpose. Next, the free stream Mach number is compared to the value of the **Mach-threshold** attribute to determine the effective Mach number for the analysis. If the Mach number is below the threshold, the effective Mach number is zero; otherwise, the actual Mach number value is used.

The next step in the analysis is to divide the airfoil into panels. This is the role of the **vl::make-airfoil**[5] function, which takes as inputs the paneling parameters and the various airfoil geometry parameters, and creates a data structure which stores the coordinates of the corresponding panels, their horseshoe vortices, and the normal vectors at the midpoints of the panels' three-quarter-chord lines. It is therefore responsible for performing Step 1 in the vortex-lattice algorithm enumerated above. Note, however, that if the function is called with a set of arguments identical to an earlier set of arguments, no new data structure is constructed; the previous result is returned.

The next operation performed by this constraint is a call to the **vl::make-problem** function. This function takes as its arguments one or more "airfoil" data structures, as created by the **vl::make-airfoil** function. In this case, only one airfoil is being solved for, so only one argument is passed to the **vl::make-problem** function. This function is responsible for calculating the downwash vectors, as described in Step 2 above, and for constructing the matrix of normal components, as in Step 3, as well as factoring it. As with **vl::make-airfoil**, the **vl::make-problem** function keeps a record of its previous arguments. Whenever a set of arguments is repeated, the previously-constructed data structures are returned; no new computations are necessary.

Next the **vl::vl-solve** function is called to solve for the vortex strengths; its arguments are the vehicle angle of attack and the "problem" data structure returned by the previous call to **vl::make-problem**. Note that the resulting values for the vortex strengths are stored in the "airfoil" data structures which are associated with the "problem" data structure. This makes it possible to solve for the aerodynamic forces on individual airfoils, without the need to refer to the intermediate "problem" data structure. Indeed, this is the role of the subsequent call to the **vl::aero-coeffs** function, which returns multiple values representing these aerodynamic coefficients. These multiple values are themselves returned by the *Rubber Airplane* constraint. Figure 4.7 lists some of the auxiliary constraints associated with the **airfoil-vortex-lattice** link-class, including those which relate these aerodynamic coefficients to the actual forces on the airfoil.

Above, mention has been made of attributes whose values represent the camber of an airfoil cross-section. In implementing the vortex-lattice method, an efficient means of rep-

[5]Note that the code which implements the vortex-lattice method has been developed as an independent program, in its own LISP package (see Reference [35]). The name for this package is "**vl**". Thus, when called from *Rubber Airplane* constraints, which are defined in a package of their own, functions in the vortex-lattice package must be preceded by the "**vl::**" prefix.

```
(defconstraint (airfoil-vortex-lattice "Number of Panels") (panels "")
               ((spanwise "") (chordwise ""))
  "Computes the total number of panels for the Vortex-Lattice analysis."
  (* spanwise chordwise))

(defconstraint (airfoil-vortex-lattice "Airfoil Lift") (lift@airfoil "N")
               ((c_l "")
                (density@flight-conditions "kg m-3" rho)
                (speed@flight-conditions "m s-1" V)
                (wing-area@airfoil "m2" S))
  "Computes the lift on the airfoil, based on lift coefficient."
  (* c_l 1/2 rho V V S))

(defconstraint (airfoil-vortex-lattice "Airfoil Induced Drag")
               (drag-induced@airfoil "N")
               ((c_di "")
                (density@flight-conditions "kg m-3" rho)
                (speed@flight-conditions "m s-1" V)
                (wing-area@airfoil "m2" S))
  "Computes the lift on the airfoil, based on lift coefficient."
  (* c_di 1/2 rho V V S))

(defconstraint (airfoil-vortex-lattice "Airfoil Aerodynamic Moment")
               (moment_mac@airfoil "N")
               ((c_m "")
                (density@flight-conditions "kg m-3" rho)
                (speed@flight-conditions "m s-1" V)
                (wing-area@airfoil "m2" S)
                (mean-aerodynamic-chord@airfoil "m" c)
  "Computes the lift on the airfoil, based on lift coefficient."
  (* c_m 1/2 rho V V S c))
```

Figure 4.7: LISP definition for the constraints which support various auxiliary calculations for the **airfoil-vortex-lattice** link-class.

resenting cross-section mean-line distribution was required. For this reason, a LISP-based database of standard NACA thickness and camber distribution has been developed for use in *Rubber Airplane*, based on the distributions listed in Reference [1]. Indeed, as indicated in Table 4.3, the `basic-airfoil` component class, which serves as a superclass for the `basic-wing`, `basic-horizontal`, and `basic-vertical` component-classes employed in this test-case design, includes as one of its own superclasses the `NACA-section-pair` component-class. This component-class defines discrete attributes for representing the thickness and mean-line distributions for both the root and tip cross-sections of an airfoil. These discrete attributes point to data structures which store the coordinates of the corresponding distributions, which are used by both the vortex-lattice functions and *Rubber Airplane*'s geometry display routines.[6] In addition, `NACA-section-pair` also provides a scalar attribute for scaling the magnitude of the mean-line distribution, as required for the "a" series of mean-line distributions (see Reference [1]). Of course, only the mean-line distribution is relevant to the aerodynamic computations discussed here, which are based on thin-airfoil theory. Airfoil profile drag is based on thickness, though, and the `wing-profile-drag`, `horizontal-profile-drag`, and `vertical-profile-drag` link-classes mentioned in Table 4.6 all rely on properties of the data structures representing NACA thickness distributions to estimate airfoil profile drag.

Finally, note from Table 4.6 that it is actually the `two-airfoil-vortex-lattice` link-class which is used to compute airfoil aerodynamics for this test case. In this way, the aerodynamic forces on both the wing and the horizontal stabilizer may be computed simultaneously. The advantage to this approach is that it automatically accounts for the aerodynamic interference of the two lifting surfaces.[7]

This `two-airfoil-vortex-lattice` link-class is not fundamentally different from the `airfoil-vortex-lattice` link-class presented above: the only major difference is that a second set of paneling and geometry parameters are required so that two "airfoil" data structures may be constructed. These two data structures are then together passed as arguments to the `vl::make-problem` function, the result of which is passed on to `vl::vl-solve`. Finally, two calls to `vl::aero-coeffs` are required, one for each airfoil, and a larger set of results must be returned as multiple-values by the constraint, for the correspondingly larger set of output parameters.

[6] Note that cubic spline functions are also provided for performing interpolations on these coordinate distributions.

[7] Of course, the interference effects of the other components, such as the fuselage and the radome, are still neglected.

4.2.4 Observations

As indicated above, it was during the development of this first test case that the need for design states was first recognized. Similarly, support for unidirectional, multiple-output constraints resulted from the requirement for such constraints while implementing the vortex-lattice method, discussed above.

Furthermore, over the course of implementing this test case, what appears to be a general feature of the component-modeling approach has been discovered. Specifically, it has been observed that modularity is improved by minimizing the number of attributes which are associated with component-classes. In practice, this means limiting component-class attributes to those needed to describe its geometry (e.g., position and dimensions) and gross properties (e.g., lift, drag, and weight). The former set of parameters is needed to define the component's geometry, so that it may be displayed. The gross-property attributes provide link-classes with a uniform interface to the component-classes, but require that any auxiliary parameters be supplied by the link-class, rather than the component-class. It is this transfer of such auxiliary parameters (e.g., aerodynamic coefficients, angle of attack, sideslip, structural load factor) to the link-classes which makes the component-classes more general-purpose and, therefore, more modular. Indeed, component-classes generally do not specify the means for computing their gross-property attributes, but rely on link-classes for these calculations, as well.

By relying on link-classes to provide auxiliary parameters and compute component gross properties, it becomes very easy for the specific design analyses employed to be altered, simply by replacing one or more link-classes. For example, a design project might initially include a design link which computes airfoil aerodynamics based on an assumed value for the airfoil lift-curve slope. Later, when improved accuracy is desired, this link might be replaced by an instance of the `airfoil-vortex-lattice` link-class, introduced above. There is no need to replace the component which represents the airfoil; only the design links need be changed. Thus, while design components are rather general-purpose, link-classes are used to implement very specific analyses. Because there is a standard interface between link- and component-classes (i.e., component gross properties), however, exchanging one design link for another which performs a similar type of analysis is a straightforward operation.

4.3 Subsonic Commercial Transport

4.3.1 Specifications

The second test case concerns the design of a subsonic commercial transport for a regional airline. The design calls for a vehicle capable of carrying 50 passengers (at a weight allocation of 300 lbs. per passenger) over a maximum range of 500 nautical miles; minimum-length

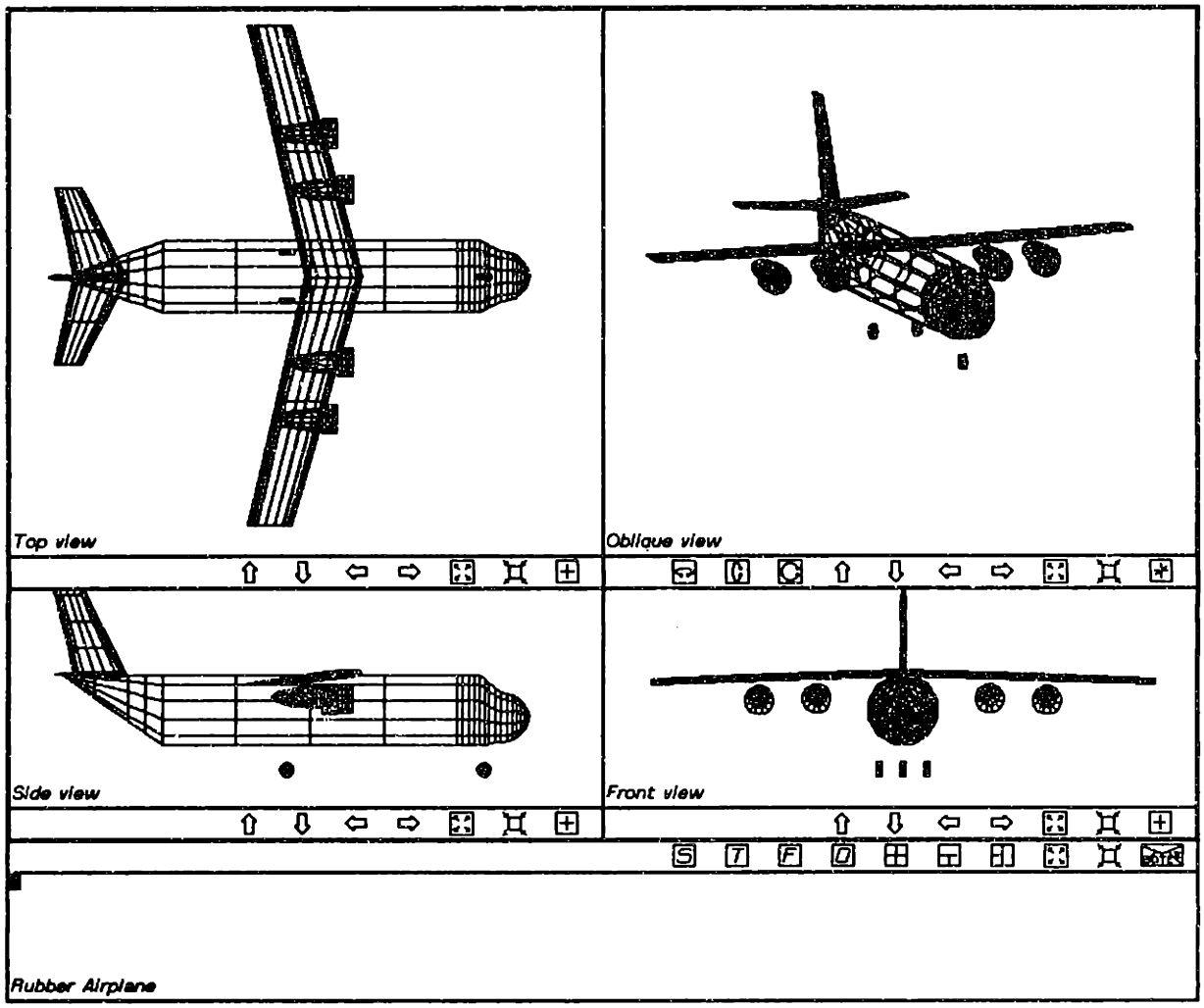

Figure 4.8: Screen image depicting the completed design for the subsonic transport test case.

runway performance is also desired. Furthermore, due to engine maintenance issues, the power plant for this vehicle must be the Avco Lycoming 502D turbofan engine.[8]

In meeting these design requirements, then, the primary focus is on selecting the appropriate number of engines for balancing cruise drag and returning acceptable takeoff and landing performance. The preliminary results of this study—based on only token efforts at design optimization—called for an aircraft which employs four of these engines, yielding an estimated gross takeoff weight of 43,000 lbs., and takeoff and landing distances of approximately 4200 and 4500 feet, respectively.

[8]The Avco Lycoming 502D provides a static thrust of 27,500 Newtons, and a cruise thrust of 6000 Newtons at an altitude of 33,000 ft. and a flight Mach number of 0.8.

4.3.2 Problem Representation

The motivation in selecting this design task as the second test case was to demonstrate the reusability of the class definitions developed for the first test case. As such, very few new component- and link-classes were needed to implement this second test case. Indeed, because both the vehicle geometry and mission profile are simpler, this design employed only 15 components, 5 design states and 38 links, as compared to 16 components, 10 design states, and 39 links for the first test case. A screen image displaying the final geometry of the design's components is presented in Figure 4.8. Note that this second test-case design employs a total of 514 attributes and 360 constraints.

The component-classes used in representing this test-case design are listed in Table 4.9. As indicated above, this set of classes is practically a subset of the component-classes used for the first test case. The only new component-classes employed in this design are the **turbojet-pair** and **passenger** classes. Note that two instances of the **turbojet-pair** component-class are used to represent the design's four engines; this class plays a role similar to that of the **turboprop-pair** component-class employed by the first test-case design. Statistics for these component-classes are presented in Table 4.10. The most commonly-occurring component-classes are listed in Table 4.11; the only noteworthy change here from the component-classes listed in Table 4.3 is the addition of the **person-mixin** component-class, which serves as a superclass for both the **crew** and **passenger** classes employed in this second design.

Table 4.12 lists the design states instantiated for this design. As was the case with the previous design, all design states used here are instances of the base **design-state** class.

Because the analyses required to implement this test case are essentially identical to those required for the first test case, there is very little difference in the sets of design links employed by the two designs. The link-classes used for this second design are listed in Tables 4.13 and 4.14. Notable changes include the replacement of class **turboprop-wing-attachment** by the **turbojet-wing-attachment** link-class, and the addition of two instances of the **simple-thrust-model** link-class, which is used to compute the propulsive properties of a set of engines, based on the properties of a single engine. The only "compatibility" issue encountered while modeling this second test case concerned the representation of engine specific fuel consumption (SFC). For the turboprop analyses of the first test case, an SFC which scaled with engine power was expected; the constraints required for analyzing the turbojet engines required by the current design task were dependent upon the more conventional form of specific fuel consumption, in which engine thrust is directly related to fuel weight flow rate. As the conversion between these two parameters is quite straightforward, this discrepancy was easily resolved.

Component Class	Description
basic-fuselage	Fuselage geometry for a transport-type aircraft.
basic-wing	Airfoil geometry for a swept, tapered wing, for which different cross-sections may be specified at the root and tip.
basic-vertical	Airfoil geometry for a vertical stabilizer.
basic-horizontal	Airfoil geometry for a horizontal stabilizer.
turbojet-pair	Geometry for a pair of wing-mounted turbojet engines.
fuel-system	Component representation of fuel and fuel tanks
forward-landing-gear	A simple forward landing gear (tire only).
rear-landing-gear	A pair of rear landing gear tires.
controls	Component representation for aircraft controls.
systems	Component representation for aircraft support systems.
air-conditioning	Component representation for air-conditioning system.
avionics	Component representation for aircraft control avionics.
crew	Component representation of aircraft crew.
passengers	Component representation of aircraft passengers.

Table 4.9: Instantiated component-classes for the subsonic transport test case.

Component Class	Superclasses	Attributes	Constraints	Instances
basic-fuselage	4	25	14	1
basic-wing	11	51	28	1
basic-vertical	11	46	26	1
basic-horizontal	11	51	28	1
turbojet-pair	2	26	10	2
fuel-system	2	19	11	1
forward-landing-gear	4	17	9	1
rear-landing-gear	4	18	9	1
controls	2	11	7	1
systems	2	17	10	1
air-conditioning	2	17	10	1
avionics	2	17	10	1
crew	3	18	10	1
passengers	3	18	10	1
Total:		377	202	15

Table 4.10: Class statistics for the instantiated component-classes of the subsonic transport test case.

Component Class	Subclasses	Description
design-component	19	Base component-class.
basic-drag-mixin	12	Provides a general-purpose **drag** attribute.
point-mass-component	10	Equates component reference and center-of-gravity positions.
basic-lift-mixin	8	Provides a general-purpose **lift** attribute.
basic-aerodynamics-mixin	7	Combines **basic-lift-mixin** and **basic-drag-mixin**.
basic-section	7	Provides general airfoil cross-section attributes.
NACA-section-pair	6	Provides attributes to represent NACA-airfoil cross sections at both the root and tip of an airfoil.
airfoil-planform	6	Provides airfoil planform parameters.
induced-drag-mixin	6	Breaks down component drag into induced drag and profile drag components.
basic-airfoil	5	Combines **NACA-section-pair** and **airfoil-planform**.
horizontal-planform-mixin	3	Represents a horizontally-oriented airfoil planform.
vertical-planform-mixin	2	Represents a vertically-oriented airfoil planform.
basic-horizontal-airfoil	2	Combines **basic-airfoil** and **horizontal-planform-mixin**.
basic-landing-gear	2	Provides general landing-gear attributes.
person-mixin	2	Provides general attributes for representing a collection of persons.

Table 4.11: Most commonly occurring component-classes among the instantiated component-classes of the subsonic transport test case.

Design State Name	Description
"Takeoff"	End of takeoff run.
"Intermediate Climb"	Intermediate flight condition during climb to cruising altitude.
"Cruise"	Beginning of cruise flight segment.
"Begin Descent"	Start of descent for landing.
"Landing"	Flight condition for vehicle landing.

Table 4.12: Instances of class **design-state** for the subsonic transport test case.

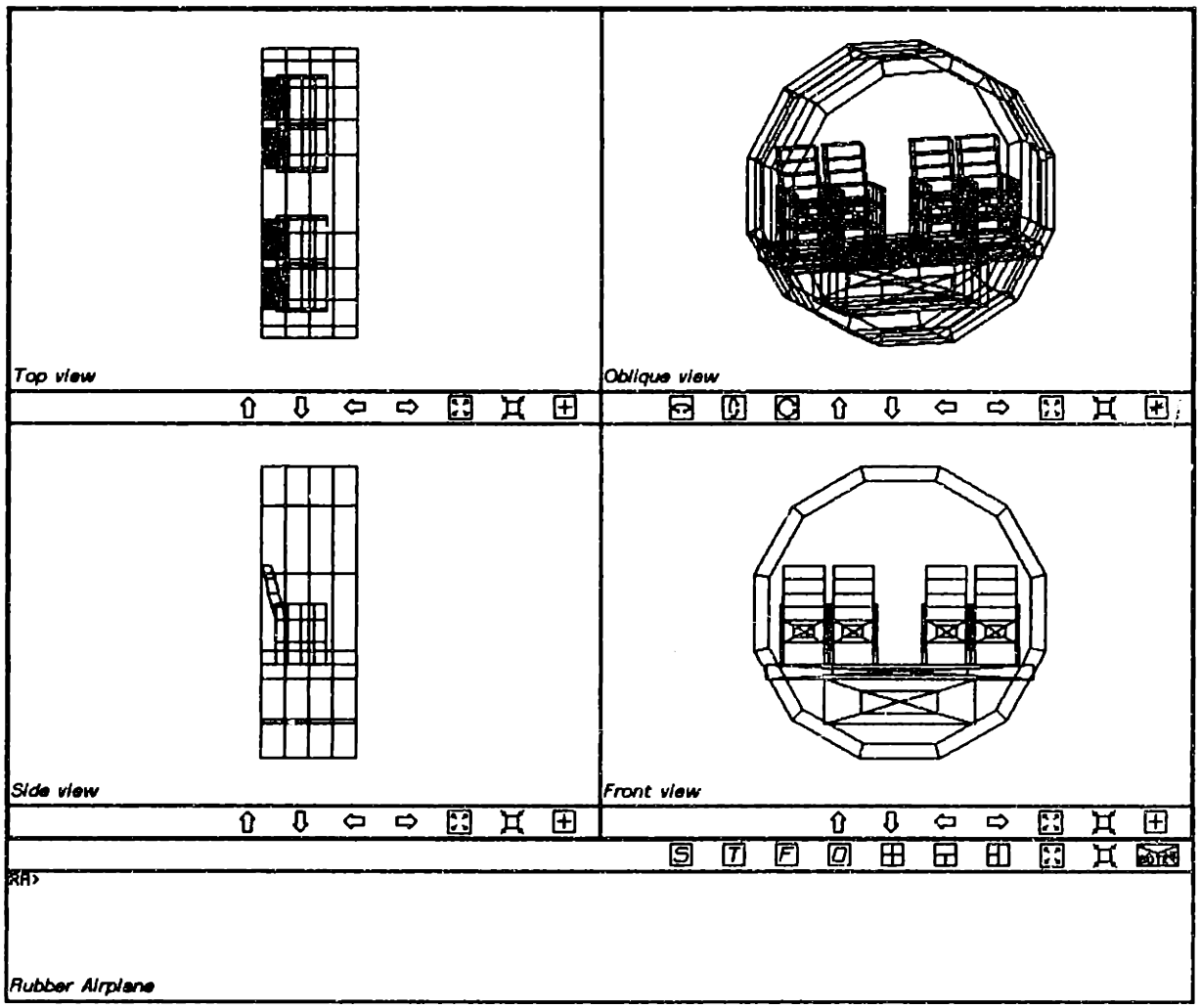

Figure 4.9: Screen image depicting the fuselage cross-section design for the subsonic transport test case.

Link Class	Description
flight-conditions	State-dependent description of the flight conditions.
gross-properties	Net weight, thrust, and aerodynamic forces on the vehicle.
wing-attachment	Sets the y- and z-coordinates of the wing based upon corresponding fuselage dimensions and coordinates.
tail-attachment	Attaches the horizontal and vertical stabilizers to the rear of the fuselage.
turbojet-wing-attachment	Attaches a symmetric pair of engines to the wings, accounting for airfoil geometry and inlet clearance.
cockpit-attachment	Positions generic objects with respect to the cockpit portion of the fuselage.
cabin-attachment	Positions generic objects with respect to the cabin portion of the fuselage.
takeoff-distance	Computes the required takeoff distance for the vehicle.
landing-distance	Computes the required landing distance for the vehicle.
fuselage-weight-model	Computes the weight of the fuselage, based on dimensions and net vehicle weight.
wing-weight-model	Computes the weight of the wing, based on dimensions and net vehicle weight.
empennage-weight-model	Computes the weight of the horizontal and vertical stabilizers, based on dimensions and net vehicle weight.
fuel-tank-weight-model	Computes the weight of the fuel tanks, based on total fuel weight.
controls-weight-model	Computes the weight of the aircraft control systems, based on net vehicle weight.
systems-weight-model	Computes the weight of the support systems, based on net vehicle weight.
forward-gear-weight-model	Computes the weight of the forward landing gear, based on net vehicle weight.
rear-gear-weight-model	Computes the weight of the rear landing gear, based on net vehicle weight.
person-weight-model	Computes the weight of a group of persons.
simple-takeoff-fuel	Equates takeoff vehicle weight to vehicle gross weight.
turbojet-cruise-fuel	Computes the fuel required for a single cruise segment
glide-fuel	Equates the vehicle weights for two design states.
total-fuel-weight	Sums the fuel weights required for each flight segment.
simple-thrust-model	Computes the properties of a set of engines based on those of a single engine.

Table 4.13: Instantiated link-classes for the subsonic transport test case.

Link Class	Description
`two-airfoil-vortex-lattice`	Computes the lift and induced drag for a pair of airfoils.
`wing-profile-drag`	Computes wing profile drag, based on wetted area.
`vertical-profile-drag`	Computes vertical stabilizer profile drag, based on wetted area.
`horizontal-profile-drag`	Computes horizontal stabilizer profile drag, based on wetted area.
`fuselage-profile-drag`	Computes fuselage profile drag, based on wetted area.
`forward-gear-profile-drag`	Computes profile drag for the forward landing gear, based on wetted area.
`rear-gear-profile-drag`	Computes profile drag for the rear landing gear, based on wetted area.

Table 4.14: Instantiated link-classes for the subsonic transport test case (continued).

Link Class	Superclasses	Attributes	Constraints	Instances
flight-conditions	2	11	6	1
gross-properties	1	14	13	1
wing-attachment	1	1	2	1
tail-attachment	3	0	6	1
turbojet-wing-attachment	1	2	3	2
cockpit-attachment	1	0	3	3
cabin-attachment	1	0	3	2
takeoff-distance	1	14	10	1
landing-distance	1	12	6	1
fuselage-weight-model	1	0	1	1
wing-weight-model	1	1	2	1
empennage-weight-model	1	4	5	1
fuel-tank-weight-model	1	0	1	1
controls-weight-model	1	0	1	1
systems-weight-model	1	0	1	1
forward-gear-weight-model	1	0	1	1
rear-gear-weight-model	1	0	1	1
person-weight-model	1	1	1	2
simple-takeoff-fuel	1	0	1	1
turbojet-cruise-fuel	3	3	3	1
glide-fuel	1	0	1	3
total-fuel-weight	1	3	3	1
simple-thrust-model	1	4	4	2
two-airfoil-vortex-lattice	1	27	20	1
wing-profile-drag	3	5	6	1
vertical-profile-drag	3	5	6	1
horizontal-profile-drag	3	5	6	1
fuselage-profile-drag	3	6	7	1
forward-gear-profile-drag	3	6	7	1
rear-gear-profile-drag	3	6	7	1
Total:		137	158	38

Table 4.15: Class statistics for the instantiated link-classes of the subsonic transport test case.

Link Class	Subclasses	Description
design-link	30	Base link-class.
basic-profile-drag-mixin	8	General-purpose induced drag model, based on wetted area and skin-friction coefficient.
body-profile-drag	3	Specializes **basic-profile-drag-mixin** for non-lifting bodies.
airfoil-profile-drag	3	Specializes **basic-profile-drag-mixin** for airfoils.
fuel-consumption-mixin	2	Provides a general-purpose **fuel-weight** attribute.

Table 4.16: Most commonly occurring link-classes among the instantiated link-classes of the subsonic transport test case.

Component Class	Description
fuselage-section	Geometry for the cross-section, including positioning of the cabin floor.
chair	Geometry of a cabin-section chair.
cargo-box	Geometry of a box for stowing luggage and other cargo.

Table 4.17: Instantiated component-classes for the fuselage cross-section design of the subsonic transport test case.

Component Class	Superclasses	Attributes	Constraints	Instances
`fuselage-section`	1	23	8	1
`chair`	1	24	8	4
`cargo-box`	1	14	4	1
Total:		133	44	6

Table 4.18: Class statistics for the instantiated component-classes of the fuselage cross-section design for the subsonic transport test case.

Class statistics for the link-classes instantiated for the second test case appear in Table 4.15. Once more, sharing of link-classes through inheritance is not common; those link-classes which are most often inherited by the design's instantiated link-classes are listed in Table 4.16.

4.3.3 Design Analysis

As is evident from the tables of component- and link-classes presented in the preceding section, the design analyses utilized in this second test case are almost identical to those utilized for the first design problem. Cruise drag and runway performance were driving factors in selecting the number of engines. Maintaining level flight in cruise was the criterion by which wing size was determined. Lifting-surface aerodynamics were computed using the vortex-lattice method, as described above. Vehicle weight was determined by means of an automatically-constructed iteration loop, with aircraft gross weight as the loop variable.

One new approach was introduced for sizing the fuselage, however. Specifically, a set of component-classes was defined which enabled the design of the cabin interior, based on a representative cross-section. Component-classes representing the passenger seats, the fuselage cross-section, and a cargo box were defined; link-classes for positioning these components relative to one another were also provided. This *Rubber Airplane* design employs 6 components, no states, and 6 links, with a total of 143 attributes and 93 constraints. The final design for the cabin cross-section appears in Figure 4.9.

A listing of the specific component-classes created for this design task is given in Table 4.17; class statistics for these component-classes appear in Table 4.18. Listings of link-class descriptions and their statistics appear in Tables 4.19 and 4.20, respectively. Note that three instances of the `identical-neighboring-chairs` link-class have been employed to position the chairs with respect to one another; this link-class defines a `chair-separation` attribute whose value may be set to zero to indicate adjacent chairs, or made non-zero to represent the aisle width.

In employing these component- and link-classes to compute the cross-section dimensions,

Link Class	Description
identical-neighboring-chairs	Constrains the spacing between a pair of chairs, and equates their dimensions.
row-of-chairs	Defines a row of chairs, based on the leftmost and rightmost chairs in the row.
row-attachment	Attaches a row of chairs to the floor of a fuselage cross-section.
cargo-box-attachment	Attaches a cargo box below the floor of a fuselage cross-section.

Table 4.19: Instantiated link-classes for the fuselage cross-section design of the subsonic transport test case.

Link Class	Superclasses	Attributes	Constraints	Instances
identical-neighboring-chairs	3	1	12	3
row-of-chairs	1	7	6	1
row-attachment	1	0	4	1
cargo-box-attachment	1	0	3	1
Total:		10	49	6

Table 4.20: Class statistics for the instantiated link-classes of the fuselage cross-section design for the subsonic transport test case.

the chair and aisle dimensions were first input as fixed values, based on recommended minimum dimensions as listed in Reference [38]. In attempting to derive acceptable values for both fuselage length and cross-section dimensions, a total of four chairs per row was selected. From here, floor position, fuselage height, and fuselage width were simultaneously adjusted, and the results were observed in the geometry display portion of the *Rubber Airplane* interface (see Appendix C), until an acceptable geometry was found. Ultimately, both fuselage height and width were set at ten feet, leaving room for a $1\frac{1}{2}$ ft. $\times$ 5 ft. cargo box beneath the cabin floor. Based on chair dimensions and required legroom, the required length for the fuselage cabin—in this case, 41 ft.—was computed based on the minimum number of rows required to seat the desired number of passengers.

4.3.4 Observations

As indicated above, this test case was intended to demonstrate the ability of the component-modeling approach to support code reusability. In this regard, the exercise was successful, insofar as only a minimum of effort was required to modify the class library in order to

support the representation of this second design task.

Of somewhat greater interest, then, is the use here of an auxiliary *Rubber Airplane* design to support the sizing of the fuselage in the primary design. In this case, the auxiliary design is used to perform the purely geometrical task of laying out the internal seating arrangement for the fuselage cabin section. A simple set of component- and link-classes has been provided (there is no need for the use of design states in this auxiliary design); note, however, that the actual task of selecting the cross-section dimensions depended upon the user observing the effect on the geometry display of variations in the corresponding dimensional parameters. While it is certainly possible to define a set of constraints for computing the minimum fuselage height and width which will accommodate the desired seating arrangement, it is much simpler for the designer to modify the component attributes by hand, and establish a satisfactory fit via visual inspection. This observation helps demonstrates the utility of the attribute-driven geometry display: not only does it provide immediate, graphic feedback on the consequences of various design decisions, but it also supports the (albeit indirect) implementation of certain design tasks which are difficult to codify.

Recall that a primary objective of this exercise was the determination of the number of engines to be employed for the vehicle design. Similarly, the auxiliary design was introduced to evaluate various internal layouts, based upon several factors, including the number of seats per row. These two parameters—number of engines and number of seats per row—represent configuration decisions; specifically, they represent the number of two types of components which are to be incorporated into the design. In *Rubber Airplane*, however, component incorporation is implemented via component-class instantiation, which can only be performed by the user. It is therefore impossible to represent these parameters as design variables—i.e., as component- or link-class attributes—since there is no mechanism for instantiating classes based on attribute values. This observation presents a fundamental shortcoming of the component-modeling approach adopted for *Rubber Airplane*: while the program provides support for (partially) automating the sizing process, all configuration decisions must be made externally, by the user.

4.4 Small-Payload Launch Vehicle

4.4.1 Specifications

In order to investigate the application of *Rubber Airplane* to alternative problem domains, the third test case concerned the design of a small-payload launch vehicle. Specifically, a multi-stage craft capable of launching a payload of 1000 lbs. to low-earth orbit (LEO) was designed. The simplified analyses employed to carry out this design task yielded a two-stage rocket with an overall length of 30 feet, and a launch weight of 35,900 lbs.

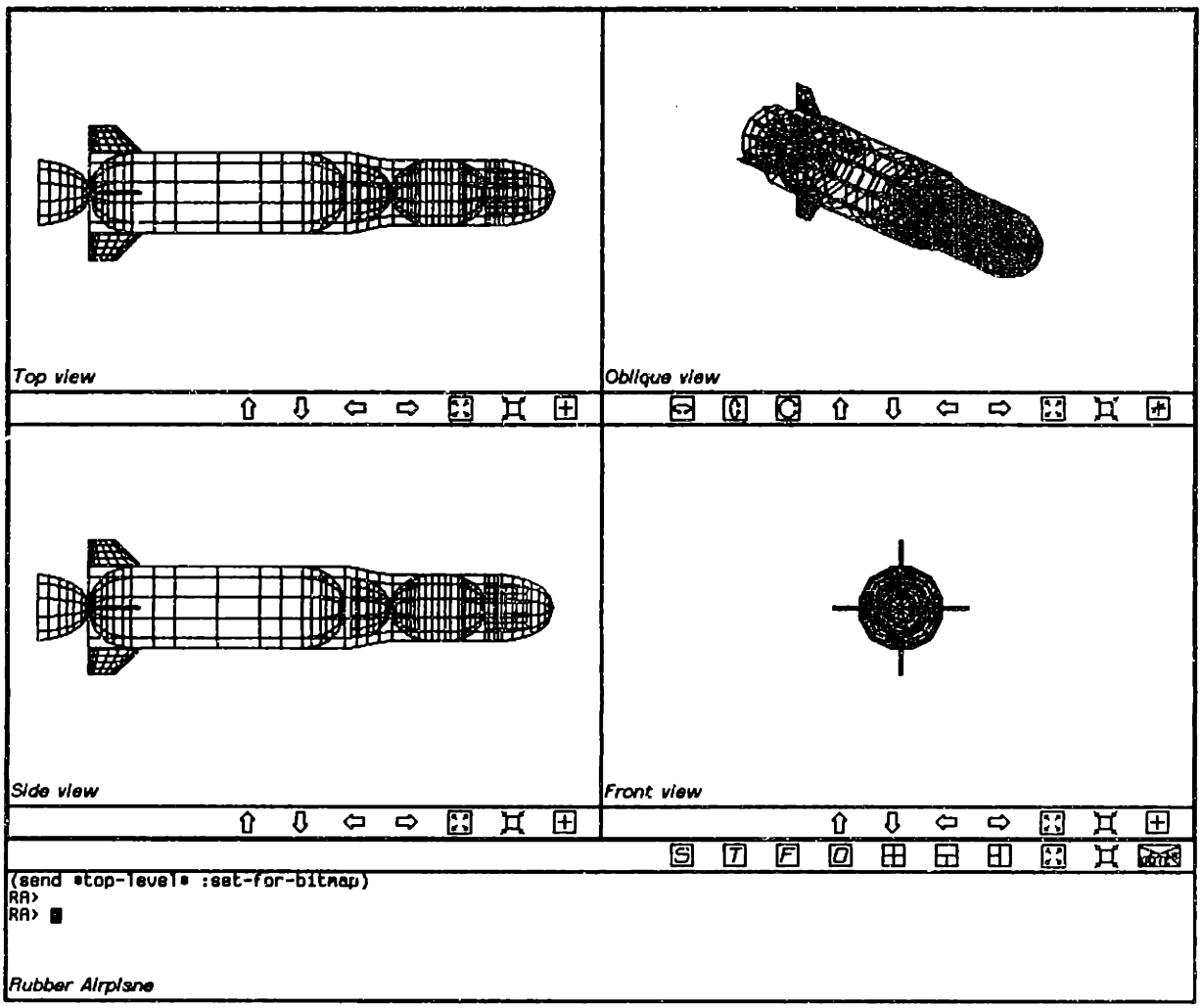

Figure 4.10: Screen image depicting the completed design for the small-payload launch vehicle test case.

4.4.2 Problem Representation

As might be expected, an entirely new set of component- and link-classes was required to implement the analysis for this design problem, which focused on component mass estimation and vehicle performance. A total of 10 components, 3 design states, and 14 links were used to model the spacecraft and its flight profile, with a total of 272 attributes and 194 constraints. A screen image depicting the geometry of the completed design is presented in Figure 4.10.

The component-classes used for this design are listed in Table 4.21. In this case, component-classes are used to represent the various physical components of the vehicle (payload, stage, nozzle, etc.); unlike the aircraft design problems, there was no need for the modeling of non-geometric subsystems (e.g., avionics, air conditioning). This may, however, be an artifact of the crudeness of this representation: were additional detail to be added,

Component Class	Description
rocket-stage	Geometry of the casing for a single rocket stage.
rocket-propellant	Geometry for a pill-shaped mass of rocket fuel.
nozzle	Geometry of a rocket nozzle.
connector	Geometry for the connector between two rocket stages.
rocket-fins	Airfoil geometry for a set of four stabilizing fins.
rocket-payload	Component representation of a cylindrically-shaped payload.
payload-shroud	Geometry of the protective shroud surrounding the payload.

Table 4.21: Instantiated component-classes for the small-payload launch vehicle test case.

Component Class	Superclasses	Attributes	Constraints	Instances
rocket-stage	2	17	10	2
rocket-propellant	3	24	13	2
nozzle	2	24	13	2
connector	2	26	14	1
rocket-fins	11	52	29	1
rocket-payload	2	16	10	1
payload-shroud	2	25	11	1
Total:		249	136	10

Table 4.22: Class statistics for the instantiated component-classes of the small-payload launch vehicle test case.

subsystem component-classes might again be needed. On the other hand, if the level of detail for the aircraft designs were improved, it might be possible to *remove* the subsystem component-classes from their representations.

Note also that all of the instantiated component-classes in this design are new; none have been carried over from the previous test cases. It was possible, however, to employ some of the same superclasses as were used in the other two designs. Specifically, the various airfoil superclasses for representing planform and cross-section were utilized in constructing the **rocket-fins** component-class. Additionally, the **point-mass-component** superclass was employed as a superclass for the **rocket-propellant** component-classes, since the coordinates chosen for its reference position happened to match those of its center of gravity. (To simplify the definition of the various attachment link-classes, the reference positions for most of the remaining component-classes were located at the center of their base cross-sections.) The class statistics for all of the component-classes instantiated for this test case are presented in Table 4.22. Those component-classes which appear most frequently as

Component Class	Subclasses	Description
design-component	9	Base component-class.
density-mixin	7	Relates component mass to density and volume.
basic-drag-mixin	4	Provides a general-purpose **drag** attribute.
basic-lift-mixin	3	Provides a general-purpose **lift** attribute.
basic-section	3	Provides general airfoil cross-section attributes.
basic-aerodynamics-mixin	2	Combines **basic-lift-mixin** and **basic-drag-mixin**.
NACA-section-pair	2	Provides attributes to represent NACA-airfoil cross sections at both the root and tip of an airfoil.
airfoil-planform	2	Provides airfoil planform parameters.
induced-drag-mixin	2	Breaks down component drag into induced drag and profile drag components.

Table 4.23: Most commo occurring component-classes among the instantiated component-classes of the small-payload launch vehicle test case.

Design State Name	Description
"Launch"	Launch of the vehicle.
"First Stage Jettison"	Exhaustion of first-stage propellant, and jettison of remaining first-stage mass.
"Second Stage Jettison"	Exhaustion of second-stage propellant, and jettison of remaining second-stage mass.

Table 4.24: Instances of class **design-state** for the small-payload launch vehicle test case.

Link Class	Description
stage-attachment	Attaches two rocket stages via a connector.
propellant-attachment	Attaches propellant to the interior of a rocket stage.
nozzle-attachment	Attaches a nozzle to the end of a rocket stage.
fin-attachment	Attaches a set of fins to the end of a rocket stage.
payload-attachment	Attaches a payload the forward end of a rocket stage.
payload-shroud-attachment	Attaches a payload shroud to the forward end of a rocket stage.
rocket-gross-properties	Provides net mass and weight attribute for the vehicle.
launch-mass	Equates the sum of all component masses to vehicle initial mass.
stage-mas	Sums the masses of all components associated with a single stage.
stage-jettison	Updates vehicle mass after stage propellant exhaustion.

Table 4.25: Instantiated link-classes for the small-payload launch vehicle test case.

superclasses of the instantiated component-classes are listed in Table 4.23.

Three instances of the **design-state** state-class were employed in representing the flight profile for the launch vehicle. A brief description of each is given in Table 4.24. Note that, for simplicity, the propellant-exhaustion and stage-jettison events for each of the two flight segments have been combined into a single design state.

Table 4.25 lists the link-classes which were instantiated for this design. These link-classes fall into two basic categories, those dealing with component positioning and attachment, and those concerned with performance analysis. In contrast to those associated with the previous two test cases (see Tables 4.5, 4.6, 4.13, and 4.14), link-classes for computing individual component weights are notably absent. As explained below, for this test case, components were responsible for computing their own weights. Statistics for the instantiated link-classes appear in Table 4.26. Note that no table of commonly-occurring link-classes

Link Class	Superclasses	Attributes	Constraints	Instances
stage-attachment	1	0	8	1
propellant-attachment	1	0	5	2
nozzle-attachment	1	0	3	2
fin-attachment	1	0	4	1
payload-attachment	1	0	3	1
payload-shroud-attachment	1	0	4	1
rocket-gross-properties	1	8	4	1
launch-mass	1	1	2	1
stage-mass	1	4	4	2
stage-jettison	1	3	4	2
Total:		23	58	14

Table 4.26: Class statistics for the instantiated link-classes of the small-payload launch vehicle test case.

has been provided; the only link-class which appeared as a shared superclass in this design was the base link-class, **design-link**.

4.4.3 Design Analysis

The implemented analyses for this test case fall into three categories: geometry, component and stage weights, and mission performance. The general characteristics of these analyses are presented below.

Component Geometry

As was the case for the aircraft design problems, it is also the case here that parameters describing component geometry (i.e., position, center of gravity, dimensions, surface area, and volume) are implemented as component-class attributes. Parameters and constraints representing relative geometric properties, such as the positions of two attached objects, are implemented via design links. Six of the ten link-classes defined for this test case implement the attachment of component groups.

Also, note that, as with the fuselage cross-section design implemented for the second test case, certain of the components included in this design were sized by visual inspection, rather than constraint computation. Specifically, the fit of the payload shroud around the payload, and of the interstage connector around the second stage nozzle, were accomplished by means of the geometry display portion of the *Rubber Airplane* user-interface (see Appendix C), rather than through specification of governing constraints.

```
(defcomponent density-mixin
              ((volume :documentation "Volume occupied by the component."
                       :low-value 1e-6 :order-of-magnitude 1
                       :high-value 10000 :dimensions "l3" :units "m3")
               (density :documentation "Mass density of the component."
                        :low-value 0.001 :order-of-magnitude 1
                        :high-value 1000
                        :dimensions "m l-3" :units "g cm-3"))
              ()
  :abstract-component
  (:required-attributes (mass "m"))
  (:category structures mixins)
  (:documentation "Relates component mass to density and volume."))

(defconstraint (density-mixin "Definition of Density") (density "kg m-3")
               ((mass "m") (volume "l3"))
  "Computes density as the ratio of mass to volume."
  (/ mass volume))
```

Figure 4.11: LISP definition for an abstract component-class which bases component mass on volume and density, **density-mixin**.

Component Weights

For the aircraft test cases, component weights were computed primarily via empirically-derived equations based on vehicle gross weight. Because component weights were based on net vehicle weight, it was necessary to implement these equations using design links.

For this design task, however, component weights were calculated based on vehicle dimensions and material properties. To simplify this task, a standard **density-mixin** component-class was defined, which provides attributes for component density and volume, as well as a constraint which relates the values of these two attributes to the component's mass. The definition of this component-class is presented in Figure 4.11. Note that classes which incorporate **density-mixin** as a superclass are responsible for supplying a constraint which computes a value for the **volume** attribute of **density-mixin**. As an example of such a constraint, consider the following constraint defined for the **rocket-stage** component class, based on its **length**, **radius**, and **thickness** parameters:

```
(defconstraint (rocket-stage "Casing Volume") (volume "m3")
      ((length "m") (radius "m") (thickness "m"))
  "Computes volume of the stage, using the thin-shell approximation."
  (* 2 pi radius length thickness))
```

(Recall that the **rocket-stage** component-class represents the casing for a rocket stage.)

Thus, the abstract **density-mixin** class provides only part of the mechanism needed to compute component weights. Inheriting subclasses are responsible for implementing the volume calculation.

In addition to component weights, however, it is also necessary to compute the net vehicle weight, as well as the weights of the various stages. Vehicle gross weight is computed by means of a collection constraint defined by the **rocket-gross-properties** link-class, in much the same way that the the **gross-properties** link-class was employed in the aircraft test cases. Stage weights, however, require an additional set of collector constraints, with collection predicates which take advantage of the **:owner-inferior-of** collection predicate. For this test case, the **stage-mass** link-class implements such a collector constraint, defined as follows:

```
(defsum (stage-mass "Net Stage Mass") mass-stage
        (:and (:owner-class design-component)
              (:owner-inferior-of stage)
              (:named "Mass"))
   "Collects all mass attributes associated with linkage STAGE."))
```

Note that "**mass-stage**" is the name of the attribute of link-class **stage-mass** whose value represents the net mass of all of the components associated with the stage; "**stage**" is the name for the linkage which represents the stage itself, whose linkage predicate requires that it be an instance of the **rocket-stage** component-class. This means that all the components to be associated with a given stage must be attached to the appropriate **rocket-stage** instance. Indirect attachment is also supported: a component which is attached to a second component is also considered to be an inferior of the second component's superior. Component instances are also considered to be inferior to themselves; thus the mass of the **rocket-stage** instance which acts as the design link's **stage** linkage will also be incorporated into the above summation constraint.

Finally, note also that the **stage-mass** link-class also defines a collector constraint which sums the propellant mass associated with a stage. For the solid rocket design employed here, this is just the mass of the single instance of the **rocket-propellant** component-class associated with the stage. If a liquid-fuel strategy were employed, however, each stage would probably require two instances of this component-class, since fuel and oxidizer would likely be stored separately.

Mission Performance

The mission performance analysis is based on the classical rocket equation, and has been implemented by means of the **rocket-gross-properties** and **stage-jettison** link-classes.

Note that the **mass** and **weight** attributes of the **rocket-gross-properties** have been defined as state-dependent attributes. The **stage-jettison** link-class uses the stage mass computed by the **stage-mass** link-class to compute the change in vehicle mass between two design states, due to the jettisoning of a stage. The **stage-jettison** link-class also implements the rocket equation to compute the change in vehicle velocity due to propellant burn. The net change in vehicle velocity from launch to jettison of the final stage determines the final orbital speed which may be obtained.

4.4.4 Observations

As indicated at the start of this section, this test case was undertaken to demonstrate the use of the component-modeling approach on a non-aircraft example. Perhaps the most challenging aspect of this particular design task was the implementation of staging; the ability of the program to incorporate this design feature gives some indication of the general applicability of the approach to engineering conceptual design.

In representing multiple staging, it was necessary—as discussed above—to compute stage weights. For the sake of generality, use of collector constraints for this purpose was desired. It was for this reason that the **:owner-inferior-of** collection predicate was added to *Rubber Airplane*, and use of this predicate keyword has enabled the specification of such collector constraints. Unfortunately, it is necessary that the user be aware of the presence of such constraints in a design, so that the necessary superior/inferior relationships among the component instances are established. Cavalier usage of link-classes which define such constraints is likely to yield erroneous results. A solution which avoids this complication would be preferred, but no viable alternative has yet been found.

4.5 Class Library

In implementing the three test cases, a fairly sizeable library of component- and link-classes was developed. This library was composed of the following elements:

- 47 component-class definitions,
- 64 link-class definitions,
- 265 attribute specifications,
- 296 constraint specifications, and
- 18 component-class geometry specifications.

Table 4.27 summarizes the contents of each of the three test-case designs, compiling the results presented in previous sections of this chapter. Note that, in a number of cases,

Test Case	Components	Links	States	Attributes	Constraints
Surveillance Aircraft	16	39	10	530	377
Commercial Transport	15	38	5	514	360
Cabin Interior	6	6	0	143	93
Launch Vehicle	10	14	3	272	194

Table 4.27: Summary of the contents of the test-case designs.

the total number of attributes or constraints employed by a given design exceeds the total number of attribute or constraint specifications provided by the class library. This result is due to

- inheritance of individual attribute and constraint specifications by multiple instantiable subclasses (e.g., inheritance of the same set of basic airfoil parameters by wing, horizontal stabilizer, and vertical stabilizer component-classes for the two aircraft designs), and
- multiple instantiation of individual component- and link-classes (e.g., multiple stage instances in the launch vehicle design).

and thus serves to demonstrate the ability of the component-modeling approach to simplify design task representation by obviating the repetitive specification of redundant design knowledge.

Chapter 5

Conclusions

5.1 Discussion of Results

5.1.1 Summary

In the preceding chapters of this dissertation, the description and specification of a general-purpose approach for performing computer-aided conceptual design has been presented. The basis of this approach is component-modeling, which organizes design knowledge according to the physical objects which comprise an engineered artifact. Component descriptions are based on attributes—the parameters describing the properties of a component—and constraints—the mathematical relationships which govern these parameters. Conceptual design, by its very nature, is unpredictable. Neither the solution, nor the appropriate solution path, may be known in advance. Furthermore, conceptual design tasks often seek to take advantage of the latest advances in the state of the art. For these reason, extreme flexibility is demanded: it is necessary that the designer be able to enter new component descriptions, based on current task needs. To simplify component definition, the use of object-oriented programming techniques, which support the definition of new component-types based upon the combination and modification of existent types, is specified.

It is very often the case in engineering design that the properties of one component are dependent upon those of one or more other components. Such interactions among groups of components are represented via design links, which may define constraints on the attributes of a selected group of components. For the purpose of facilitating the definition of general-purpose multi-component constraints, the association of collector constraints with design links is also specified. Rather than relating specific attributes from a pre-specified set of components, as is the case with normal design-link constraints, the parameters of a collector constraint are chosen based on satisfaction of an appropriate collection predicate; i.e., the attributes to be related by a collector constraint are identified via pattern-matching.

Finally, because component properties may be time-dependent, some means for imple-

menting variation in attribute values is needed. The use of design states has been introduced, along with the ability to declare any attribute or constraint as being state-dependent. Any number of design states may be included in a design; for each design state, a corresponding independent instance of each state-dependent attribute or constraint is created. Design states thus allow the incorporation of step-wise time dependencies into the representation of a design problem.

To add flexibility to the actual mathematical analyses which model a design, the use of constraint propagation is adopted. Constraint propagation allows a single declarative statement of a mathematical constraint to be used to infer multiple imperative relationships among the constraint's parameters. Thus, an inverted form of the constraint may be applied if the current status of its parameters makes such use desirable. The ability of constraint propagation to automatically modify mathematical relationships in order to accommodate an evolving problem description is of great benefit in conceptual design where, as mentioned above, the appropriate solution path is not necessarily known in advance.

In order to demonstrate the practicality and utility of this approach, a testbed implementation has been developed. The resulting program, *Rubber Airplane*, supports the object-oriented description of components, links, and states, as class definitions. Attributes and constraints may be associated with these classes, and link-classes may define collector constraints. Inheritance of classes, and specialization of inherited attributes and constraints, is also supported. *Rubber Airplane* also allows for the specification of component geometry, based on attribute-values, and the program interface provides for the graphical display of component geometries.

In using the program, it is first necessary to define any required component-, link-, or state-classes. To actually model a design task, various instances of these classes are created; corresponding instances of the attribute and constraint specifications associated with a class are also created. The instantiated attributes may then be manipulated as necessary to solve the design problem: constraint propagation is used to ensure satisfaction of the relationships which govern attribute values, as specified by the constraints . Note as well that multiple instances of a single class be may constructed, as needed.

To actually investigate the utility of the approach, application of the prototype design tool to a number of test-case design problems was required. Three tasks from the domain of aerospace vehicle design were chosen and implemented using *Rubber Airplane*. The appropriate component- and link-classes were defined, and instances of these classes were used to model the design problems. Design analysis was focused on mission performance, and, in the case of the aircraft designs, vehicle aerodynamics. The types of analyses implemented ranged from handbook formulas for estimating component weights to a complete implementation of the vortex-lattice method for computing lifting-surface aerodynamic forces.

In this way, the ability of the constraint-based component-modeling approach to accommodate varying levels of analytical sophistication has been shown.

Perhaps the key feature of the approach advocated here is its ability to treat arbitrary design knowledge—in the form of the mathematical relationships which govern a design task—as data to be manipulated by the computer, rather than as code to be executed. The use of constraint propagation frees the user from concern over the computational path used to solve a design problem. Object-oriented programming techniques provide a ready means for organizing design knowledge, and for incrementally developing appropriate design representations; the use of inheritance allows for the specification of a modular database of component- and link-based design knowledge. The system described here, then, is perhaps better described as an environment which supports and simplifies the design process, rather than as a computer program which "performs" engineering conceptual design.

5.1.2 Representational Boundaries

Perhaps the most important issue to be considered in evaluating this approach is its ability to represent the design knowledge required to model design tasks in engineering conceptual design. The test cases which have been implemented using *Rubber Airplane* have been taken from the realm of aerospace engineering, where parametric design, driven by the mathematics which model the underlying physics, plays a key role. In contrast, other engineering design tasks, such as architectural layout, electronic circuit board design, and certain problems in mechanical design, are more dependent on geometry. While the inclusion of geometry display features enhances the utility of the *Rubber Airplane* program (see Chapter 4), because the geometry display is parameter-driven, it is not as convenient for this second sort of design task as, say, the graphically-oriented interface of conventional CAD systems. This is not to say that purely geometric design using the current approach is necessarily unwieldy; the cross-section design developed for the second aircraft test case demonstrates the use of *Rubber Airplane* for just such a task. The approach is, however, better suited to parametric design in which geometry plays only a secondary role.

In the domain of parametric design, then, constraint-based component-modeling has been shown to be a viable, general-purpose means for representing and manipulating design information. Because *Rubber Airplane* constraints are written using standard LISP functions, a wide variety of analytical methods may be implemented. There are no technical obstacles to calling arbitrarily complex programs (e.g., the vortex-lattice program described in Chapter 4) from *Rubber Airplane*; the invocation of programs on remote host computers is also feasible. While there are no inherent limitations on depth of analysis, though, consideration of program performance may impact the choice of physical models to employ in completing a conceptual design task. First of all, the use of computationally-intensive

methods will degrade program responsiveness. Of perhaps greater significance, though, is the general principle that, as analysis becomes more detailed, inversion of the computational method becomes more difficult, or even impossible. The use of non-invertible constraints reduces flexibility, and may be over-restrictive in the context of conceptual design.

Finally, note that it is probably not coincidental that the component-modeling approach closely resembles the manner in which engineering knowledge, particularly in aerospace engineering, is taught. While this knowledge is organized primarily around analytical fields (e.g., structures, aerodynamics, propulsion), within the various disciplines subject matter is often presented based on structural—i.e., *component*—types. Examples of domain-dependent, component-specific subject matter include:

- the drag profile of a blunt body, such as an aircraft fuselage or landing gear;
- deflection and bending moment due to beam loading, which may be used to predict airfoil structural requirements; and
- the thrust characteristics of various gas-turbine engines.

Computer programs based on component-modeling might therefore serve well as pedagogical tools. Furthermore, recall that design-link definitions require declaration of the component-types to which they may be applied, through their linkage specifications. Such specifications therefore provide explicit documentation of the interrelationships within a group of components, information which has the potential for significant educational value.

5.1.3 Shortcomings of the Current Implementation

Of course, there are some deficiencies in the *Rubber Airplane* implementation of constraint-based component-modeling. Perhaps the least robust aspect of the current program is its use of exclusively numerical techniques for implementing constraint propagation. These methods work quite well for inverting individual constraints, but the use of numerical methods for solving computational loops is less than satisfactory, due their strong dependence on initial conditions.

As mentioned in Chapter 3, the simultaneous Newton-Raphson and iteration-loop methods are best suited to solving computational loops based on incremental changes in the values of the loop parameters. Once the initial sizing of a design has been accomplished, this limitation on these methods is acceptable: minor variations in parameter values do not adversely impact the convergence of these numerical methods. During the beginning stages of the design process, however, when many new component- and link-instances are being added, and parameter values are subject to large fluctuations, these methods are prone to instability. For this reason, at this stage in the design, it may be necessary to modify the

computational path, in order to avoid formation of computational loops, until values close to the desired results are obtained. The original path may then be restored, causing the formation of a computational loop at a time when the current values of its parameter are more conducive to numerical convergence.

For example, in the subsonic transport test case presented in Chapter 4, it was necessary during the early stages to enter various fixed values for the aircraft's gross weight, based upon which the number of passengers is computed, without the need for computational loops. Once the result was near the desired number of passengers, the fixed value for gross weight could be removed; fixing the number of passengers to the desired value would then result in the formation of a computational loop for computing the gross weight. By delaying loop construction, the likelihood of convergence was improved—at the expense, though, of effectively performing the initial iterations through the loop by hand.

A second problem resulting from the use of numerical methods concerns the possible presence of redundant constraints. While constructing a new design model, for example, it is quite possible that multiple instances of a given constraint might inadvertently be created. Similarly, it is possible that the definitions of a pair of link-classes are incompatible, such that they both define the same relationship among a group of parameters. If both of these classes are instantiated by the same design, redundant constraints will again be present. Unfortunately, because all constraints are treated as black boxes, there is no means for detecting such circumstances, since the internal calculations performed by the constraints cannot be examined. The difficulty arises from the possibility that the program will attempt to perform loop propagation on a set of redundant constraints: given, for example, two instances of the constraint $y = f(x)$, it is a simple matter for the loop detection algorithm to decide that a computational loop may be constructed from the pair of constraints. The corresponding set of simultaneous equations is singular, however, and the loop-solving algorithm will fail.

One simple means for avoiding this particular case is to check the set of parameters associated with each newly instantiated constraint. If it matches that of an existing constraint, then it is suspect, and the user may be called upon to determine the validity of the new constraint. It may also be the case, however, that a set of constraints is redundant only in combination. Consider, for instance, the following system of equations:

$$\begin{aligned} a &= b + c \\ d &= a + e \\ d &= b + c + e \end{aligned}$$

This system of equations is also singular. No unique pair of equations in this set is redundant; the system is only singular when the three equations are combined. If these equations

represent, say, constraints contributed by three different design links, then detection of the redundancy cannot take place until actual loop propagation is attempted, and subsequently fails.

Note fr[illegible]he description of the loop-detection algorithm in Chapter 3 that no means is provided for storing the contents of computational loops, such that they might be reused. Recall that, when the value of an attribute whose value-supplier is `:user` is changed, the attribute's value-supplier is first set to `:guess`, and then back to `:user`, so that the appropriate retractions occur. If the attribute is a member of a loop, the contents of t[illegible] loop (i.e., its attributes and constraints) must be re-computed. While the loop detection algorithm performs well, it is nevertheless inefficient to repeat the search in such cases.

Another deficiency of the current implementation is the lack of an interface for tracing through the constraint propagation network. While the current interface includes the display of attribute value-suppliers, it provides no means for displaying, for example, the sequence of constraint applications which relate a pair of attributes. Neither are there any provisions for displaying computational loops. Determining dependencies among *Rubber Airplane* attributes is therefore a much more tedious task than it need be.

Performance of the program—in terms of calculation speed and general responsiveness—was acceptable. For the test cases examined here, there is no obvious delay between setting the value-supplier of an attribute to `:user` and seeing the resulting calculations displayed on the screen, except when fixing the attribute enables propagation of one or more vortex-lattice constraints. (Recall that the vortex-lattice constraint, because it computes aerodynamic forces which are state-dependent, is itself state-dependent. There will thus be multiple state-instances of this constraint.) In such cases, though, some delay is to be expected. The only other performance problem associated with the current program involves geometry delays. Because no special graphics hardware is present in the Lisp Machine used for this research, update of the geometry display portion of the user interface is, at present, unacceptably slow.

A potential problem with the approach described here involves the modeling of time-dependent phenomena for artifacts which have mixed modes of operation. Currently, state-instances of state-dependent attributes and constraints are constructed for every design state present in a design. Consider, though, the design of a system which employs two or more different kinds of design states. For example, an aerospace plane would require state-dependent flight conditions to model its operation in the atmosphere, as well as state-dependent orbital characteristics to model the space segments of its mission profile. Creating state-instances of the flight condition attributes and constraints for the extra-atmospheric design states is wasteful and unnecessary, as would be the construction of flight-segment state-instances for the orbital parameters. A possible solution to this problem would be the

addition of some sort of predicates for determining when state-instantiation is appropriate. Alternatively, it might be more straightforward to divide such design tasks between two or more *Rubber Airplane* designs, as was done for the internal and external vehicle designs of the second aircraft test case (see Chapter 4). If this latter approach were adopted, however, some means for directly linking the common parameters of the two designs would be desirable, in order to automate the transfer of results between the two individual designs.

A final difficulty involves the definition of design-entity classes. Currently, class definitions are entered via a text editor, and entered into the program by loading the appropriate LISP code. This approach requires that the user ensure the compatibility of inherited classes, and know how to find the names of attributes and classes which are to be referenced in linkage or collection specifications. Thus, while the component-modeling approach has been motivated by a desire to help manage the complexity of conceptual design tasks, the current implementation ultimately serves to replace the bookkeeping associated with the design itself by a need for the user to be familiar with the details of the class library; one form of complexity management has been replaced by another. Two possible means for addressing this issue are:

- *Providing a more "intelligent" interface for defining component- and link-classes.* If the program were to support searches of the class library, and automatically inform users of the consequences of incorporating one class as a superclass of another class (i.e., through inheritance), much of the overhead involved in defining new classes could be eliminated.

- *Adding the ability to define collections of components and links which may be instantiated as a whole.* Such a collection might be referred to as a "package" or "assembly" and would include specification of how the included entities should be associated with the linkages of the assembly's design links. For example, an `aircraft-aerodynamics` assembly might consist of appropriate wing, horizontal stabilizer, and vertical stabilizer component-instances, as well as an instance of the vortex-lattice link-class, for computing their aerodynamics. A related `canard-aerodynamics` assembly might replace the horizontal stabilizer component-instance with an appropriate forward-mounted airfoil.

In the first case, class definition is simplified, since the program itself aids in determining the dependency relationships among classes. In the second case, class instantiation is simplified, since multiple components, along with the appropriate link instances, could be created simultaneously. Additionally, the compatibility of the design entities of an assembly could be verified by the program in advance, further reducing the workload of the program user. There is the possibility of overlap among assemblies, though: two assemblies could

include specification of the same component. For example, an `aircraft-stability` assembly might also require wing and tail surface components, and would therefore conflict with the aforementioned `aircraft-aerodynamics` assembly. In such cases, some means for supporting the incremental instantiation of assemblies, which avoids duplicate instantiations, would be desired.

5.2 Comparison to Related Work

The described system shares many features in common with the various design tools discussed in Chapter 1. Like GRADE [8], *Rubber Airplane* supports component-based geometry display. Like *Paper Airplane* [11] and MARKSYMA [30], *Rubber Airplane* employs constraint propagation to add flexibility to parametric design. The results of the parametric design are then linked to the geometry display, as is the case with MATHPAK [31].

In addition, however, the component-modeling approach implemented by *Rubber Airplane* calls for the use of object-oriented programming techniques in organizing the design knowledge required for this parametric design. In this respect, *Rubber Airplane* resembles the Concept Modeler [32], the successor to MATHPAK, which provides the user with a limited set of object-types from which to construct a design representation. Various constraints are associated with these object-types; once an object-type is instantiated, its constraints are subject to constraint propagation, as in *Rubber Airplane*. Specific constraints associated with the attachment of pairs of objects are also provided in the Concept Modeler, though the notion of a general "design link" data structure is not present. Furthermore, while the Concept Modeler supports the interactive addition of new constraints to a design representation, it does not allow the user to define new object-types. Its use is therefore restricted to those design problems which utilize only the component-types which are provided by the program.

Two object-oriented design systems which do support the definition of new component-types are the commercial systems discussed in Chapter 1, ICAD Inc.'s ICAD program [29] and Wisdom System's Concept Modeller [24]. Again, though, unlike *Rubber Airplane*, no explicit data structures for managing component interactions are provided. Instead, component interactions are modeled based on specification of a part/sub-part hierarchy among components, similar to the superior/inferior component designations supported by *Rubber Airplane*. Objects higher up in the part/sub-part tree may access any of the parameters associated with their sub-parts, and it is in this manner that component dependencies are implemented. Thus, the sub-parts of a given component must be specified simultaneously with the component definition. *Rubber Airplane*'s use of pattern-directed linkage specification allows component interactions to be described in a more general fashion, yielding

greater code modularity.

As indicated in Chapter 1, ICAD and the Concept Modeller utilize rule-based backward-chaining to perform demand-driven computations, thereby gaining computational efficiency. Constraint propagation, as employed by *Rubber Airplane*, uses forward-chaining, such that constraint calculations are performed whenever possible, rather than whenever necessary. In order to succeed, however, backward-chaining effectively requires the specification in advance of a complete computational path for each of the parameters; efficiency is thus gained at the expense of flexibility.

Finally, it should be pointed out that none of the above systems provides any special features for handling time-dependent phenomena. Indeed, of all the systems discussed in Chapter 1, only the THINGLAB program [5] includes any provisions for time-varying parameters. Recall that THINGLAB combines constraint propagation with an object-oriented programming environment for developing computer-based simulations. As reported in Reference [23], recent extensions to THINGLAB have included the addition of a special variable representing time, which cannot itself be constrained, but may be referenced by the constraints on other parameters. This is, of course, a completely different approach from that taken here, which supports only step-wise time-dependency of attributes and constraints.

5.3 Suggestions for Further Research

5.3.1 Program Enhancements

As indicated in Section 5.1.3, the prototype implementation of the concepts presented here, *Rubber Airplane*, suffers from a number of performance deficiencies. In the sections below, means for addressing some of these issues are presented. Additional features which would enhance the use of the program as a design tool are also discussed.

Symbolic Algebra

In Section 5.1.3, certain difficulties deriving from the current program's use of numerical techniques for solving computational loops are discussed. For this reason, supplementing the current approach with analytical techniques for constraint inversion and the transformation of systems of equations is suggested. Note that symbolic techniques are the *only* means by which constraint redundancies, as discussed in Section 5.1.3, may be detected in advance. As indicated in Chapter 1, though, such techniques can be computationally expensive, and are not always applicable to the general types of constraints currently supported by *Rubber Airplane*. Given adequate initial conditions, a well-posed problem should always succumb to numerical solution; symbolic methods are not as rigorous. For this reason, research into

hybrid systems, which utilize numerical methods in cases where analytical techniques are inefficient or inapplicable, might yield the best results.

Linked Designs

As discussed in Section 5.1.3, the ability to set up links between two different *Rubber Airplane* designs, such that attribute values may be transferred from one design to another, would be of use in cases where design tasks may be decomposed into multiple, separate designs. Such a feature could have been exploited in the second test-case design, which employed an auxiliary design for determining the internal arrangement of the fuselage cabin (see Chapter 4). The ability to establish inter-design connections would also enable the representation of vehicles such as the aerospace plane, which employ multiple modes of operation over their mission profiles. Design states associated with each particular mode of operation (e.g., for the aerospace plane, atmospheric operation and orbital operation) could be assigned to a corresponding *Rubber Airplane* design, and dependency links could be used to propagate the results of the various individual designs. In effect, multiple designs would be able to share certain component and link instances—or at least some of their attributes—while other design entities would be unique to a particular linked design.

Layered Constraints

As the solution to a design is developed, it is often desirable to improve the level of accuracy with which parameter values are calculated, by replacing some of the analysis routines with more rigorous versions of the same computations. In the current implementation, this process can be performed by replacing a class-instance that implements one level of analysis with an instance of a different class, which implements a more detailed form of the same analysis. If some sort of annotation of such related classes were supported, though, this process could be automated: at the user's request, a class-instance could be replaced by an instance of the class which implements a more rigorous model of the same phenomenon. If such a feature were available, it would not be necessary for the user to search the class library for the appropriate replacement class and, in the case of link-classes, reestablish the appropriate linkages.

Along similar lines, some means of supplying default values for attributes might be advantageous. Currently, it is possible to specify an initial value for an attribute by means of the `:value` argument to the `defattribute` macro, but it is always the case that when an attribute is instantiated, its value-supplier is set to `:guess`. Recall that the values of attributes whose value-suppliers are `:guess` are not propagated; recall also that only attributes whose value-suppliers are `:guess` may be computed via constraint propagation. It might perhaps be useful to add another possible value-supplier, `:default`, such that the val-

ues of attributes whose value-suppliers are `:default` may be propagated, but these values may also be overridden by constraint calculations. Retraction of default-based computations may be problematic, but it can simplify the initial construction phases of a new design model. Consider, for example, a hypothetical **gear-deployed?** discrete attribute, associated with a **landing-gear** component-class, intended for use in aircraft design problems. Such an attribute would likely be designated as state-dependent, implying that multiple state-instances of the attribute should be created for the designs which incorporate this component-class. Under most flight conditions, the value of this **gear-deployed?** attribute should be "false"; only during takeoff and landing must the gear be deployed. This attribute is thus a prime candidate for specification of a default value, which in this case would be "false".

Improved Geometry Representation

Perhaps the most complicated aspect of defining component geometries (see Appendix B) in the current implementation is the computation of relative object positions when multiple geometric objects are combined to represent the geometry of a single component. As mentioned in Chapter 1, the two commercial systems discussed there (ICAD Inc.'s ICAD program and Wisdom System's Concept Modeller) support the symbolic description of relative object positioning. For example, one object part may be defined as being "above", "below", or "in front of" another part, without the need to explicitly specify the mathematical relationship between the positions of the two objects. The addition of such capabilities to *Rubber Airplane* would greatly simplify the task of component geometry specification.

To date, the types of geometric objects available in *Rubber Airplane* for describing component geometries have proven adequate. These geometry-types are by no means exhaustive, however. Some interesting research by Pentland [28] has suggested the use of so-called "superquadrics" as a general-purpose object-type for modeling geometry: superquadrics are solid bodies—the three-dimensional analog to two-dimensional superellipses [13]—which may be arbitrarily deformed, twisted, and bent to depict a wide variety of shapes, including most of those which have already been implemented for *Rubber Airplane*. The representational flexibility of superquadrics with respect to geometry seems particularly well-suited to the representational flexibility of the component-modeling approach advocated here; investigation of the possible combination of these two approaches seems likely to yield useful results.

Finally, recall that component geometry is based on attribute values. As attribute values are modified, the geometry is updated to reflect the new values. As indicated in Chapter 4, certain aspects of many design tasks are most easily accomplished by visual examination of design geometry, in order to determine the "fit" of two or more components.

In performing such tasks, trial-and-error manipulation of attribute values is required in order to size the relevant components. One approach for simplifying these tasks would be to make the association between attribute values and geometry display bi-directional rather than uni-directional, such that modification of the geometry would cause corresponding changes in attribute values, as well as the reverse. The resulting system would therefore need to incorporate conventional CAD techniques for manipulating component geometry. This is no easy task, but the resulting system could prove to be extremely powerful, insofar as it would be required to integrate both parametric and geometric design.

Class Definition and Instantiation

Currently, *Rubber Airplane* requires that the user keep track of the dependencies among the various design-classes himself, and that he be able to choose classes for instantiation with little assistance from the program. The use of an improved interface for defining design-entity classes, and for browsing through the class library, was recommended in Section 5.1.3. Such an interface could automatically display inheritance dependencies, as well as perform searches through the class library based on the names and documentation associated with classes and their attributes and constraints, in order to find those classes which implement some desired feature.

In order to simplify the incorporation of design links and the components they relate, the addition of pre-defined "packages" or "assemblies" of *Rubber Airplane* components and links has been suggested. The difficulty associated with design links lies in recognizing when all of the required components are present. Furthermore, for the naive user, lack of knowledge of the available link-classes can lead to the absence of required analyses from a design model. An additional means for simplifying link instantiation, then, is to monitor all class instantiations, and compare the set of instantiated classes with the linkage specifications of all defined link-classes. When the current contents of a design are sufficient to instantiate a given link-class, the user may then be given the opportunity to add the corresponding instance, based on this suggestion from the program. Such computer-generated assistance could be particularly beneficial to novice users; one potential disadvantage to such an approach, though, is the overhead associated with monitoring the linkage specifications of all the defined link-classes.

5.3.2 *Rubber Airplane* as a Platform for Other Systems

The approach to computed-aided conceptual design outlined in this document also lends itself to the development of layered systems for engineering problem-solving. For instance, a top-level program might act as an interface between the user and a program such as *Rubber Airplane*, guiding the user through a design task, automatically incorporating component-

and link-classes into a design based on responses from the user. In cases where the top-level program cannot accommodate a user's requests, though, he may directly access the components and links which comprise the design, taking advantage of the flexibility of the underlying design representation.

One example of such an application might be an expert system for developing baseline designs from high-level design specifications entered by the user. Once the baseline design has been created, the user is free to adjust its parameters as necessary, relying on constraint propagation to ensure satisfaction of the governing equations. This baseline design might be further modified through the addition of new representational elements, based on design-entity classes. In this way, constraint-based component-modeling can be employed to accommodate the performance limitations of the expert system.

An alternative approach would be to improve the performance of a general-purpose system such as *Rubber Airplane* through the use of embedded systems. Serrano, in his doctoral dissertation on the Concept Modeler [32], advocates the use of a "Conceptualizer" module which utilizes domain-specific knowledge for a particular class of design problems to guide both problem representation (i.e., class instantiation) and constraint propagation. To some extent, *Rubber Airplane* already incorporates some domain-based knowledge, through its use of the "computational loop" heuristic for detecting systems of equations. Serrano suggests even greater use of domain-dependent knowledge—including, for instance, parameter-specific heuristics—to improve program performance and provide computer-based decision support during design development and evaluation.

5.3.3 Exploratory Design

In this dissertation, component-modeling and constraint propagation have been proposed as means for implementing flexible computer aids for engineering conceptual design. In investigating this premise, a prototype design tool has been developed, which has then been used to implement a number of test case designs. Note, though, that the issue of concern in this exercise was the ability of these techniques to support representative analyses from the chosen problem domain of aerospace engineering.

The objective here, then, has been knowledge representation, and the prototype system has demonstrated the successful application of this approach to modeling tasks in aerospace conceptual design. An important aspect to consider, though, is the application of these results to the conceptual design process. Conceptual design is an exploratory process; knowledge representation is not. While the goal of this research has been the description of a system for supporting exploratory design, the primary concern of the present work has been concept definition and demonstration. Explicit testing of the prototype implementation purely as a tool for performing conceptual design has not been performed. For this reason,

the use of *Rubber Airplane* and related systems for additional exercises in conceptual design is recommended. Only through repeated testing of novel approaches such as those advocated here can their applicability to the unique demands of engineering conceptual design be fully demonstrated.

Appendix A

Dimensions and Units

A.1 Syntax

In *Rubber Airplane* expressions representing dimensions and units are input and displayed as LISP strings. Non-dimensionality is denoted by the empty string, "". The syntax for dimension- and unit-strings involving but a single dimension or unit is:

"<*symbol*><*whitespace**><*exponent*>"

where "<*symbol*>" is a symbol representing the appropriate dimension or unit (see Tables A.1– A.6), "<*whitespace**>" indicates the optional presence of one or more whitespace characters (spaces or tabs), and "<*exponent*>" is a non-zero integer denoting the exponent for the dimension or unit designated by "<*symbol*>". Note that indication of sign is required only for negative exponents (though it may be specified for positive exponents, as well), and that if the exponent is unity, it may be omitted altogether. Thus, valid single-dimension dimensions expressions include "`l +1`" (corresponding to dimensions of length), "`l +3`" (corresponding to length cubed, or volume), and "`t -1`" (corresponding to inverse time, or frequency). Note that, by omitting all optional elements, these example dimension-strings may be re-written more succinctly as "`l`", "`l3`", and "`t-1`", respectively. Similarly, the example unit-strings "`ft`", "`m3`", and "`s-1`", refer to units of feet, cubic meters, and inverse seconds, respectively.

If we define the syntactical elements of single-unit and single-dimension expressions as the composite syntactical element, <*single-expr*>, i.e.,

<*single-expr*> = <*symbol*><*whitespace**><*exponent*>

then the syntax for defining composite expressions, involving two or more dimensions or units, may be concisely described as:

"<*single-expr*><*whitespace*><*single-expr*>...<*whitespace*><*single-expr*>"

where "<*whitespace*>" indicates the required presence of one or more whitespace characters. Thus, a composite dimension- or unit-expression is simply a LISP string containing a sequence of single dimension- or unit-expressions, delimited by (required) whitespace characters. Valid composite dimension expressions include "`m +1 l +1 t -2`" (corresponding to dimensions of force), "`m +1 l +1 t -2 l -2`" (corresponding to force per unit area, or pressure), and "`f +1 l -2`" (which uses the derived dimension for force, "`f`" (see Section A.2), to again represent force per unit area). Omitting optional whitespace and exponents, these expressions may be abbreviated as "`m l t-2`", "`m l t-2 l-2`", and "`f l-2`", respectively. Examples of valid composite unit expressions are "`kg m s-2`" and "`lb ft-2`", corresponding to kilogram-meters per second-squared and pounds per square foot, respectively.

Finally, note that multiple occurrences of the same unit or dimensions within a composite expression, such as the two appearances of the symbol for length, "`l`", in the second composite-dimensions example, "`m l t-2 l-2`", is permitted. When computing conversion factors, or comparing the dimensionality of two unit- or dimension-strings, all expressions are converted into a canonical internal form, which explicitly combines such multiple occurrences.

A.2 Defined Dimensions

Dimensions currently defined for use in the *Rubber Airplane* program appear in Table A.1. Note that, for the convenience of the user, a number of so-called **derived dimensions** are provided, such as force ("`f`"), power ("`P`"), and voltage ("`V`"). Such dimensions may be completely expressed in terms of the irreducible **canonical dimensions**, such as mass ("`m`"), length ("`l`"), time ("`t`"), and electrical charge ("`q`"). Derivations for all derived dimensions are included in Table A.1.

A.3 Defined Units

A listing of the units available with *Rubber Airplane* is presented in Tables A.2– A.6. For the sake of brevity, the various conversion relationships among the units are not included. However, those units which are employed as **internal units** within the program are marked by a dagger (†). *Rubber Airplane* data structures store the actual values of scalar attributes in these internal units, converting to other units as required by constraints or when requested for attribute display purposes.

Dimension	Name	Derivation
"a"	amount of substance	
"A"	angle	
"C"	capacitance	$q^2t^2m^{-1}l^{-2}$
"c"	currency	
"i"	current	qt^{-1}
"q"	electric charge	
"E"	energy	$t^{-2}ml^2$
"f"	force	$t^{-2}ml$
"L"	inductance	$q^{-2}ml^2$
"l"	length	
"I"	luminous intensity	
"M"	magnetic flux	$q^{-1}t^{-1}ml^2$
"B"	magnetic inductance	$q^{-1}t^{-1}m$
"m"	mass	
"P"	power	$t^{-3}ml^2$
"p"	pressure	$t^{-2}ml^{-1}$
"R"	resistance	$q^{-2}t^{-1}ml^2$
"T"	temperature	
"t"	time	
"V"	voltage	$q^{-1}t^{-2}ml^2$

Table A.1: Listing of defined dimensions, showing derivation of composite dimensions.

Type	Unit	Name
Time	"cen"	centuries
	"dy"	days
	"dec"	decades
	"fn"	fortnights
	"hr"	hours
	"us"	microseconds
	"mil"	millennia
	"ms"	milliseconds
	"min"	minutes
	"ns"	nanoseconds
	"ps"	picoseconds
	"s"	seconds†
	"wk"	weeks
	"yr"	years
Length	"Ang"	angstroms
	"AU"	astronomical units
	"cm"	centimeters
	"Re"	earth radii
	"fath"	fathoms
	"ft"	feet
	"fm"	femtometers
	"f"	fermis
	"fr"	furlongs
	"in"	inches
	"km"	kilometers
	"ly"	light years
	"Mm"	megameters
	"Mpar"	megaparsecs
	"m"	meters†
	"um"	micrometers
	"mi"	miles
	"mm"	millimeters
	"mil"	mils
	"nm"	nanometers
	"NM"	nautical miles
	"par"	parsecs
	"pm"	picometers
	"SM"	statute miles
	"yd"	yards

Table A.2: Listing of defined units for time and length. Dagger (†) denotes *Rubber Airplane* internal units.

Type	Unit	Name
Area	"acre"	acres
Volume	"cc"	cubic-centimeters
	"gal"	gallons
	"l"	liters
	"ul"	microliters
	"ml"	milliliters
	"pt"	pints
	"qt"	quarts
Velocity	"fps"	feet-per-second
	"kt"	knots
	"c"	light-speeds
	"mph"	miles-per-hour
Acceleration	"g's"	g's
Mass	"AMU"	atomic mass units
	"Me"	earth-masses
	"me"	electron-masses
	"g"	grams
	"kg"	kilograms†
	"MT"	metric tons
	"ug"	micrograms
	"mg"	milligrams
	"ng"	nanograms
	"pg"	picograms
	"lbm"	pounds-mass
	"sl"	slugs
	"Ms"	solar-masses
Force	"dyne"	dynes
	"kN"	kilonewtons
	"N"	newtons
	"lbf"	pounds-force
Weight	"kton"	kilotons
	"Mton"	megatons
	"oz"	ounces
	"lb"	pounds
	"ton"	U.S. tons

Table A.3: Listing of defined units for area, volume, motion, mass, and force. Dagger (†) denotes *Rubber Airplane* internal units.

Type	Unit	Name
Pressure	"atm"	atmospheres
	"bar"	bars
	"kbar"	kilobars
	"kPa"	kilopascals
	"ksi"	kilopounds-per-square-inch
	"Mbar"	megabars
	"MPa"	megapascals
	"ubar"	microbars
	"mbar"	millibars
	"Pa"	pascals
	"psf"	pounds-per-square-foot
	"psi"	pounds-per-square-inch
Energy	"Btu"	British thermal units
	"cal"	calories
	"eV"	electron volts
	"erg"	ergs
	"J"	joules
	"keV"	kilo-electron volts
	"kcal"	kilocalories
	"kJ"	kilojoules
	"kTNT"	kilotons of TNT
	"kwh"	kilowatt-hours
	"MeV"	mega-electron volts
	"MTNT"	megatons of TNT
	"mJ"	millijoules
	"TNT"	tons of TNT
Power	"GW"	gigawatts
	"hp"	horsepower
	"kW"	kilowatts
	"MW"	megawatts
	"mW"	millawatts
	"W"	watts

Table A.4: Listing of defined units for pressure, energy, and power. Dagger (†) denotes *Rubber Airplane* internal units.

Type	Unit	Name
Angular Measurement	"deg"	degrees
	"rad"	radians†
	"rev"	revolutions
Angular Velocity	"rpm"	revolutions-per-minute
Frequency	"GHz"	gigahertz
	"Hz"	hertz
	"kHz"	kilohertz
	"MHz"	megahertz
Temperature	"deg-C"	degrees-Centigrade
	"deg-F"	degrees-Fahrenheit
	"K"	degrees-Kelvin†
	"R"	degrees-Rankine
Viscosity	"cp"	centipoise
	"up"	micropoise
	"mp"	millipoise
	"poise"	poise
Currency	"ct"	cents
	"$"	dollars†
Amount of Substance	"gm-mol"	gram-moles
	"kg-mol"	kilogram-moles
	"mol"	moles†
	"lb-mol"	pound-moles
Luminous Intensity	"cd"	candelas†

Table A.5: Listing of defined units for angles, frequency, temperature, etc. Dagger (†) denotes *Rubber Airplane* internal units.

Type	Unit	Name
Electrical Charge	"C"	coulombs†
	"e"	electron-charges
	"mC"	millicoulombs
Electrical Current	"A"	amperes
	"kA"	kiloamperes
	"uA"	microamperes
	"mA"	milliamperes
Voltage	"kV"	kilovolts
	"MV"	megavolts
	"mV"	millivolts
	"V"	volts
Resistance	"kohm"	kilohms
	"Mohm"	megohms
	"ohm"	ohms
Conductance	"mho"	mhos
Capacitance	"F"	farads
	"uF"	microfarads
	"mF"	millifarads
	"nF"	nanofarads
	"pF"	picofarads
Inductance	"H"	henrys
	"uH"	microhenrys
	"mH"	millihenrys
Magnetic Flux	"Wb"	webers
Magnetic Inductance	"G"	gauss
	"kG"	kilogauss
	"uG"	microgauss
	"mG"	milligauss
	"T"	teslas

Table A.6: Listing of defined units for electromagnetic quantities. Dagger (†) denotes *Rubber Airplane* internal units.

Appendix B

Component Geometry

B.1 Overview

As mentioned in Chapter 2, *Rubber Airplane* also provides a means for describing and displaying the three-dimensional geometry of design components. Component geometries are defined using the **defgeometry** macro, which associates a set of geometric objects with a specific component-class. The properties of these geometric objects—position, orientation, and dimensions—may be specified as constants, or related to the corresponding component-class's attributes. Geometry specifications are inherited, and, in cases where more than one of a component-class's superclasses have a geometry specification, the class precedence list (see Chapter 2) is used to determine which specification to apply.

A large number of three-dimensional object-types are provided, which may be combined as needed in constructing component geometries. These object-types include:

- boxes with arbitrary length, width, and height;
- cones with arbitrary cross-section and inclination;
- cylinders with arbitrary cross-section, inclination, and taper;
- cylinders with a different cross-section for each face;
- bodies of revolution with arbitrary cross-section;
- partial bodies of revolution with arbitrary cross-section and included angle;
- tori with arbitrary radius and cross-section; and
- partial tori with arbitrary radius, cross-section, and included angle.

In addition, so-called "symmetric cylinders" are also provided. Whereas normal cylinders are defined as the (possibly tapered) projection of a cross-section along the length of an

arbitrary vector, symmetric cylinders apply the projection along a reflected vector, as well. Symmetric cylinders are particularly useful in representing aircraft wing geometry.

As can be seen from the above description of object-types, many rely on description of one or more associated cross-sections in order to complete their specifications. For this reason, *Rubber Airplane* also provides mean for defining various types of two-dimensional curves to be associated with three-dimensional objects. Supported curve-types include:

- rectangles and ellipses;
- semi-rectangles and semi-ellipses;
- regular polygons;
- airfoil cross-sections based on NACA profiles; and
- generalized open and closed curves based on sets of scaled coordinates for the corner points.

Note that open curves are required for the definition of bodies of revolution. All other object-types which are based on cross-sections—cones, cylinders, and tori—employ closed curves. Like the properties of the three-dimensional objects with which they are associated, properties of cross-sectional curves (e.g., dimensions and scaling) may also be related to the attributes of the component-classes for whose geometry they are employed.

Once a geometry specification has been associated with a component-class, instantiation of the component-class also results in the creation of appropriate structures for displaying the component's geometry as a wireframe drawing via the *Rubber Airplane* user interface (see Appendix C). If the geometry is dependent upon the values of any of the component's attributes, the display is automatically updated whenever the values of these attributes are changed. Example geometry displays appear in Figures *insert figure references here*.

B.2 Definition of Component Geometries

As indicated above, geometry specifications are associated with component-classes by means of the **defgeometry** macro. The syntax for such specifications is as follows:

```
(defgeometry component-class
   (object-name object-type
      (property-keyword value-expression)
      (property-keyword value-expression)
      ...)
   (object-name object-type
```

```
    (property-keyword value-expression)
    (property-keyword value-expression)
    ...)
...)
```

The first argument to **defgeometry**, the *component-class* argument, should be a symbol naming the component-class with which the geometry specification is to be associated. This argument is followed by one or more lists of object specifications, each of which describes one of the three-dimensional objects which will be combined to represent the component's geometry.

The first element of each object specification, the *object-name* argument, should be a symbol representing the name of the three-dimensional object. Each of the objects comprising a component's geometry must have a unique *object-name*.[1] Next, the *object-type* argument must be specified. The value of this argument should be a symbol which identifies the type of geometric object which is to be employed. The object-types—also referred to as **geometry-classes**—which are currently supported are:

cone	**elliptical-cone**
cylinder	**two-curve-cylinder**
symmetric-cylinder	**two-curve-symmetric-cylinder**
body-of-revolution	**partial-body-of-revolution**
torus	**partial-torus**

box

As indicated above, "symmetric cylinders" project their cross-sections in two symmetric directions. "Two-curve" cylinders have a different cross-section at each end, with the cross-section varying linearly between the two curves along the axis of projection. The adjective "partial" is used here to refer to bodies of revolution and tori which do not revolve their cross-sections through a complete revolution about the axis of rotation, but only through some specified included angle. Finally, note that "elliptical cones" refer here to cones which vary the tapering of their cross-sections elliptically along the axis of projection, rather than linearly.

The remainder of each geometry object specification consists of pairings of *property-keyword* and *value-expression* arguments. The *property-keyword* arguments name the various properties associated with the geometry-class indicated by the specification's *object-type* argument, and the corresponding *value-expression* arguments are used to relate these properties to the attributes of the component-class whose geometry is being defined. The

[1] Currently, these *object-name* arguments serve no purpose other than code annotation. Eventually, they could be used to allow one object-specification to reference another. For instance, one object could be a reflection of another about some arbitrary plane of symmetry. The ability of a component-class's geometry objects to reference one another could also be used to simplify the positioning of adjacent objects.

Property	Description
`:position-x`	X-coordinate of the object's position.
`:position-y`	Y-coordinate of the object's position.
`:position-z`	Z-coordinate of the object's position.
`:rotation-z`	Rotation of the object's coordinate axes about the x-axis.
`:rotation-y`	Rotation of the object's coordinate axes about the y-axis.
`:rotation-x`	Rotation of the object's coordinate axes about the z-axis.

Table B.1: Properties common to all classes of geometric bodies.

properties associated with each of the supported object-types are listed in Tables B.1–B.4. Table B.1 lists the properties which are common to all geometry-classes. Table B.2 identifies those properties which are common to two sets of related geometry-classes: cones and cylinders, and tori and bodies of revolution. Tables B.3 and B.4 list the properties which are specific to the individual geometry-classes; in particular, Table B.4 lists those properties which are specific to "two-curve" cylinders.

For each property listed in the specification, a corresponding *value-expression* must be supplied. Three forms of *value-expression* arguments are supported. Each *value-expression* argument may assume one of three types of values:

- a constant,
- a symbol, or
- a LISP **lambda**-expression.

If a *value-expression* argument is expressed as a constant, then the corresponding property of the geometry-class is assigned the fixed value of this constant. If a symbol is provided, this symbol should name one of the attributes associated with the component-class whose geometry is being defined; the value of the corresponding property is equated with the value of the attribute.

In the LISP programming language, **lambda**-expressions provide a means for defining unnamed functions. The format of a **lambda**-expression is as follows:

(lambda (*arg1 arg2* ...)
body)

where *arg1*, *arg2*, ... are symbols naming the arguments to the function, and *body* is a sequence of LISP expressions which perform some series of computations based on the values of those arguments. In the context of **defgeometry** *value-expression* arguments,

Object Type	Properties	Description
Cones and Cylinders	:normal-x	X-component of the vector normal to the curve which forms the cross-section of the cone or cylinder.
	:normal-y	Y-component of the vector normal to the curve which forms the cross-section of the cone or cylinder.
	:normal-z	Z-component of the vector normal to the curve which forms the cross-section of the cone or cylinder.
	:base-x	X-component of the vector which points in the direction of the base of the curve which forms the cross-section of the cone or cylinder.
	:base-y	Y-component of the vector which points in the direction of the base of the curve which forms the cross-section of the cone or cylinder.
	:base-z	Z-component of the vector which points in the direction of the base of the curve which forms the cross-section of the cone or cylinder.
	:project-x	Distance along the x-axis by which the curve is to be projected.
	:project-y	Distance along the y-axis by which the curve is to be projected.
	:project-z	Distance along the z-axis by which the curve is to be projected.
	:root-twist	Rotation of the cross-section at the base of the cone or cylinder.
	:tip-twist	Rotation of the cross-section at the end of the cone or cylinder.
Bodies of Revolution and Tori	:revolve-x	X-component of the vector representing the axis of rotation for the body of revolution.
	:revolve-y	Y-component of the vector representing the axis of rotation for the body of revolution.
	:revolve-z	Z-component of the vector representing the axis of rotation for the body of revolution.
	:start-x	X-component of the vector indicating the direction in which the the first cross-section of the body of revolution should be drawn.
	:start-y	Y-component of the vector indicating the direction in which the the first cross-section of the body of revolution should be drawn.
	:start-z	Z-component of the vector indicating the direction in which the the first cross-section of the body of revolution should be drawn.

Table B.2: Properties shared by particular classes of geometric bodies.

Object Name	Properties	Description
box	:x-dimension	Length of the box parallel to the x-axis (prior to rotation).
	:y-dimension	Length of the box parallel to the y-axis (prior to rotation).
	:z-dimension	Length of the box parallel to the z-axis (prior to rotation).
cone, elliptical-cone	:curve	A closed curve for the cross-section of the cone.
cylinder, symmetric-cylinder	:curve	A closed curve for the cross-section of the cylinder.
	:taper	Fractional taper to be applied to the cross-section along the axis of projection.
body-of-revolution	:curve	An open curve for the cross-section of the body of revolution.
partial-body-of-revolution	:curve	An open curve for the cross-section of the body of revolution.
	:included-angle	Included angle of rotation from the starting cross-section to the final cross-section.
torus	:curve	A closed curve for the cross-section of the torus.
	:radius	Radius from the axis of rotation about which the cross-section is to be revolved.
partial-torus	:curve	A closed curve for the cross-section of the torus.
	:radius	Radius from the axis of rotation about which the cross-section is to be revolved.
	:included-angle	Included angle of rotation from the starting cross-section to the final cross-section.

Table B.3: Additional properties for specific geometry classes.

Property	Description
`:curve-1`	A closed curve for the cross-section of the base of the cylinder.
`:normal-x1`	X-component of the vector normal to the curve which forms the cross-section of the base of the cylinder.
`:normal-y1`	Y-component of the vector normal to the curve which forms the cross-section of the base of the cylinder.
`:normal-z1`	Z-component of the vector normal to the curve which forms the cross-section of the base of the cylinder.
`:base-x1`	X-component of the vector which points in the direction of the base of the curve which forms the cross-section of the base of the cylinder.
`:base-y1`	Y-component of the vector which points in the direction of the base of the curve which forms the cross-section of the base of the cylinder.
`:base-z1`	Z-component of the vector which points in the direction of the base of the curve which forms the cross-section of the base of the cylinder.
`:twist1`	Rotation of the cross-section of the base of the cylinder.
`:curve-2`	A closed curve for the cross-section of the end of the cylinder.
`:normal-x2`	X-component of the vector normal to the curve which forms the cross-section of the end of the cylinder.
`:normal-y2`	Y-component of the vector normal to the curve which forms the cross-section of the end of the cylinder.
`:normal-z2`	Z-component of the vector normal to the curve which forms the cross-section of the end of the cylinder.
`:base-x2`	X-component of the vector which points in the direction of the base of the curve which forms the cross-section of the end of the cylinder.
`:base-y2`	Y-component of the vector which points in the direction of the base of the curve which forms the cross-section of the end of the cylinder.
`:base-z2`	Z-component of the vector which points in the direction of the base of the curve which forms the cross-section of the end of the cylinder.
`:twist2`	Rotation of the cross-section of the end of the cylinder.

Table B.4: Additional properties for the `two-curve-cylinder` and `two-curve-symmetric-cylinder` geometry classes.

lambda-expressions are employed to relate the values of geometry object properties to the result of some arbitrary computation, to be performed based on the values of some subset of the attributes defined for the component-class with which the **defgeometry** specification is associated. The attributes to be related are indicating by naming them in the argument list (i.e., the *arg1*, *arg2*, ... elements) of the **lambda**-expression; the body of the expression is then applied to compute a value for the geometry object's property whenever the values of any of the corresponding attributes change.

B.3 Curve Specification

As indicated in Section B.1, a number of geometry-classes depend upon specification of a cross-section. Cross-sections are represented as properties of the appropriate geometry-classes, as indicated in Tables B.3 and B.4. The format for specifying cross-sectional properties is slightly different from that of other geometry-object properties:

(object-property-keyword curve-type
(curve-property-keyword value-expression)
(curve-property-keyword value-expression)
...)

As with normal property-specifications, the first element of the specification, the *object-property-keyword*, is a keyword naming the appropriate property of the geometry-class being specified. For cones, single-curve cylinders, bodies of revolution, and tori, the appropriate keyword for defining the cross-section is **:curve**. For two-curve cylinders, two cross-sections must be specified; thus, two keywords are required, **:curve-1** and **:curve-2**.[2]

The second element of the cross-section specification, the *curve-type* argument, should be a symbol naming the curve-type to be used for the cross-section. Two sets of curve-types are provided, one for representing closed curves and another for open curves. The curve-types which are currently supported for defining closed curves are:

airfoil-profile	**rectangle**	**ellipse**
regular-polygon	**square**	**pentagon**
hexagon	**octagon**	**circle**
	closed-curve	

The **airfoil-profile** curve-type is based upon a database of NACA airfoil thickness and mean-line distributions, developed in conjunction with the vortex-lattice aerodynamic analysis routines discussed in Chapter 4. The curve-types which are provided for defining open curves are:

[2]Note that the projection of the cylinder extends *from* the **:curve-1** cross-section *to* the **:curve-2** cross-section. Note also that it is required that the two curves which serve as the cross section of the two-curve cylinder must have the same number of corner points. Examples of valid pairings are: an ellipse and a circle, two airfoil profiles, and a square and a rectangle.

Curve Name	Properties	Description
airfoil-profile	:thickness	Thickness distribution structure.
	:mean-line	Mean-line distribution structure.
	:mean-line-scaling	Scaling of the mean-line, normal to the chord.
	:chord	Chord of the airfoil.
rectangle, ellipse	:width	Width of the rectangle or ellipse.
	:height	Height of the rectangle or ellipse.
regular-polygon	:number-of-sides	Number of sides for the regular polygon.
	:radius	Enclosing radius of the regular polygon.
square, pentagon, hexagon, octagon, circle	:radius	Enclosing radius of the polygon or circle.
closed-curve	:points	A list of sublists of the form (*x-coord y-coord*), representing a sequential specification of the corner points of a generalized closed curve.
	:scaling-x	Scaling factor to be applied to the x-coordinates appearing in the :points list.
	:scaling-y	Scaling factor to be applied to the y-coordinates appearing in the :points list.

Table B.5: Defined properties for closed curves.

semi-rectangle semi-ellipse semi-circle closed-curve

Note that the **closed-curve** and **open-curve** curve-types allow arbitrary curves, based upon scaled lists of corner points, to be defined. Recall also that, as indicated above, open curves are required for specifying cross-sections of bodies of revolution. All other geometry-classes which require cross-sections employ closed curves.

The remaining elements of the cross-section specification, the *curve-property-keyword* and *value-expression* pairings, allow the properties of the selected curve-type to be related to the values of the attributes of the component-class whose geometry is being defined. A listing of the properties associated with the various closed-curve curve-types is presented in Table B.5. Properties associated with the open-curve curve-types appear in Table B.6. The

Curve Name	Properties	Description
semi-rectangle, **semi-ellipse**	**:width**	Width of the semi-rectangle or semi-ellipse (i.e., the half-width of the corresponding full rectangle or ellipse).
	:height	Height of the semi-rectangle or semi-ellipse.
semi-circle	**:radius**	Radius of the semi-circle.
open-curve	**:points**	A list of sublists of the form (*x-coord y-coord*), representing a sequential specification of the corner points of a generalized open curve.
	:scaling-x	Scaling factor to be applied to the x-coordinates appearing in the **:points** list.
	:scaling-y	Scaling factor to be applied to the y-coordinates appearing in the **:points** list.

Table B.6: Defined properties for open curves.

value-expression arguments of the cross-section description are specified in the same manner as for object descriptions, using constants, attribute names, and **lambda**-expressions.

B.4 Examples

The first example component-class geometry definition, presented in Figure B.1, employs a body of revolution to represent the geometry of a fuselage-mounted radome. The definition of the component-class is included in Figure B.1, so that the attribute references in the **defgeometry** specification may be identified. A semi-ellipse is chosen as the cross-section for the body of revolution, whose dimensions are obtained from the **height** and **diameter** attributes of the **fuselage** component-class. The z-axis serves as the axis of revolution, and the first cross-section is drawn parallel to the x-axis. The geometry of an actual instance of this **radome** component-class is depicted in Figure B.2. (For further details on the *Rubber Airplane* window-based interface, see Appendix C).

A somewhat more complicated example appears in Figures B.3 and B.4. In Figure B.3, the definition of a **fuselage** component-class is presented. As indicated in the **defgeometry** specification presented in Figure B.4, three geometry-classes are combined in order to represent the geometry of this component-class. A cylinder is used to represent the cabin portion of the fuselage. The nose is represented by an elliptically-tapered cone. The tail section is represented by a conventional cone, which has been tilted upwards so that its upper surface is coplanar with the top of the cylinder used for the cabin section.

```
(defcomponent radome
       ((radius :documentation "Horizontal radius of the radome."
                :low-value 0.5 :order-of-magnitude 5 :high-value 10
                :value 3 :dimensions "1" :units "m")
        (diameter :documentation "Horizontal diameter of the radome."
                  :low-value 1 :order-of-magnitude 10 :high-value 20
                  :value 6 :dimensions "1" :units "m")
        (height :documentation "Vertical dimension of the radome."
                :low-value 0.1 :order-of-magnitude 1 :high-value 5
                :dimensions "1" :units "m"))
       (design-component)
  (:category structures surveillance)
  (:documentation
     "Attributes represent a radome with elliptical cross-section."))

(defconstraint (radome radome-diameter) (diameter "m") ((radius "m"))
  "Constrains the radome's diameter to be twice its radius."
  (* 2 radius))

(defgeometry radome
  (ellipsoid body-of-revolution
             (:curve semi-ellipse
                  (:width diameter)
                  (:height height))
             (:revolve-x 0) (:revolve-y 0) (:revolve-z 1)
             (:start-x 1) (:start-y 0) (:start-z 0)
             (:position-x position-x)
             (:position-y position-y)
             (:position-z position-z)
             (:rotation-x 0) (:rotation-y 0) (:rotation-z 0)))
```

Figure B.1: LISP definition for the radome component-class, including geometry specification.

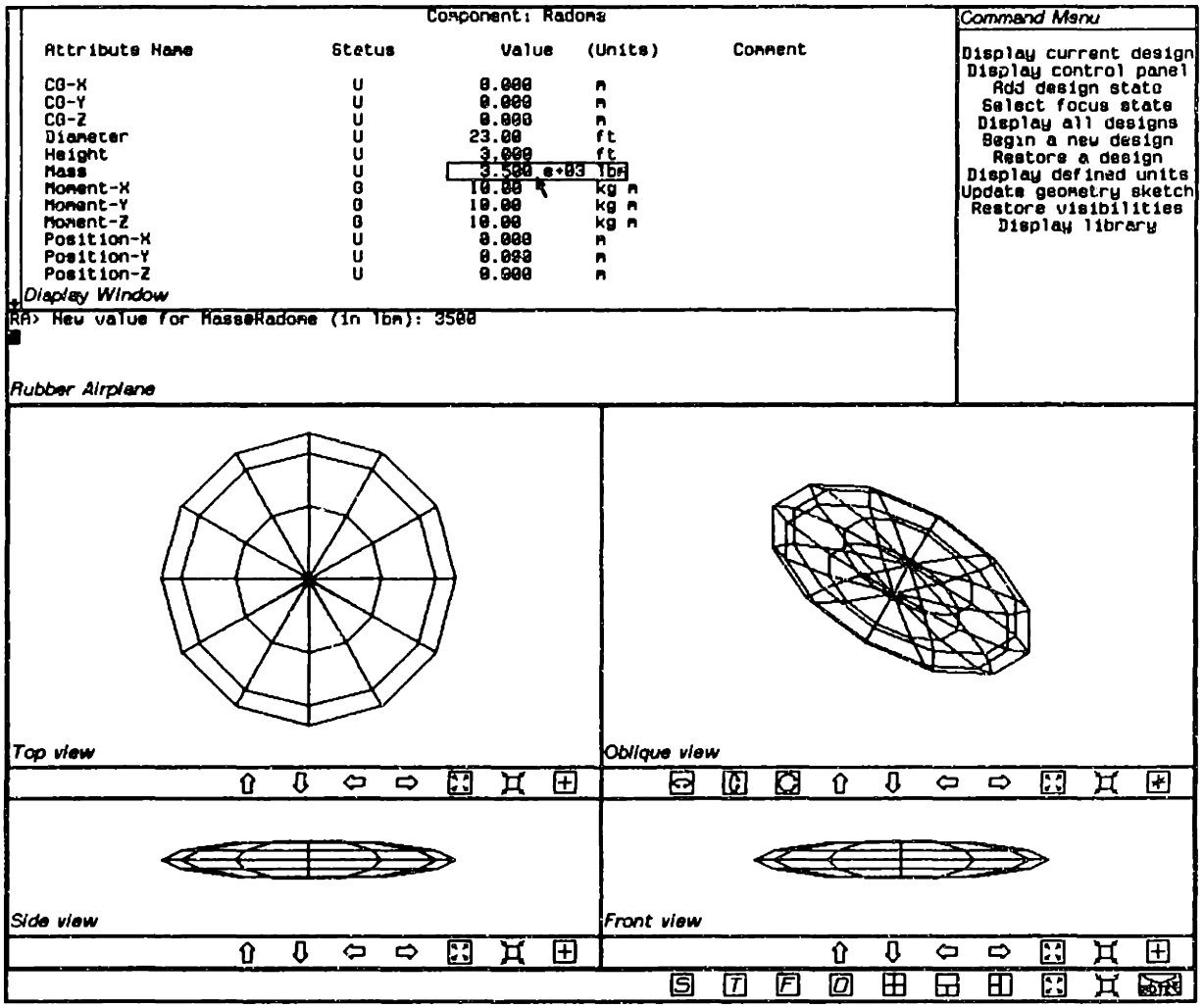

Figure B.2: Screen image depicting an instance of the **radome** comoponent-class.

Note that elliptical cross-sections have been chosen for each of the three geometry objects, the dimensions of which are derived from the **height** and **width** attributes of the **fuselage** component-class. The normal to each of these cross-sections is the x-axis; the base of each cross-section is in the direction of the z-axis. Each of the three geometry objects is projected along the x-axis according to the corresponding **cabin-length**, **nose-length**, and **tail-cone-length** attributes. A **lambda**-expression has been employed to tilt the cone representing the tail section in the z-direction, by a distance of half the **height** attribute. Note also the use of **lambda**-expressions to determine the positions of the various geometry objects, and to change the sign of the **nose-droop** attribute for the **:project-z** property of the elliptical cone which represents the nose of the fuselage. A screen image displaying an instance of the **fuselage** component-class is presented in Figure B.5.

```
(defcomponent fuselage
       ((cabin-length
          :documentation "Length of the cabin portion of the fuselage."
          :low-value 2 :order-of-magnitude 10 :high-value 50
          :dimensions "l" :units "m")
        (nose-length
          :documentation "Length of the nose portion of the fuselage."
          :low-value 0.5 :order-of-magnitude 5 :high-value 10 :value 3
          :dimensions "l" :units "m")
        (nose-droop
          :documentation
          "Vertical droop of the nose leading edge from the centerline."
          :low-value 0.5 :order-of-magnitude 1 :high-value 5 :value 0.75
          :dimensions "l" :units "m")
        (tail-cone-length
          :documentation "Length of the tail portion of the fuselage."
          :low-value 0.5 :order-of-magnitude 5 :high-value 10 :value 3
          :dimensions "l" :units "m")
        (total-length
          :documentation "Total length of the aircraft fuselage."
          :low-value 4 :order-of-magnitude 20 :high-value 70 :value 16
          :dimensions "l" :units "m")
        (width
          :documentation "Width of the aircraft fuselage."
          :low-value 1 :order-of-magnitude 5 :high-value 15 :value 2
          :dimensions "l" :units "m")
        (height
          :documentation "Height of the aircraft fuselage."
          :low-value 1 :order-of-magnitude 5 :high-value 15 :value 2
          :dimensions "l" :units "m"))
       (design-component)
  (:category structures fuselage)
  (:documentation "Attributes and constraints represent a simple
        cylindrical aircraft fuselage."))

(defconstraint (fuselage length) (total-length "m")
       ((cabin-length "m") (nose-length "m") (tail-cone-length "m"))
  "Calculates the total length of the fuselage based on the length of the
cabin, nose, and tail-cone sections."
  (+ cabin-length nose-length tail-cone-length))
```

Figure B.3: LISP definition for the **fuselage** component-class.

```
(defgeometry fuselage
  (cabin cylinder
         (:curve ellipse (:width width) (:height height))
         (:normal-x 1) (:normal-y 0) (:normal-z 0)
         (:base-x 0) (:base-y 0) (:base-z 1)
         (:taper 1)
         (:project-x cabin-length) (:project-y 0) (:project-z 0)
         (:root-twist 0) (:tip-twist 0)
         (:position-x (lambda (position-x cabin-length)
                        (- position-x (/ cabin-length 2))))
         (:position-y position-y) (:position-z position-z)
         (:rotation-x 0) (:rotation-y 0) (:rotation-z 0))
  (tail-cone cone
             (:curve ellipse (:width width) (:height height))
             (:normal-x 1) (:normal-y 0) (:normal-z 0)
             (:base-x 0) (:base-y 0) (:base-z 1)
             (:project-x (lambda (tail-cone-length) (- tail-cone-length)))
             (:project-y 0) (:project-z (lambda (height) (/ height 2)))
             (:root-twist 0) (:tip-twist 0)
             (:position-x (lambda (position-x cabin-length)
                            (- position-x (/ cabin-length 2))))
             (:position-y position-y) (:position-z position-z)
             (:rotation-x 0) (:rotation-y 0) (:rotation-z 0))
  (nose elliptical-cone
        (:curve ellipse (:width width) (:height height))
        (:normal-x 1) (:normal-y 0) (:normal-z 0)
        (:base-x 0) (:base-y 0) (:base-z 1)
        (:project-x nose-length) (:project-y 0) (:project-z 0)
        (:root-twist 0) (:tip-twist 0)
        (:position-x (lambda (position-x cabin-length)
                       (+ position-x (/ cabin-length 2))))
        (:position-y position-y) (:position-z position-z)
        (:rotation-x 0) (:rotation-y 0) (:rotation-z 0)))
```

Figure B.4: LISP specification of the geometry for the **fuselage** component-class.

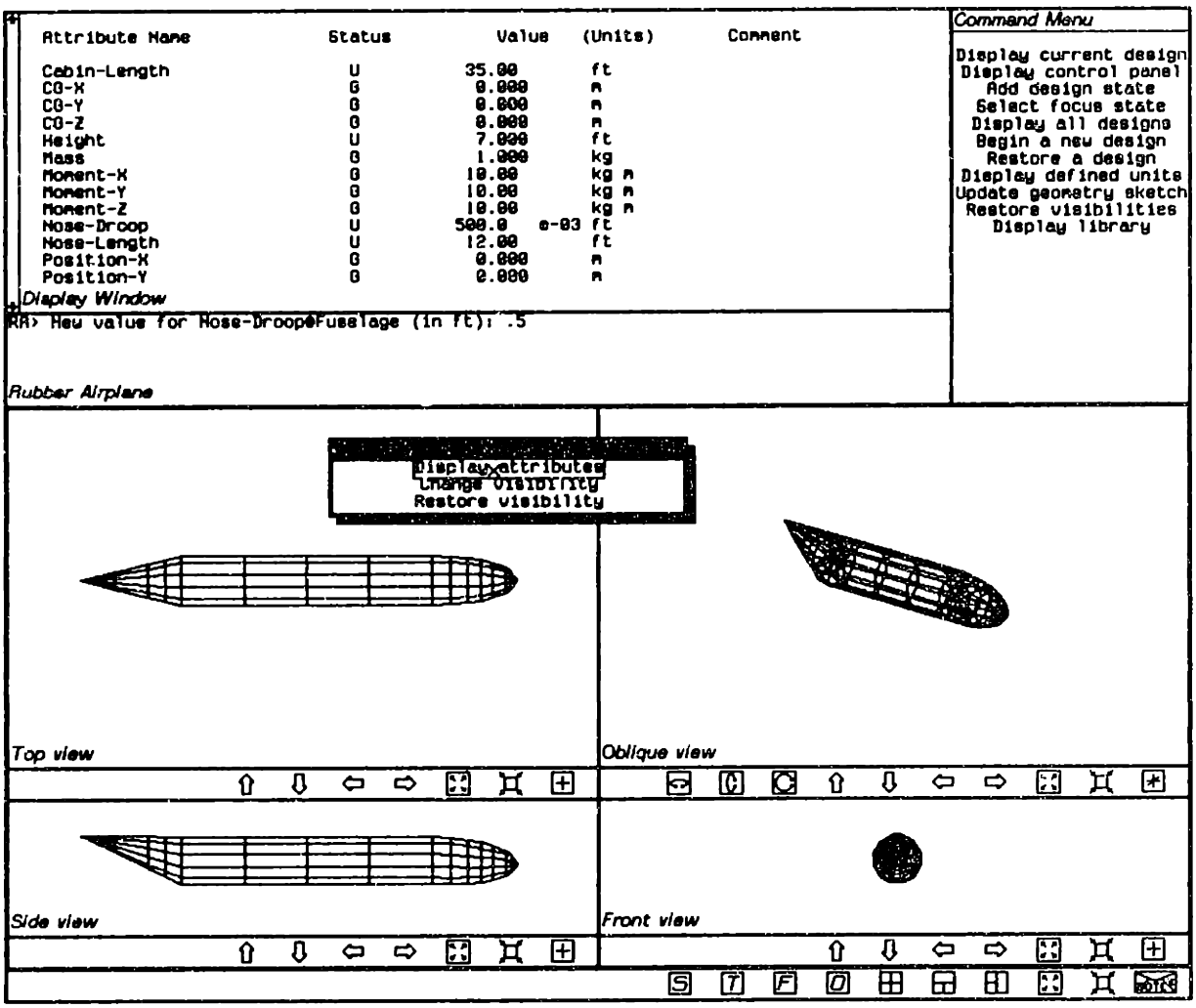

Figure B.5: Screen image depicting an instance of the **fuselage** component-class.

Appendix C

User Interface

C.1 Overview

As indicated in previous chapters, definitions of design-entity classes are specified using a text editor, such as the Lisp Machine Zmacs editor. Once these classes have been defined, however, the *Rubber Airplane* window-based interface may be employed to create and modify instances of these classes when modeling a design problem. While a complete description of this interface is not appropriate here, this appendix is intended to provide a brief account of its general features.

The *Rubber Airplane* interface consists of five basic elements:

- a command menu, for initiating various top-level tasks;
- a scroll window, for displaying textual information;
- an interaction window, for providing messages to the user, and for obtaining input from the user based on program prompts;
- a three-dimensional geometry display, for graphically depicting the components of a design; and
- the standard Lisp Machine "who line" window, for documenting the available mouse-based commands, based on the current position of the mouse.

The location of these various elements in the standard *Rubber Airplane* screen configuration is shown in Figure C.1. In the sections which follow, the role of each of these interface elements is discussed.

C.2 The Who Line

The "who line" is a basic part of the interface of all window-based applications on the Lisp Machine [33]. It provides a standard means for providing on-screen documentation

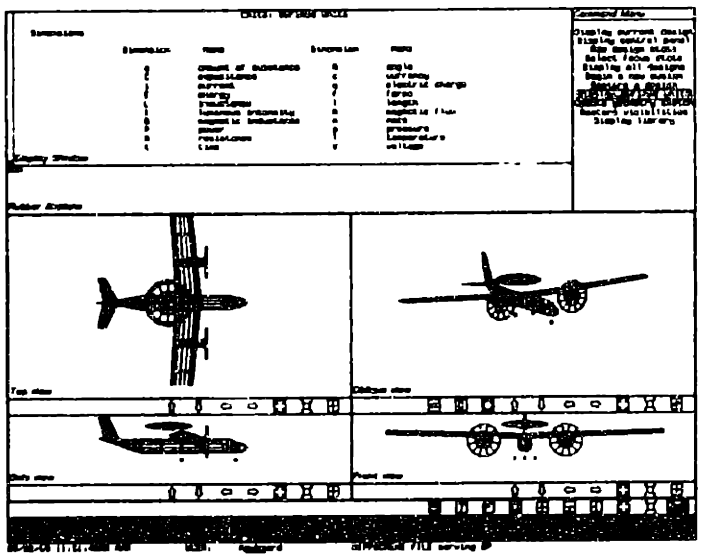

Figure C.1: Standard *Rubber Airplane* screen configuration.

for mouse-based commands. The first line of the who line describes the actions which will result from clicking each of the three buttons on the mouse, based on the current location of the mouse. The second line allows the display of a additional documentation, as needed (see, for example, Section C.6, below). As the mouse is moved over different elements of the interface, the who-line documentation is updated as necessary to reflect the available mouse-based commands corresponding to its changing position.

C.3 Command Menu

The command-menu window, in the upper right-hand corner of the screen image in Figure C.1, allows the user immediate access to a number of top-level operations. Menu items are selected by pointing at them with the mouse, and clicking one of the three mouse-buttons. Whenever the mouse is over one of the menu-items, corresponding documentation is displayed in the who line.

Display current design. Selecting this first item in the command menu brings up a listing—in the scroll window (see Section C.4)—of all of the component-, state-, and link-instances associated with the current design.[1]

[1] As indicated in Chapter 2, *Rubber Airplane* always maintains one of its active designs as the "current design", so that design-specific operations, such as geometry display and focus-state selection, need not require explicit specification of a design.

Display control panel. Selecting this item causes the scroll window to display the settings of various control switches for the current design. For each design, four such switches are available:

Automatic Geometry Updating: This switch controls whether or not the geometry display (see Section C.6) should automatically be updated whenever the values of the relevant component attributes change. The default setting for this switch is "on".

Verbose Geometry Updating: If this switch is activated, a message is displayed in the interaction window (see Section C.5) whenever a component's geometry is updated. Initially, this switch is turned off.

Interactive Loop Detection: This switch controls whether or not interaction loop detection (and solution), as discussed in Chapter 3, should take place. The default setting for this switch is "on".

Verbose Loop Detection: If this switch is activated, a message is displayed in the interaction window whenever a computational loop has been detected. The message lists the attributes and constraints which form the loop. Initially, this switch is turned off.

From the control panel display, the settings of these control switches may be toggled.

Add design state. By selecting this menu item, the user may add a new `design-state` instance to the current design. The user is prompted for a name for the new design state, as well as its position in the ordered list of all the design states associated with the current design.

Select focus state. This menu item allows the user to select any one of the current design's design states as the focus state (see Chapter 2). Selecting this menu item with the left mouse-button makes the design state which follows the current focus state in the design's order list of states the new focus state. Clicking the middle mouse-button chooses the state which precedes the current focus state as the new focus state. If the right mouse-button is pressed, a menu of all of the design's design states is presented, from which the desired focus state may be selected.

Display all designs. When this menu item is chosen, a listing of all of the designs known to *Rubber Airplane* is displayed in the scroll window. This listing is mouse-sensitive, and selection of a design from this listing will make it the current design. From this listing, the user may also request that the contents and/or status of a design be copied into a file, from which this information may be restored for use in subsequent *Rubber Airplane* sessions.

Begin a new design. Selecting this item prompts the user for the name of a new *Rubber Airplane* design, which is then made the current design.

Restore a design. When this item is selected, the user is prompted for the name of a file, from which a design may be loaded. The restored design then becomes *Rubber Airplane*'s current design.

Display defined units. Selecting this item causes a listing of all defined dimensions and units to be displayed in the scroll window. (Note that this is the selected menu item in Figure C.1.)

Update geometry sketch. If the *Automatic Geometry Updating* switch (see above) has been disabled, selecting this menu item causes the geometry display for all of the current design's components to be updated.

Restore visibilities. As indicated below, the display of each component's geometry may be disabled by the user. Selecting this menu item re-enables the geometry display for all components of the current design.

Display library. Selecting this item brings up a listing of the top-level category in the *Rubber Airplane* library of design-entity classes. From this listing, the various sub-categories may be displayed. For each category, the classes which have been defined for it are also displayed; selecting a class's name with the mouse allows the class to be instantiated. New design-entity instances are added to the current design.

C.4 Scroll Window

The scroll window enables the display of mouse-sensitive textual data in a scrolling field. If an item of text is mouse-sensitive, moving the mouse over it will cause it to be highlighted: a rectangular box will be drawn around it. In addition, appropriate documentation will be displayed in the who line, describing the corresponding results of clicking each of the mouse-buttons while over the highlighted text.

Various panels of information are available for display in the scroll window. As indicated in the preceding section, a listing of all active designs may be displayed, as may be the design entities of a single design. Similarly, a listing of defined dimensions and units may be displayed. In addition, component-, state-, and link-instances, as well as their individual attributes and constraints, may also be displayed.

Note that, in addition to the standard window configuration, as depicted in Figure C.1, the scroll window may be enlarged by removing the geometry display. This alternative configuration appears in Figure C.2.

In general, each of the three mouse-buttons has its own unique binding for each mouse-sensitive item within the scroll window. Here, the term "binding" is used to refer to the action which results from clicking a mouse-button. Thus, when a given item is highlighted, each mouse-button yields a different result when pressed. There is, however, a set of con-

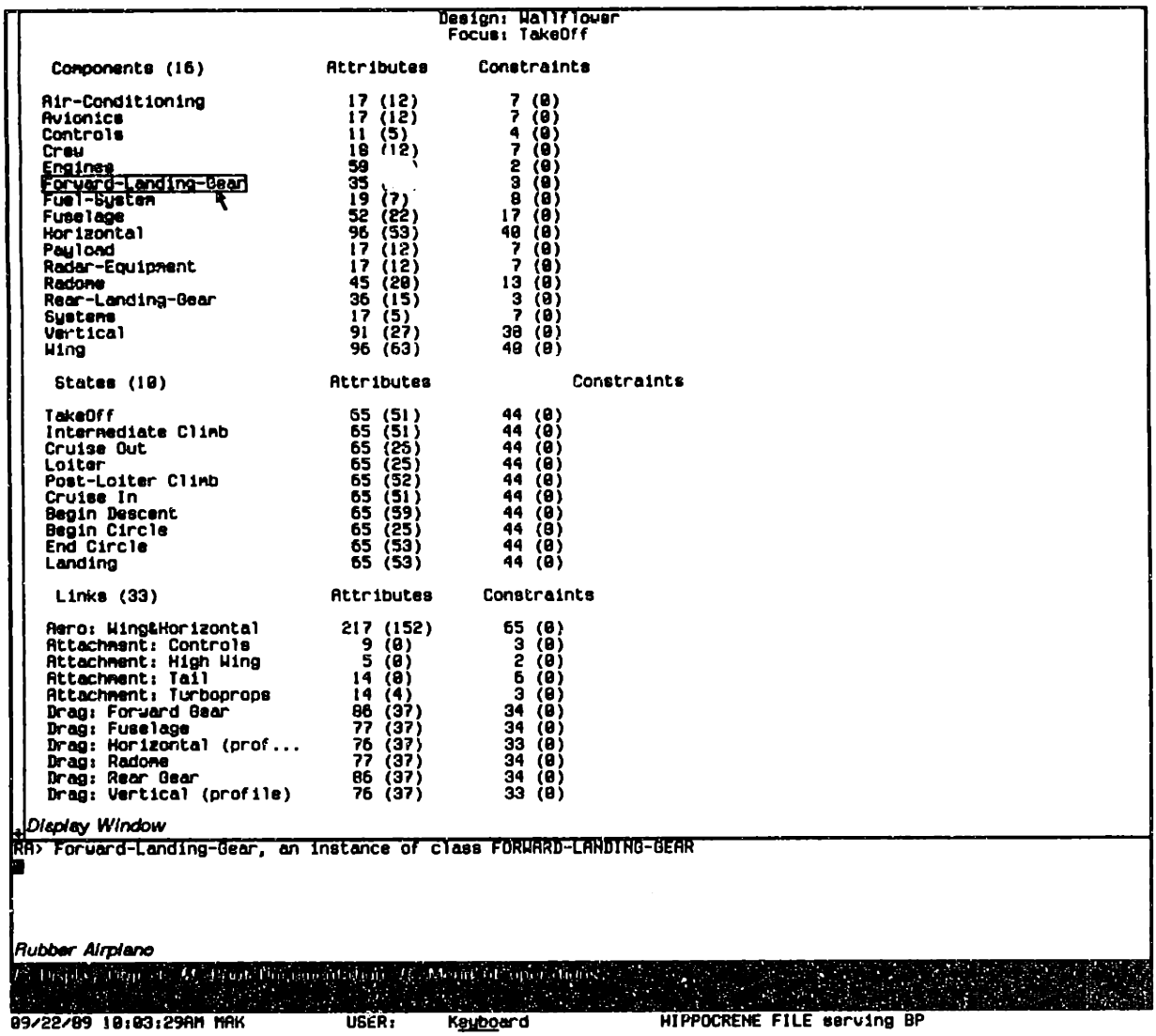

Figure C.2: Alternative interface configuration, employing an enlarged scroll window.

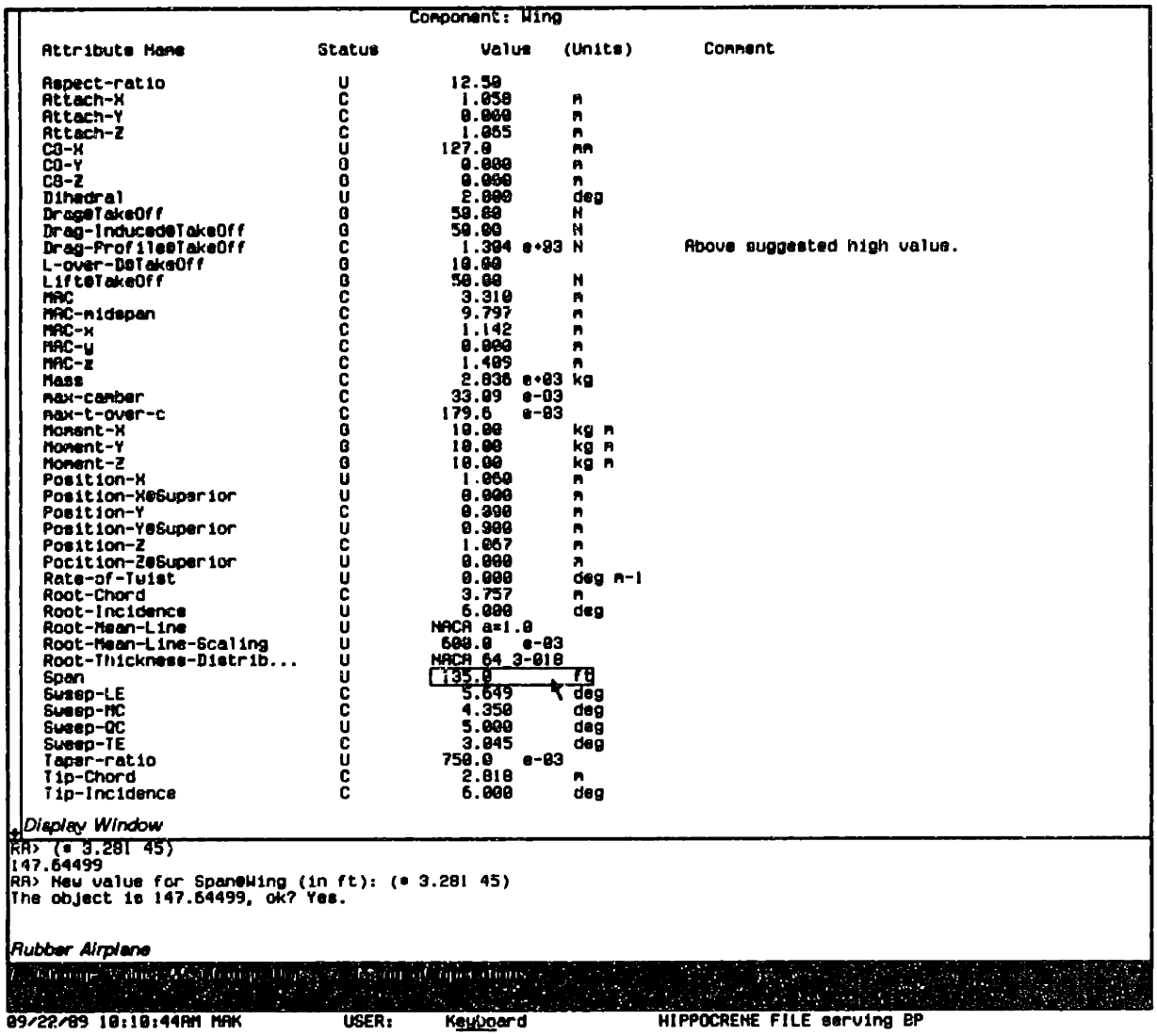

Figure C.3: Use of the scroll window and the left mouse-button to change the value of a scalar attribute.

ventional *types* of actions which most scroll-window mouse-sensitive items support. For example, clicking the right mouse-button almost always presents the user with a menu of all possible actions which may be performed on the object or property associated with the selected item. The middle mouse-button is primarily used for displaying the documentation associated with objects. This use of the middle mouse-button is illustrated in Figure C.2: in displaying a design, the middle mouse-button has been pressed in order to display the documentation for the design's **Forward-Landing-Gear** component in the interaction window.

In the case of items which present the name of an object, the left mouse-button allows the user to display additional information about the indicated object. For example, clicking on the name of a component when displaying a design will cause the component display

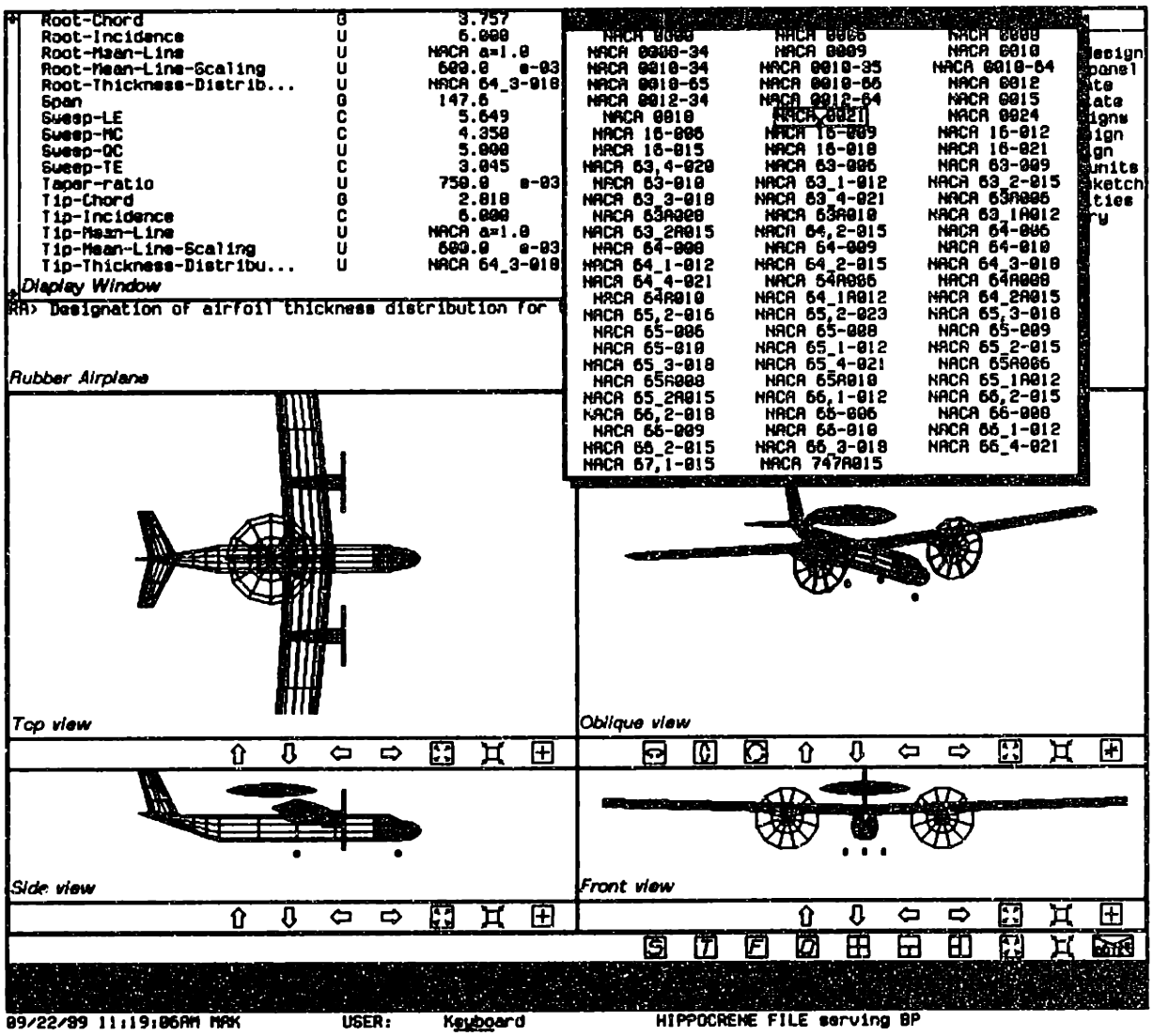

Figure C.4: Use of the scroll window and the left mouse-button to change the value of a discrete attribute.

to be replaced by a listing of the component's attributes. (Note, for example, the who-line documentation for the left mouse-button in Figure C.2.) In the context of properties of objects, the left mouse-button is typically employed to allow the user to change the value of that property. For example, clicking on the value of an attribute prompts the user for a new value. Figure C.3 illustrates the use of the right mouse-button to change the value of a scalar attribute; the user is prompted for a new value for the attribute in the interaction window. Figure C.4 illustrates its use in changing the value of a discrete attribute; in this case, the user is presented with a menu of possible values, from which a new value for the attribute may be selected.

As mentioned above, constraints may also be displayed via the scroll window. One noteworthy feature of such constraint displays is the ability of the user to assign so-called "trial values" to the attributes associated with the displayed constraint. The constraint itself may then be solved using these trial values, but the results are *not* propagated through the constraint network. Trial values thus allow constraints to be tested in isolation, for varying values of their parameters. An example constraint display is presented in Figure C.5, where the trial value of the constraint's **Span** attribute has been modified.

C.5 Interaction Window

As indicated in the preceding sections, the interaction window acts as a means for displaying short messages, and for prompting the user for input. In Figure C.2, the interaction window has been used to display the documentation string for a component-instance. In Figure C.3, the interaction window has been used to read in a new value for a scalar attribute.

In addition, the interaction window also incorporates a standard Lisp read-eval-print loop, enabling it to serve as a Lisp Listener window, as well: Lisp expressions typed into the interaction window are automatically evaluated, and the results are displayed. Use of the interaction window as a Lisp Listener is also illustrated in Figure C.3, where the expression `(* 3.281 45)` has been evaluated prior to setting the scalar attribute's value. Indeed, this expression has then been re-used in entering the new value for the attribute.[2]

C.6 Geometry Display

As discussed in Appendix B, *Rubber Airplane* provides macros for describing the geometry of design components in terms of their attributes. The geometry display portion of the *Rubber Airplane* interface is used to present these component geometries to the user. Four

[2]Note that not all prompts in the interaction window will accept LISP expressions as input; the scalar-value prompt is a special case. When not prompting for input, however, the interaction window always behaves as a normal Lisp Listener.

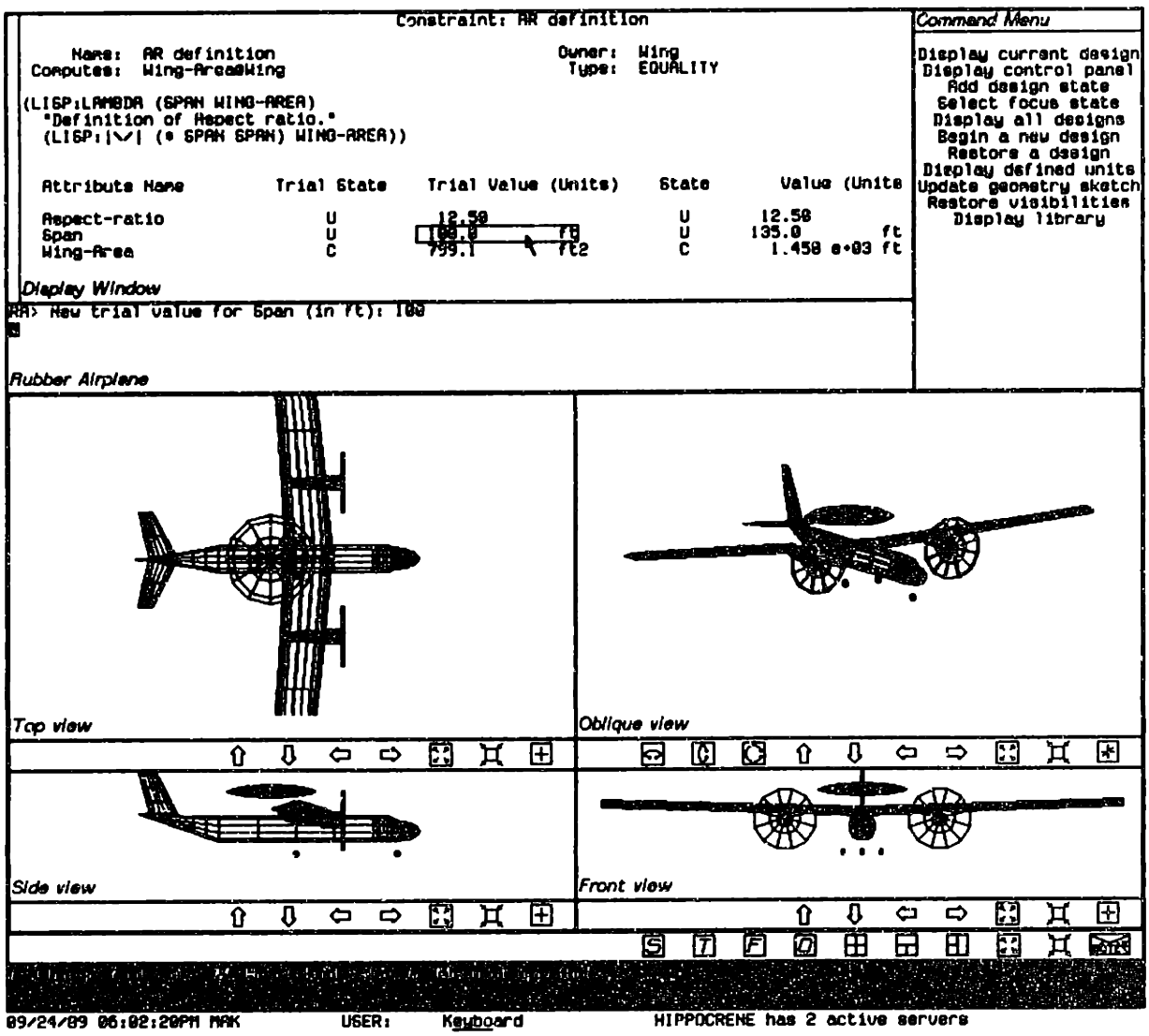

Figure C.5: Use of the scroll window to display a constraint.

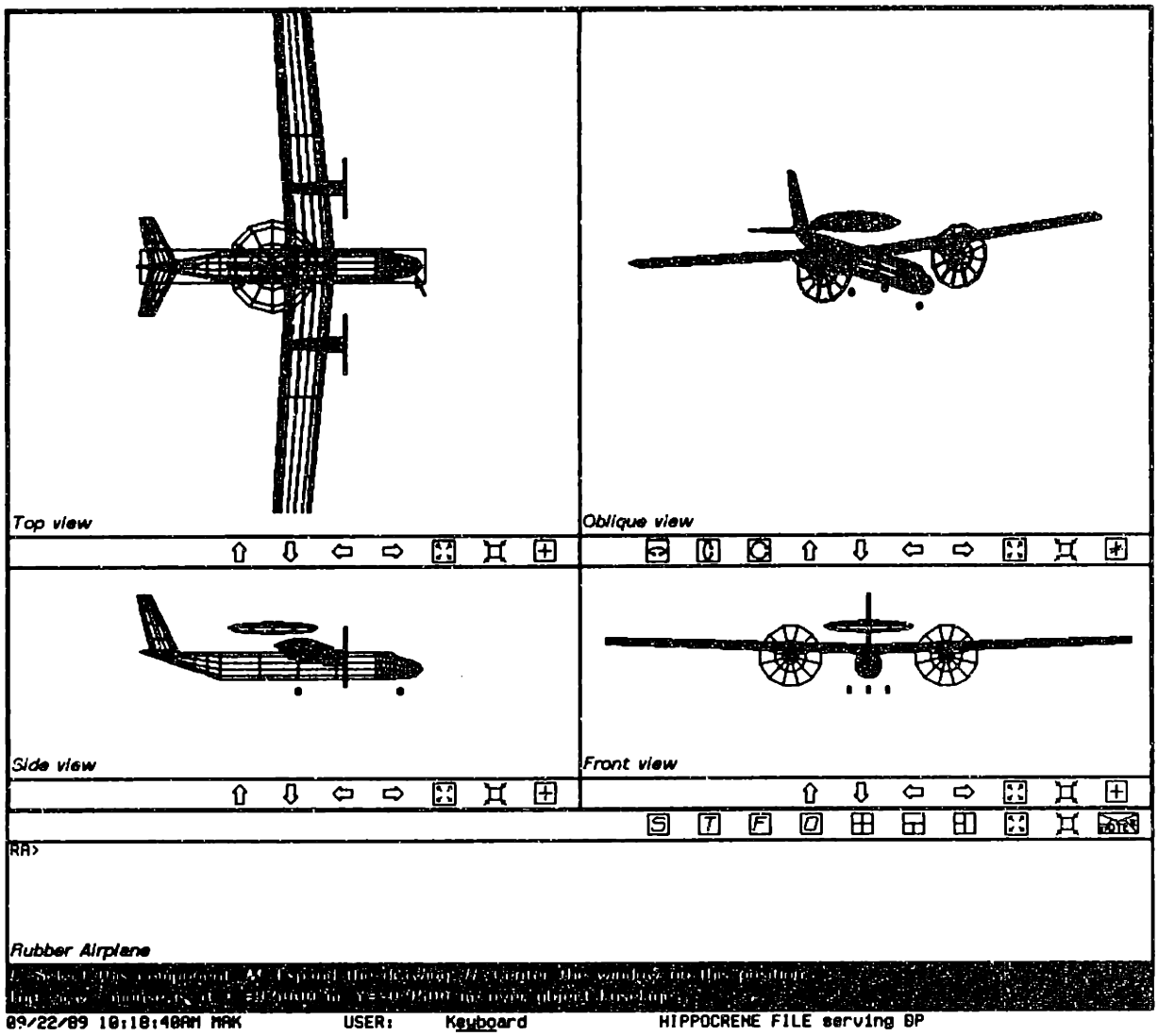

Figure C.6: Alternative interface configuration, employing an enlarged geometry display.

different views of a design's three-dimensional geometry may be displayed simultaneously, showing three orthogonal views (front, side, and top), and one oblique view. These visual images provide the designer with additional means for judging the merits of a given design solution. Note that a screen configuration which enlarges the geometry display by hiding the scroll window is available; this alternative configuration is depicted in Figure C.6.

This geometry display also supports additional interface features, beyond simple image display. For example, icons are provided below each of the four view-specific windows, for panning and zooming the displays, as well as for changing the orientation of the oblique view. A set of icons at the bottom of the geometry display allows the views to be zoomed simultaneously, and allows one or more views to be hidden, so that the remaining view(s) may be enlarged. Furthermore, as the mouse is moved over the various displays, its position

is tracked in the who line. When the mouse is over an orthogonal view, the coordinates corresponding to its current location are indicated; when the mouse is over the oblique view, the current rotation angles are displayed. In addition, when the mouse is over a specific component, the component's geometry is highlighted by drawing a rectangular box around it, and the name of the component appears in the who line. For example, in Figure C.6, the mouse has been positioned over the design's **Fuselage** component, in the top-view display. A rectangular box has been drawn around the component, and both the component's name and the x- and y-coordinates of the mouse's position appear in the who line.

Clicking the mouse while it is over the geometry display also has various effects. Clicking the middle mouse-button switches the screen configuration between the standard configuration (see Figure C.1) and the configuration in which the geometry display is enlarged (see Figure C.6). Clicking the right mouse-button causes the view to pan such that the view becomes centered on the mouse's current position. If the mouse is over a component, clicking the left mouse-button presents a menu of operations which may be performed on the component. (Note that this menu may be seen in Figure B.5 on page 219 of Appendix B.) This menu allows the user to change the visibility of the component within the geometry display—to suppress or restore its appearance—and to cause the component's attributes to be displayed in the scroll window.

Bibliography

[1] Ira H. Abbott and Albert E. von Doenhoff. *Theory of Wing Sections.* Dover Publications, Inc., New York, 1959.

[2] Cas Bil. *ADAS Command Reference Manual.* Department of Aerospace Engineering, Delft University of Technology, Delft, April 1985.

[3] D. G. Bobrow, L. G. DeMichiel, R. P. Gabriel, S. Keene, G. Kiczales, and D. A. Moon. A Common LISP object system specification. ANSI X3J13 Document 87–002, American National Standards Institute, Washington, DC, March 1987.

[4] Richard Bogen. *MACSYMA Reference Manual, Version 10.* Mathlab Group, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA, January 1983.

[5] Alan Borning. *Thinglab—A Constraint-Oriented Simulation Laboratory.* PhD thesis, Stanford University, Stanford, CA, March 1979. A revised version is available as Xerox Palo Alto Research Center Report SSL–79–3 (July 1979).

[6] Bruce G. Buchanan. Expert systems: Working systems and the research literature. *Expert Systems*, 3(1):32–51, January 1986.

[7] Kathryn M. Chalfan. A knowledge system that integrates heterogeneous software for a design application. *The AI Magazine*, VII(2):80–84, Summer 1986.

[8] Randall Smith (Lockheed-Georgia Company). Personal communications, June–August 1986.

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

[10] O.J. Dahl and K. Nygaard. Simula—an algol-based simulation language. *Communications of the ACM*, 9:671–678, 1966.

[11] Antonio L. Elias. Knowledge engineering of the aircraft design process. In Janusz S. Kowalik, editor, *Knowledge Based Problem Solving*, chapter 6. Prentice-Hall, Englewood Cliffs, NJ, 1986.

[12] G. A. Gabriele and K. M. Ragsdell. *OPT: A Nonlinear Programming Code in FORTRAN IV—User's Manual.* Purdue Research Foundation, Lafayette, IN, January 1976.

[13] Martin Gardiner. The superellipse: A curve that lies between the ellipse and the rectangle. *Scientific American*, 213(3):222–232, September 1965.

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

[15] James Gosling. *Algebraic Constraints.* PhD thesis, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA, May 1983.

[16] Thomas J. Gregory. Computerized preliminary design at the early stages of vehicle definition. In *Aircraft Design Integration and Optimization.* AGARD, June 1974. AGARD Conference Proceedings No. 147.

[17] Mark Donald Gross. *Design as Exploring Constraints.* PhD thesis, Department of Architecture, Massachusetts Institute of Technology, Cambridge, MA, February 1986.

[18] L. R. Jenkinson and D. Simos. A computer program for assisting in the preliminary design of twin-engined propeller-driven general aviation aircraft. *Canadian Aeronautics and Space Journal*, 30(3):213–224, September 1984.

[19] R. S. Justice, A. L. Harcrow, and C. E. Izurieta. SGASP version 3.0 documentation. Lockheed Report LG87ER0171, Lockheed Aeronautical Systems Co., October 1987. Volume 1: User Guide, Volume 2: Developer Guide.

[20] Mark A. Kolb. On the numerical solution of simultaneous non-linear equations in computer-aided preliminary design. Master's thesis, Department of Aeronautics and Astronautics, Massachusetts Institute of Technology, Cambridge, MA, January 1986.

[21] Ronnie M. Lajoie. Integration of engineering models in computer-aided preliminary design. Master's thesis, Department of Aeronautics and Astronautics, Massachusetts Institute of Technology, Cambridge, MA, January 1987.

[22] C. Edward Lan. A quasi-vortex-lattice method in thin wing theory. *Journal of Aircraft*, 11(9):518–527, September 1974.

[23] Wm. Leler. *Constraint Programming Languages: Their Specification and Generation.* Addison-Wesley Publishing Company, Reading, MA, 1988.

[24] Michael P. Lukas and R. Brooks Pollock. Automated design through artificial intelligence techniques. Technical Paper WTP88–1, Wisdom Systems, Chagrin Falls, Ohio, May 1988.

[25] Marvin Minsky. A framework for representing knowledge. In Patrick H. Winston, editor, *The Psychology of Computer Vision*, chapter 6, pages 211–277. McGraw-Hill Book Co., New York, 1975.

[26] Alan R. Mitchell. Market supremacy through engineering automation. *Aerospace America*, 25(1):24–27, January 1987.

[27] Jack Moran. *An Introduction to Theoretical and Computational Aerodynamics.* John Wiley & Sons, Inc., New York, 1984.

[28] Alex P. Pentland. Toward an ideal 3-D CAD system. In *SPIE Conference on Machine Vision and the Man-Machine Interface*, San Diego, CA, January 1987. Society of Photo-Optical Instrumentation Engineers. SPIE Order No. 758-20.

[29] Lawrence W. Rosenfeld and Avrum P. Belzer. Breaking through the complexity barrier. Technical report, ICAD Inc., Cambridge, MA, January 1986.

[30] Mark Sapossnek. MARKSYMA: A parametric design tool based upon the analytic and numeric solution of systems of equations. Master's thesis, Department of Mechanical Engineering, Rensselaer Polytechnic Institute, Troy, NY, December 1985.

[31] David Serrano. MATHPAK: An interactive preliminary design package. Master's thesis, Department of Mechanical Engineering, Massachusetts Institute of Technology, Cambridge, MA, January 1984.

[32] David Serrano. *Constraint Management in Conceptual Design.* PhD thesis, Department of Mechanical Engineering, Massachusetts Institute of Technology, Cambridge, MA, October 1987.

[33] Richard M. Stallman, David Moon, and Daniel Weinreb. *Lisp Machine Window System Manual.* Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, MA, August 1983.

[34] Guy L. Steele Jr. *The Definition and Implementation of a Computer Programming Language Based on Constraints.* PhD thesis, Department of Electrical Engineering and Computer Sciences, Massachusetts Institute of Technology, Cambridge, MA, July 1980.

[35] Guy L. Steele Jr. *Common LISP: The Language.* Digital Press, Burlington, MA, 1984.

[36] Ivan E. Sutherland. Sketchpad, a man-machine communication system. In *Proceedings of the Spring Joint Computer Conference*, Detroit, MI, May 1963.

[37] Siu Shing Tong. Coupling symbolic manipulation and numerical simulation for complex engineering designs. In *Conference on Expert Systems for Numerical Computing*. International Association of Mathematics and Computers in Simulation, December 1988.

[38] Egbert Torenbeek. *Synthesis of Subsonic Airplane Design*. Delft University Press, Delft, 1982.

[39] Daniel Weinreb and David Moon. Flavors: Message passing in the Lisp Machine. AI Memo 602, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, MA, November 1980.

[40] Stephen Wolfram. *Mathematica: A System for Doing Mathematics by Computer*. Addison-Wesley Publishing Company, Reading, MA, 1988.