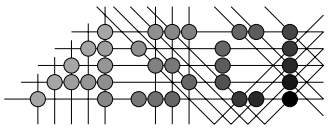
Constraint-Based Feature Validation



Advanced School for Computing and Imaging

This work was carried out in graduate school ASCI.
ASCI dissertation series number 24.

Constraint-Based Feature Validation

PROEFSCHRIFT

ter verkrijging van de graad van doctor
aan de Technische Universiteit Delft
op gezag van de Rector Magnificus Prof.dr.ir. J. Blaauwendraad
in het openbaar te verdedigen ten overstaan van een commissie,
door het College voor Promoties aangewezen,
op dinsdag 20 januari 1998 om 13:30 uur
door

Maurice Hubertus Petrus Johannes DOHMEN

informatica ingenieur
geboren te Geleen

Dit proefschrift is goedgekeurd door de promotor:
Prof.dr.ir. F.W. Jansen

Toegevoegd promotor:
Dr. W.F. Bronsvoort

Samenstelling promotiecommissie:
Rector Magnificus, voorzitter
Prof.dr.ir. F.W. Jansen, Technische Universiteit Delft, promotor
Dr. W.F. Bronsvoort, Technische Universiteit Delft, toegevoegd promotor
Prof.dr. I. Horváth, Technische Universiteit Delft
Dr.ir. F.J.A.M. van Houten, Universiteit Twente
Prof.dr. W. Gerhardt, Technische Universiteit Delft
Prof.dr. B. Brüderlin, Technische Universität Ilmenau
Dr. M.J. Pratt, Rensselaer Polytechnic Institute en National Institute of Standards and Technology
Prof. J.M. Hennessey, Technische Universiteit Delft, reservelid

Typeset in GillSans

CIP-DATA KONINKLIJKE BIBLIOTHEEK, DEN HAAG

Dohmen, Maurice H.P.J.

Constraint-based feature validation / Maurice H.P.J. Dohmen. - [S.l. : s.n.]. Ill.
Thesis Technische Universiteit Delft. - With ref. - With summary in Dutch.
ISBN 90-9011285-5
NUGI 855
Subject heading: feature validation / constraint satisfaction / feature modeling

© **Copyright 1997 by M.H.P.J. Dohmen**
All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior written permission of the author.

Preface

The research described in this thesis was carried out at the Laboratory of the Computer Graphics & CAD/CAM group of Delft University of Technology. Building on their research experience in geometric modeling, my promotors, Erik Jansen and Wim Bronsvoort, initiated a series of PhD research projects in our group that exploit the emerging area of feature modeling. Winfried van Holland was the first PhD student in this series, and focused on assembly features. I focused on constraint-based feature validation, an area in which I could further develop my interest in constraints, which began during my study. I carried out my project in cooperation with Klaas Jan de Kraker, who focused on feature conversion for concurrent engineering. As a team, we created a completely new multiple-view feature modeling system. During our research period, Rafael Bidarra joined our team, to work on feature interaction validation. Recently, Alex Noort started a research project on enhanced feature views. Together, they continue the feature modeling research.

Various people of the Computer Graphics & CAD/CAM group contributed in a variety of ways to my research. I wish to thank them all for their contributions, and for the pleasant working environment they provided. In particular, I would like to mention a few of them with whom I closely cooperated.

Klaas Jan de Kraker always was a sparring partner for me. Our close cooperation began with the first ideas for the design of SPIFF, and has now come to an end after we have proofread each other's thesis. He never hesitated to remind me that, whenever my ideas were becoming too speculative, I should be more concise and practical.

My supervisor Wim Bronsvoort taught me how to perform research in a precise way; his ideas and comments were a valuable source of inspiration for me. I enjoyed the many discussions we had about topics of my research, and perhaps even more, those about more personal topics.

In the feature modeling research project, Winfried van Holland and Rafael Bidarra provided a good source of feedback. Kees Seebregts and Aadjan van der Helm integrated the different SPIFF software modules. Graduate students contributed important parts of the software. Arnout Kuiper and Alex Noort built the geometric constraint solver. Bart Vergouwe made the graphical user interface. Jan Wouter Versluis developed the feature visualization software. I

would like to thank them as members of the SPIFF team for their contributions and the stimulating cooperation.

A final word of thanks goes out to Marjo de Kraker, who encouraged me to go on.

Maurice Dohmen
Amsterdam, December 1997

Contents

1 Introduction 1
- 1.1 Multiple-view feature modeling 1
- 1.2 Feature validation 3
- 1.3 Research objectives 4
- 1.4 Outline 4

Part I Background 7

2 Constraint satisfaction techniques 9
- 2.1 Constraint satisfaction basics 10
- 2.2 Finite domain CSPs 14
- 2.3 Infinite domain CSPs 18
- 2.4 Geometric domain CSPs 25
- 2.5 Discussion 31

3 Feature validation approaches 33
- 3.1 Constraint-based geometric modeling 34
- 3.2 Feature validation aspects 37
- 3.3 Approaches to feature validation 39
- 3.4 Discussion 44

Part II Feature validation 47

4 Multiple-view feature modeling 49
- 4.1 Multiple views 49
- 4.2 Product model 51
- 4.3 Implementation 55

5 Validity specification and maintenance 59
- 5.1 Constraint types 59
- 5.2 Feature and constraint specification 64
- 5.3 Link constraints 67
- 5.4 Constraint conflicts 70

Part III Constraint satisfaction 75

6 Constraint management 77
6.1 Overview 77
6.2 Constraint graph 79
6.3 Constraint maintenance 82
6.4 Optimizations 86
6.5 Discussion 91

7 Algebraic constraints 95
7.1 SkyBlue 95
7.2 Mapping 98
7.3 Implementation 100
7.4 Discussion 101

8 Geometric constraints 103
8.1 Degrees of freedom analysis 103
8.2 Mapping 112
8.3 Over- and underconstrained models 115
8.4 Implementation 122
8.5 Discussion 123

Part IV Results and conclusions 129

9 Results 131
9.1 SPIFF 131
9.2 Multiple-view modeling session 133

10 Conclusions and future work 139
10.1 Conclusions 139
10.2 Future work 141

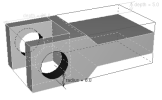
Bibliography 143

Index 151

Summary 155

Samenvatting 157

Curriculum Vitae 159



1

Introduction

In product development, geometric modeling has played a major role in the past. For example, in product design, it was used to generate images of the product, while in process planning, NC-code was generated. For many such applications, which correspond to some product life cycle phase, functional information is required. This information was provided by engineers, sometimes facilitated by automatic extraction from the geometric model.

The 1980s showed a paradigm shift from geometric modeling to feature modeling. A feature combines shape information with nonshape information, such as information about its function for the user in a design application, or its manufacturing process in a manufacturing application. Multiple interpretations of a product in terms of features are possible; each interpretation, or *view*, corresponds to an application.

An important issue in feature modeling is *validation*, which concerns the meaning of a feature as specified by its functional information. A feature modeling system should ensure that product modifications by a user are in accordance with the meaning of the features.

This chapter first gives a brief introduction of feature modeling concepts, in the context of multiple views. Next, it introduces basic feature validation concepts. Then, it gives the main objectives of the research work presented in the thesis. Finally, the chapter gives an outline of the thesis.

1.1 Multiple-view feature modeling

The concept of feature has been investigated by many researchers; Shah and Mäntylä [1995] give an overview of this research. We use the following definition: a *feature* is the representation of shape aspects of a physical product that are mappable to a generic shape and that have functional significance. From the definition, it follows that a feature corresponds to a shape. Figure 1.1 shows a product with some features. Each application has its own set of features, e.g.

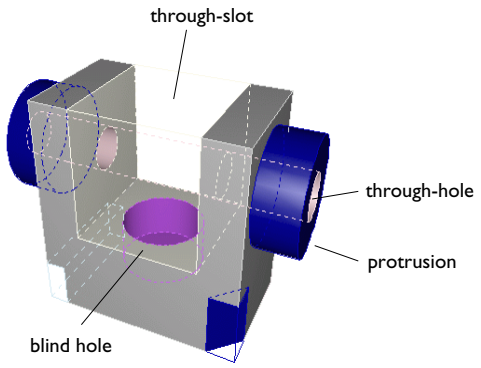


Figure 1.1: Example of a product with some features.

a manufacturing application has only features that correspond to manufacturing operations, such as milling and cutting.

The term feature refers to the *generic* object or *feature class* that encapsulates both the shape and nonshape information. This information is generic information applicable to a whole class of features. A feature is characterized by *parameters*, such as dimensions and position. A feature is *instantiated*, i.e. a *feature instance* of a generic feature is created, by specifying values for its parameters. A feature has a parametrized shape; a topologic entity of a feature, such as a face or the volume, is called a *feature element*.

In a *feature model*, feature instances including their relations are stored. A feature model can be obtained in two main ways. Firstly, a feature model can be obtained by extracting information from a previously created geometric model; this process is called *feature recognition*. Secondly, a feature model can be constructed by specifying feature instances; this is called *design by features*. The approaches may also be combined. Bronsvoort and Jansen [1993] give an overview of such approaches.

To reduce product development time and to improve product quality, the traditional sequential way of product development is more and more replaced by a concurrent approach. An example is *design for X*, where the X can stand for any product life cycle phase. Design for X means that already in the design phase, product requirements from other phases, e.g. manufacturability requirements, are taken into account; thus, design for X gives a tight coupling of the design phase and other phases, which improves the quality of the product. Another example is to execute product life cycle phases concurrently, to reduce the time-to-market.

In an ideal product development environment, multiple views of a product exist; each view represents an application or engineering discipline. A view is an interpretation of the product in terms of the features relevant for that application. To support concurrent engineering, it is required that product modifications made in any of the views are reflected in the other views, and that the models in all views remain valid. Thus, *simultaneous* maintenance of all views is required in a multiple-view feature modeling system.

1.2 Feature validation

A feature class has an intrinsic, functional, meaning; *feature validity conditions* express the rules that a feature instance should obey in order to behave in accordance with the meaning of its class. For example, a blind hole feature should always have a bottom in the model. This is part of the meaning of the blind hole; functions of the blind hole, e.g. in an assembly connection, possibly rely on the presence of this bottom. In traditional geometric modeling, such information was not captured; therefore, the model could easily be corrupted when modified. An ideal feature modeling system is able to detect any violation, and take appropriate action.

Feature modeling should provide a means to specify validity conditions for feature models, and to maintain the validity when it is edited. Specification and maintenance of feature validity conditions, or *feature validation*, is the main subject of this thesis. Constraints are used to specify such feature validity conditions, and constraint satisfaction techniques are applied to maintain feature validity; hence, we speak of *constraint-based feature validation*.

Early commercial feature modeling systems, e.g. Pro/ENGINEER™ [Parametric 1993], do not provide adequate functionality to specify and maintain feature validity conditions. Features are incorporated in those systems as a kind of macro, specifying a combination of geometric modeling operations, but validity conditions such as the presence of certain feature faces cannot be specified; these systems lack a persistent form of validity maintenance, in which feature validity is maintained under product modifications.

Constraint-based feature validation does provide persistent feature validity maintenance. Constraints specify required relations between objects; constraints for feature validation can be of different domains. For an example, see Figure 1.1. The through-slot feature class may have a minimum width constraint, because smaller slots cannot be manufactured, unless other equipment would be used. Different through-slot classes may be defined, corresponding to different types of equipment. Another example is a constraint between the through-hole instance and the protrusion instance specifying that the through-hole axis should

be centered with respect to the protrusion axis. Yet another constraint is used to specify the presence of the blind hole bottom. The first constraint is an algebraic constraint, the second is a geometric constraint, and the third one is a topologic constraint; a feature modeling system with proper feature validation thus has to be able to deal with different types of constraints from different domains. Furthermore, it should be able to perform feature validation in multiple views, which is an aspect of the simultaneous maintenance of the views.

1.3 Research objectives

The main objective of the research presented in this thesis is to provide a scheme for feature validation in a multiple-feature view environment The objective involves the following research topics:

feature validity specification What feature validity conditions are important, what constraint types are required, and how can they be specified?

feature validity maintenance How can the constraints of different types and from different domains be maintained and integrated, how do they interact, what constraint satisfaction techniques are required?

multiple-view maintenance How are changes in a view reflected in the other views, while ensuring the validity of all features?

The scheme should be verified by making an implementation of a multiple-view feature modeling system.

The research has been performed in close cooperation with Klaas Jan de Kraker, who focuses on feature conversion techniques to support concurrent engineering [De Kraker 1997]. Feature conversion involves propagation of changes made in one view to the other views. Where appropriate, a brief explanation of his work is given.

1.4 Outline

The thesis is divided into four parts:

Part I Background provides an overview of previous research done on constraints in Chapter 2, and on feature validation in Chapter 3.

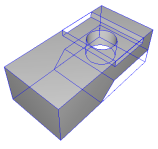
Part II Feature validation presents our approach to feature modeling in a multiple-view environment, in Chapter 4, and our approach to feature validation in such an environment, in Chapter 5.

Part III Constraint satisfaction presents the constraint management scheme to maintain the validity of all views, in Chapter 6, and focuses on maintenance of algebraic and geometric constraints, in Chapters 7 and 8.

Part IV Results and conclusions presents a demonstration session with the modeling system SPIFF that was implemented to test the multiple-view modeling approach, in Chapter 9; it ends with concluding remarks about achievements and limitations of the validation approach and about future research directions, in Chapter 10.

Parts of the research of this thesis have been published elsewhere [Dohmen et al. 1995, Dohmen et al. 1996, De Kraker et al. 1997*a*, De Kraker et al. 1997*b*, Noort et al. 1997]. Chapter 2 has been published in extended form as a survey article on constraint satisfaction [Dohmen 1995].

Part I Background



2

Constraint satisfaction techniques

CAD/CAM involves computer tools to model the characteristics and behavior of objects. With such tools, a geometric model is built by specifying relations between geometric entities. Such relations can specify, for example, coplanarity of faces, parallelism between edges, or coincidence of vertices. These relations are of a geometric nature. Other types of relations include algebraic relations, e.g. specifying that the height dimension of a cylinder should be twice its radius dimension, and topologic relations, such as one specifying that a face overlaps with another face. All such relations are to be maintained by the modeling system.

In general, relations within or between entities in a model that are maintained by a system can be represented by *constraints* on *constraint variables*. The set of specified constraints has to be kept consistent by *constraint satisfaction*, both when the value of a variable changes, e.g. during interaction with the model, and when the set itself changes, after addition or deletion of a constraint. The use of constraints in geometric modeling enables checking and maintenance of the validity of the geometric model. Furthermore, it enhances specification of the model, because a user merely has to specify the desired constraints declaratively.

There are various ways of expressing model relations in terms of constraints. For example, a topologic relation between faces may also be specified as a number of distance constraints between edges or vertices. When satisfying these constraints, their interdependency should be taken into account. For this, a formal representation known as constraint satisfaction problem, or CSP, has been described in the literature. In this formal representation, the domain of the constraint variables is important for the satisfaction techniques that can be applied.

Figure 2.1 on the following page depicts a classification of the CSPs discussed in this chapter. The classification is based on the type of domain of the variables. An important distinction is made between CSPs with variables having finite domains and with variables having infinite domains. Infinite domain CSPs are divided into CSPs that contain algebraic constraints and those that contain geometric

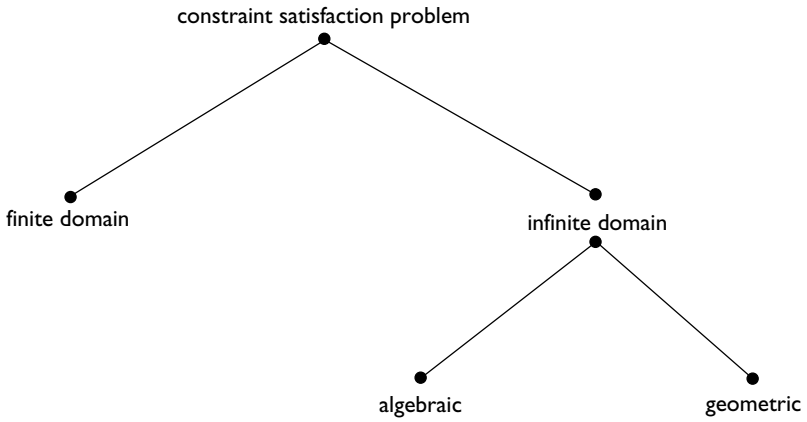


Figure 2.1: A classification of constraint satisfaction problems, based on domain type of the variables.

constraints. For algebraic CSPs, general-purpose solving techniques for infinite domain CSPs are discussed, whereas for geometric CSPs focus will be on specialized geometric techniques. Many of these specialized techniques use knowledge of the geometric domain to perform constraint satisfaction more effectively, for problems that general satisfaction techniques cannot handle, or more efficiently, by faster satisfaction.

The remainder of this chapter is organized as follows. Section 2.1 presents basic concepts that are important to constraint satisfaction. Sections 2.2 and 2.3 give an overview of general constraint satisfaction techniques for finite and infinite domain constraint satisfaction problems, respectively. Section 2.4 gives an overview of geometric constraint satisfaction techniques. Finally, in Section 2.5, some conclusions are drawn about the use of constraint satisfaction techniques in geometric modeling.

2.1 Constraint satisfaction basics

Many engineering problems involving constraints can be formally represented as constraint satisfaction problems. This section gives an overview of basic definitions and representations for constraints, together with their formalization as CSPs. Also, the propagation of information between constraints is discussed.

Constraints

A *constraint* is defined as a specification of a relation that should hold, defined on one or more entities or *constraint variables*. A constraint may be *parameterized* with one or more parameters, e.g. a distance value for a distance constraint. The number of variables a constraint has, determines its *arity*. A constraint can be *unary* (one variable), *binary* (two variables), or, in general, *n-ary*. Furthermore, the *direction* in which a constraint operates is important. A constraint with two or more variables is *unidirectional* if it specifies a hierarchical (asymmetrical) relation: only one distinguished variable is determined by the constraint, using the other variable(s) as reference(s). Other constraints are *bidirectional* constraints (for nonhierarchical, symmetrical relations), and, in general, *multidirectional* constraints.

Constraint satisfaction problems

A *constraint satisfaction problem* [Tsang 1993] has three ingredients: variables, domains, and constraints. The constraints are defined on one or more variables, each variable with its own domain. The set of variables and the set of constraints are assumed to be finite. A *label* is a (variable, value)-pair representing an assignment of a value to a variable. An *n-compound label* represents the simultaneous assignment of *n* values to *n* variables. A compound label is said to *satisfy* a constraint, if simultaneous assignment of the values to the variables of the constraint according to the compound label, is included in the relation that the constraint specifies.

Finding solutions to or *satisfying* a CSP (also called *solving* a CSP) boils down to determining compound labels or *solution tuples* that satisfy all the constraints. CSPs can be distinguished on the basis of whether *any* solution tuple should be found, *all* solution tuples, or an *optimal* solution with respect to some optimization criterion. Many problems contain only unary or binary constraints. These are called *binary* CSPs, and a great deal of research has focused on this type of CSP. The CSP definition given above refers to *general* CSPs. CSPs can be classified according to the cardinality of the domains of the variables. If the domains are finite, we call them *finite* CSPs. These are dealt with in Section 2.2. *Infinite* CSPs are discussed in Section 2.3.

A CSP can be *underconstrained*, in which case it has more than one solution, or even infinitely many solutions if the variable domains are infinite. If a CSP is *overconstrained*, in general it has no solutions, but it may have one solution if there are no conflicting constraints. In case the CSP does not have a solution, analysis may identify one or more constraints to be removed. The analysis of

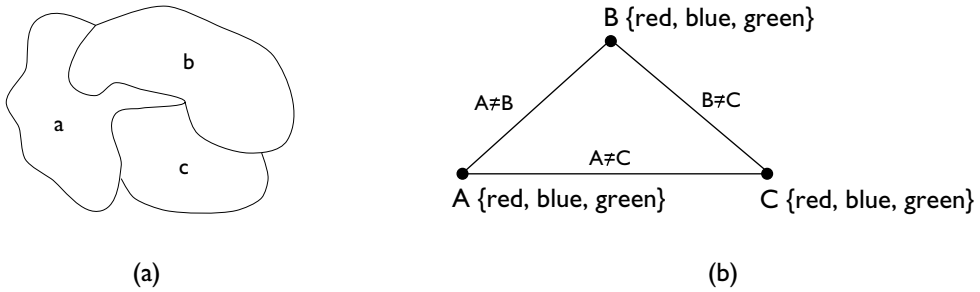


Figure 2.2: A map-coloring problem (a), and the corresponding CSP, represented as a graph (b).

CSPs with respect to overconstrainedness is difficult but necessary, particularly in geometric modeling, where often single solutions are searched for.

Representation of CSPs

Constraint satisfaction problems can be represented in many ways. A very common way is the representation of a binary CSP as a *graph*, in which the *nodes* represent variables and the *arcs* represent constraints. If the graph is undirected, i.e., if all constraints are nonhierarchical, the arcs are called *edges*. Sharing a variable between constraints is expressed by the incidence relation in the constraint graph: two constraints share a variable if their corresponding edges have the node representing this variable in common. For example, Figure 2.2(a) depicts a map with three countries that should be assigned colors (say, red, blue or green) different from those of their neighbors. Such an example of a *map-coloring problem* can be formulated as a CSP, in this case one with three variables. In Figure 2.2(b), the corresponding CSP is represented as a graph with three nodes and three edges. The variable domains are shown in braces, next to the corresponding node. The constraints are shown next to the corresponding edge.

This representation is not suitable for general CSPs, because these may contain constraints between more than two variables. In a *hyper-graph*, *hyper-edges* between two or more nodes represent constraints on two or more variables. Some formalizations generalize these concepts by considering constraints as *variables*, thus allowing constraints on constraints. Such *higher-order* graphs [Tyugu and Uustalu 1993] cannot be expressed as hyper-graphs, since a constraint can at the same time be used both as a variable (participating in a higher-order constraint) and as a constraint. A CSP can also be represented by a number of *logic formulae*, such as first-order logic predicates, with the advantage that inex-

act constraints such as inequalities can be expressed. Variables may have infinite domains.

Extending the CSP definition

For problems with many solutions, it may be necessary to determine the *best* solution, according to some criterion. The above definition of a CSP is not suitable for this, and therefore needs to be extended. Another type of problem not captured above is found in situations where no solution exists, but where we are interested in quasi-solutions that satisfy the constraints to a certain degree. The first type of extended CSP is called *constraint satisfaction optimization problem* (CSOP). The task is to determine optimal solutions with respect to an optimization function defined over the set of solution tuples. Usually all solutions must be found and compared. Typical examples of CSOPs are found in kinematic simulations, where energy functions are minimized. The second type is referred to as *partial constraint satisfaction problem* (PCSP), where the task is to determine optimal compound labels with respect to an optimization function defined over the set of compound labels. Typical examples are scheduling problems. A PCSP is in fact a generalization of a CSOP. Usually PCSPs are overconstrained problems. A typical example of a PCSP is a *minimal violation* problem, where the task is to label all variables, while minimizing a global cost function associated with the compound labels. Another example is a *maximal utility* problem, where the task is to label variables so that as many of the constraints are satisfied as possible, by maximizing a global utility function defined over the compound labels.

Constraint propagation

In order to find solutions to a CSP, information has to be communicated between variables. With regard to a particular variable, *constraint propagation* is the process of removing incompatible values or selecting allowed values from other variables that are connected to the considered variable by constraints. It is a local process, but if applied recursively, it becomes global propagation.

Some categories of constraint propagation, classified on the type of the inferred information are [Davis 1987]:

value inference also called *propagation of known states*. Here exact values are propagated along constraint arcs to connected variables. It is applied in numerical constraint propagation systems, e.g. Thinglab [Borning 1981]. It is particularly useful when the constraints are *functional* constraints, meaning that *input* variables of a constraint uniquely determine the *output* variables [Tyugu and Uustalu 1993, Sannella 1993]. In this case the variables can be labeled in a single pass.

interval inference Instead of exact values, intervals representing ranges of values are propagated, using interval arithmetic. It is useful in temporal reasoning, where constraints such as **A overlaps B** are defined on time intervals, and also in spatial reasoning, which can be considered as a generalization of temporal reasoning with two- or higher-dimensional intervals. In [Hyvönen 1992] interval inference is explored.

label inference Here the domain of the variable is restricted by considering the constraints attached to the variable. It is applied, for example, in consistency achieving algorithms used for problem reduction.

constraint inference In this type of propagation, constraints are represented as sets of allowed compound labels. New constraints on a subset of the variables are inferred from existing constraints, and added to the constraint graph. Such a new constraint restricts its variables, by removing compound labels.

The propagation of information between constraint variables is influenced by the dependencies of the constraint variables. If a variable is dependent on itself through one or more constraints, this is called a constraint *loop* or *cycle*. These terms refer directly to the representation of the CSP as a graph. Constraint satisfaction techniques often apply special techniques to deal with constraint loops.

2.2 Finite domain CSPs

Much research has been done on finite domain CSPs, in which the domains of the variables are finite sets. Examples are found in map-coloring problems and product material selection. The finiteness of the domains allows the examination of values for a variable one by one. If there is no value in the domain that satisfies all constraints on that variable, the CSP is unsatisfiable. The relations imposed by the constraints are typically expressed *extensionally*, i.e. as a set of allowed labels, see, for example, Figure 2.3.

Figure 2.4 depicts the classes of satisfaction techniques for finite CSPs that are discussed in this section. They include techniques for *problem reduction*, *search* techniques, and *solution synthesis* techniques. Problem reduction techniques try to transform the problem into equivalent problems that are hopefully easier to solve, e.g. by removing redundant values. Search techniques try to find solutions by examining several values for the variables. In solution synthesis, solutions are constructed by extending partial solutions (compound labels) with additional labels. Before discussing these techniques in more detail, we will discuss the notion of consistency.

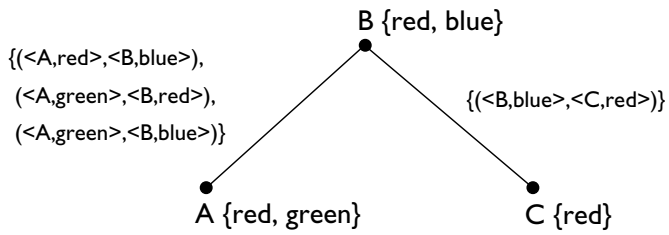


Figure 2.3: A CSP with extensionally specified constraints.

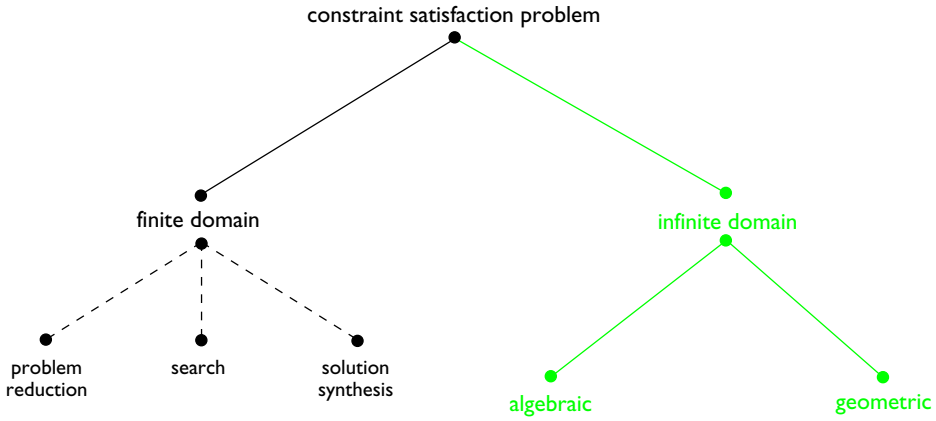


Figure 2.4: Solving techniques for finite domain CSPs.

Consistency

Many CSPs that have solutions nevertheless contain values in the domains of some variables that are not consistent with the constraints on those variables. Although such a CSP is consistent with respect to all variables and constraints, it is locally inconsistent.

In the literature, two important concepts are distinguished: *satisfiability* and *k-consistency*. A CSP is satisfiable, if all constraints concerning some set of variables can be satisfied. A CSP is *k*-consistent, if it is possible to extend any compound label satisfying all constraints on a particular subset of *k*-1 variables, with some label for an additional variable, such that all constraints on the set of *k* variables are satisfied. This is a *local* form of consistency. In general, a CSP is said to be *consistent* with respect to a certain property, if all compound labels violating this property have been removed. For binary finite CSPs, 1-, 2- and 3-consistency are called *node*-, *arc*- and *path*-consistency respectively.

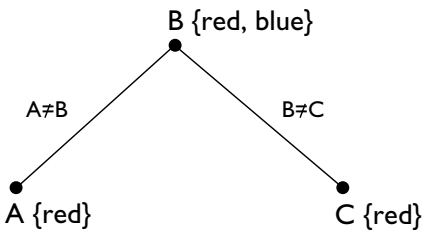


Figure 2.5: A CSP, which is 3-consistent, but not 2-consistent.

Consider, for example, the CSP depicted in Figure 2.5 (after [Freuder 1982]). If we consider sets of two variables, we see that the allowed 2-compound labels are **(**<**A,red**>**,** <**B,blue**>**)**, **(**<**A,red**>**,** <**C,red**>**)**, and **(**<**B,blue**>**,** <**C,red**>**)**. Each of these compound labels can be extended to the compound label **(**<**A,red**>**,** <**B,blue**>**,** <**C,red**>**)** satisfying the two constraints for the set of three variables (it is a solution). If we consider sets of one variable, one allowed label for **B**, <**B,red**>, cannot be extended to satisfy the constraints **A**$\neq$**B** or **B**$\neq$**C**. Thus, the CSP is not 2-consistent. However, it is 3-consistent, and has a solution. Therefore, we see that a constraint satisfaction problem can be locally inconsistent and yet have valid solutions.

Problem reduction techniques

Problem reduction techniques try to reduce redundancy in the CSP to obtain an equivalent problem that may be easier to solve. This may be done by removing redundant values from the domains, or redundant compound labels from the constraints. Such techniques are often used as a preprocessing step for solution search techniques. If we consider, for example, a path **(x, y, z)** consisting of two arcs in a binary CSP and want to achieve path-consistency, we may remove all redundant values for **y**, in this case, those values that do not occur as part of a compound label in the constraints **(x, y)** or **(y, z)**. For example, in the CSP depicted in Figure 2.5, the value **red** for **B** is redundant and may thus be removed, because of the constraint **B**$\neq$**C**. This is an example of label inference (cf. Section 2.1).

Search techniques

The general idea of search techniques is to determine solutions to a CSP by searching the space of possible labelings. When, during the search, a dead-end is reached, the algorithm will backtrack, i.e. go back to a previous decision point

and choose another labeling. These techniques are particularly useful in loosely constrained situations, since it can be expected that there will not be many backtracks. The techniques can be used to search a single solution, or all solutions.

The basic search strategy is doing an exhaustive search by examining all values for a variable one by one, without making use of constraint propagation. A well-known strategy is chronological backtracking, where variables are labeled one value at a time. The label is checked against all previously assigned labels; if it is incompatible it is rejected, and another label is tried. When all labels for a variable have been examined and none is acceptable, the previously labeled variable is assigned a new label (backtracking), and the search continues.

In some search strategies, information is inferred about failure reasons, to avoid repeatedly searching useless paths when backtracking occurs. Keeping information about reasons for failure of a certain choice is difficult, since many inferences may be required in case of combined failure reasons, and since values may not only be rejected because of certain constraints but also due to problem reduction. To cope with this, Assumption-based Truth Maintenance Systems [De Kleer 1986] attach justifications or assumptions to inferences made. By keeping this information, it is possible to reason with default values, to avoid the re-execution of rules, and to make better choices when backtracking.

Stochastic search techniques search the whole search space in a nondeterministic manner, guided by heuristics. They are in general incomplete (they may not find all solutions), but may be faster than deterministic search strategies discussed above. One example of a stochastic search technique is hill-climbing, where the search space is traveled looking for maximum values of some evaluation function. Heuristics for traveling, i.e. choosing a new variable/value pair, include choosing the pair that gives the largest increase of the evaluation function value, and choosing the pair that violates as few constraints as possible.

Another stochastic search technique is simulated annealing, in which the search space is traveled in a manner similar to hill-climbing, but without the problem of staying in local maxima. Because also small decreases in the value of the evaluation function are accepted, simulated annealing is able to find absolute maxima. Simulated annealing is not limited to finite CSPs, but also works for infinite domains.

Solution synthesis

In solution synthesis the goal is to find all solutions to a problem by constructing allowed compound labels instead of by removing redundant labels. One approach is to consider all constraints on some subset of the variables as one implicit constraint on all these variables together. A new graph is built that contains these constraints, and which is solved using constraint inference.

Comparison

Problem reduction techniques and search techniques are complementary. Problem reduction boils down to deterministically deleting values, whereas general search techniques select values nondeterministically. Between the two extremes lie hybrid algorithms. Preprocessing of the CSP or bookkeeping of previous choices is done in order to remove nondeterminism. Experimental results indicate that optimal algorithms are likely to be found in these hybrid algorithms.

In general, search algorithms look for single solutions to a CSP. A CSP may be loosely constrained and thus have many solutions, or it may be tightly constrained. Tsang [1993] argues that, if a single solution is required, a loosely constrained problem can be solved efficiently (without many backtracks) using search techniques. If, however, the problem is tightly constrained, the number of backtracks may be much higher. In this case, problem reduction techniques help to prune the search space.

Solution synthesis algorithms try to construct all solutions. They are best suited for tightly constrained problems, when the number of elements represented in the nodes constructed will be relatively small. In this case also, problem reduction techniques can be applied. If the problem is loosely constrained and all solutions are searched for, the problem is hard by nature. Problem reduction and solution synthesis then do not perform much better than general search techniques.

Finite domain techniques are of little use when the domain sizes become large or even infinite, or when the number of variables can change due to a labeling action. In highly underconstrained problems, the number of solutions can be large. For such problems other techniques are needed, some of which will be presented in the following section.

2.3 Infinite domain CSPs

When the domains of the variables are allowed to be infinite, constraints are usually expressed intensionally, for example, as predicates containing inequalities. Infinite CSP satisfaction techniques can be categorized as syntactical or semantical.

Syntactical techniques try to derive general solutions to a problem, by deriving new formulae from a given set of formulae, e.g. by applying symbolic-algebra techniques. These general solutions can then be evaluated with different value assignments for variables. The variables and formulae are first only manipulated symbolically, without considering their values. Examples are resolution theorem-proving, and the Gröbner basis approach. Although syntactical satisfaction sys-

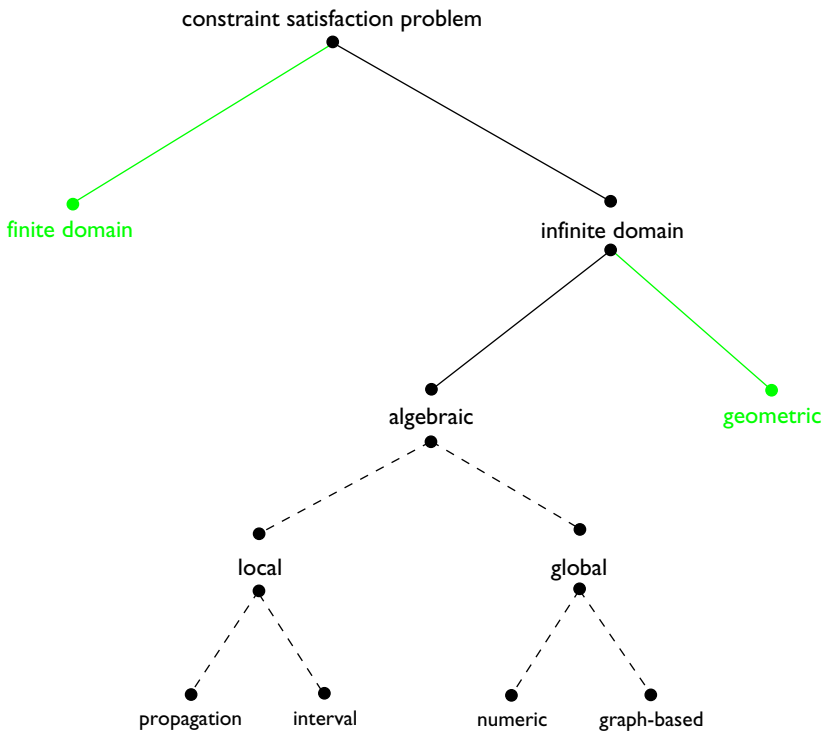


Figure 2.6: Solving techniques for infinite domain CSPs.

tems are general-purpose problem solvers, they have limited use for constraint satisfaction because of their low execution speed. Therefore, they will not be considered any further in this section.

Semantical techniques, on the other hand, determine solutions directly, making explicit use of the *values* of the variables, as in arithmetic. In logic programming, for instance, use is made of semantical techniques such as unification, and in numeric-equation solving, use is made of iterative semantical techniques such as relaxation. Many semantical constraint satisfaction techniques have been investigated, and this section presents a number of important ones. We only consider here the infinite CSPs that contain algebraic constraints on variables that have numerical values. Infinite CSPs with geometric constraints are elaborated in the next section.

Figure 2.6 depicts the classes of satisfaction techniques for infinite CSPs with algebraic constraints. Solving techniques are characterized by the extent to which they consider only local information in the constraint graph, or also global information.

Local propagation techniques

Local techniques include propagation of known states, in which values are propagated along constraint arcs in the constraint graph (cf. Section 2.1.) Another one is propagation of degrees of freedom, in which parts of the constraint graph that contain variables with enough degrees of freedom are removed temporarily; the fewer constraints a variable has, the more degrees of freedom it has. The remaining graph is solved, and then the values for the removed variables can be determined by propagation of the values of the labeled variables. In fact, this technique tries to translate parts of the constraint graph into functional constraints, which have input variables that uniquely determine the output variables. For a functional constraint, an assignment can be compiled to obtain the values of the output variables, which speeds up the propagation process. None of these local propagation techniques can handle loops in the constraint graph.

In computer graphics, local propagation techniques were used from the early days on. Sutherland [1963] developed the system Sketchpad. With this system, one could interactively make drawings using a light pen. First, a rough drawing with lines and circles was constructed, specifying constraints like point-on-circle. After this, the drawing could be beautified by calling a constraint solver. This solver used propagation of degrees of freedom to satisfy the constraints, and relaxation (cf. Section 2.3) to deal with loops in the constraint graph.

Building on Sutherland's work, Borning [1981] developed the Thinglab system. It was an extension of the object-oriented programming language Smalltalk, with special object definitions for arithmetical constraints. These constraints consisted of a predicate to be satisfied, and a number of procedural methods for doing so. The selection of a method depended on which variables were known and which were unknown. The selected methods were compiled into a plan, which was executed during the constraint satisfaction process to speed up computation when only the values of the input variables changed, and the values of the other variables could be determined in one pass. The satisfaction techniques used were propagation of known states, propagation of degrees of freedom, and relaxation. In later versions of Thinglab, graphical constraint specification techniques [Borning 1986], and constraint hierarchies [Borning et al. 1987] were added. Borning's work was continued in the development of the SkyBlue solver [Sannella 1992]. This approach is used in this work, and is elaborated in Section 7.1.

Figure 2.7(a) on the next page gives an example of local propagation approaches. The graph represents the expression **2A + 5 = B** as two constraints. If the value for **B** is known, say **11**, values are propagated along the arcs as shown in the figure, and thus the value for **A** is determined by local propagation. To avoid determining a satisfaction method for each of the constraints every time,

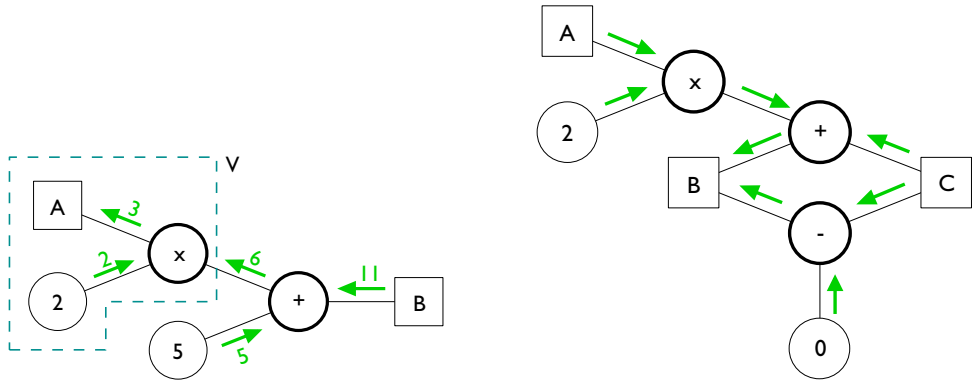


Figure 2.7: Examples of a graph that is solved with local propagation techniques (a), and one that cannot be solved with local techniques (b).

as done in local propagation, propagation of degrees of freedom can be used. **A** has only one constraint, and we may replace **A** and its constraint by a single variable **V**, indicated by the dashed box. This replacement is compiled as an assignment **A := V/2**. A similar step is then done for **B** and its constraint, leading to an assignment **A := (B-5)/2**, which gives the value of **A** as a function of **B**.

Figure 2.7(b) depicts a graph containing a loop. The expressions are **2A + B = C** and **B = C**. If **A** and **C** are both known, then **B** is determined by two paths in the graph, as depicted. Such a situation cannot be handled with local propagation techniques.

Interval techniques

Local propagation techniques determine single, exact values. If we allow inexact values, or even sets of values, we need a way to represent such values. Hyvönen [1992] describes a generalization of CSPs called tolerance CSPs, in which *tolerances* are sets of values for a variable. Tolerance CSPs generalize both finite and infinite CSPs. Tolerances may thus be discrete or continuous. For underconstrained situations tolerances can represent multiple solutions. The domain of a variable is a tolerance of maximum size. Exact value solutions can be represented as singleton tolerances containing a single value. Finally, propagation of known states is replaced with tolerance propagation, which in the case of infinite domains is interval inference (cf. Section 2.1). Borning et al. [1996] describe a

similar approach to local propagation of intervals, which also handles constraint priorities.

To handle constraint graph loops, local techniques do not suffice. Techniques are needed that can be applied to larger parts of the constraint graph. There are two classes of such techniques: iterative numeric techniques, and graph-based techniques.

Numeric techniques

Numeric techniques iteratively find approximations to solutions for a set of constraints, starting with initial values, until some convergence criterion is met. These constraints are usually expressed as simultaneous equations. Examples of such techniques are:

relaxation Here initial guesses for the values of the variables to be relaxed are made, and then the error of assigning these values is estimated by error functions that are associated with each constraint. New guesses are made on the basis of these errors and some heuristics, until the errors are within some bounds. This naive relaxation [Leler 1988] can be improved, e.g. by perturbing the value of each variable in turn, until a (large) decrease of an error estimate is found. Another technique is to relax only a subset of the variables in the graph instead of all variables, and use propagation of known states to determine the values of the other variables.

Newton-Raphson iteration Here the Jacobian matrix is utilized to compute new values for the variables. The Jacobian contains the partial derivatives of the equation functions with respect to the variables in the equations.

Simplex technique This technique can solve systems of linear inequalities and equalities with an optimization function. Inequalities can be interpreted as half-spaces of possible values, and equalities as hyper-planes. In order to solve a system, the intersection of these spaces is determined.

Newton-Raphson iteration is better suited for solving nonlinear equations than relaxation, because relaxation will converge slowly due to the fact that only the values for the variables are used. On the other hand, Newton-Raphson cannot solve under- or overconstrained systems of equations, since in those cases the Jacobian matrix is not a square matrix and therefore cannot be inverted. Techniques have been developed for these cases that change the system of equations [Light and Gossard 1982, Rocheleau and Lee 1987].

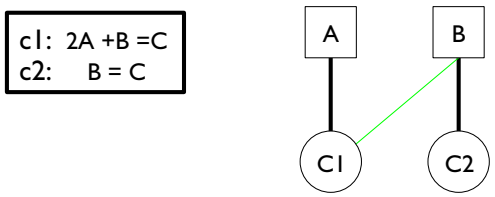


Figure 2.8: Maximum matching for the graph of Figure 2.7(b).

Many graphics systems employ numeric techniques. Gleicher [1992] describes the drawing system Briar, in which direct manipulations result in forces acting on the manipulated object. A force is assumed to be proportional to the associated velocity, not to the acceleration. Constraints are satisfied by forces that are applied to (re-)establish a relation. This requires a linear system of equations, solved differentially [Gleicher and Witkin 1991] during direct manipulation, using iterative expression evaluation in a constraint graph.

Van Overveld [1993] developed a relaxation technique that solves a set of geometric constraints by simultaneous constraint satisfaction in an uncoupled manner. Constraints are defined on 3D points. For every constraint type, a satisfaction algorithm needs to be specified that computes correction vectors for the points involved. A constraint set is solved, by first computing the correction vectors for the points according to the constraints, without taking into account the coupling between constraints that share one or more variables. Next, these correction vectors are multiplied with a factor smaller than one, and applied one after another to the points. These two steps are repeated until some convergence criterion is met. The factor controls the convergence process.

Graph-based techniques

Graph-based techniques exploit the structure in a constraint set by representing the constraints in a constraint graph. They consider not only the locally available constraint information, but also substructures in the constraint graph. An example of a graph-based technique is graph matching. This technique identifies the unknown variables for every constraint, and builds a directed graph of dependency relations between the variables, expressing which variables depend on which other variables. This directed graph, which represents the solution plan for the constraint set, may contain cycles, which are solved numerically. The matching algorithm detects globally and locally under- and overconstrained sets. Graph matching is employed, for example, in the CAD system of Serrano and Gossard [1988] to solve algebraic constraints on dimensions.

For an example, consider again Figure 2.7(b) on page 21. Let us assume that we want to compute the values of **A** and **B**, given the value of **C**. Figure 2.8 on the page before shows a *maximum matching* for this graph, indicated by bold lines. A maximum matching contains a subset of all edges in this graph, such that each variable is connected to only one constraint. A bold line from a variable indicates that it is an output for the corresponding constraint. Using only those edges in the original constraint graph that are part of the maximum matching, the original constraint graph with a loop has been converted into a directed graph. In this way, the values of **A** and **B** are determined.

Comparison

Both propagation of known states and propagation of degrees of freedom are local techniques. They provide fast and efficient satisfaction, particularly if both techniques are combined and propagation is compiled. Because these techniques are not capable of solving cycles in the constraint graph, they have to be combined with global satisfaction techniques, e.g. numeric techniques.

An advantage of numeric techniques is that they can be used as optimization techniques, since in systems without a solution, they will find optimal compound labels for the variables according to certain criteria. See Section 2.1 where Partial CSPs were discussed. Numeric techniques have the disadvantage that convergence is not guaranteed, since it may depend on the initial values. Furthermore they are slow, and in underconstrained situations they will find at most one solution, possibly arbitrary or possibly depending on the initial values. The latter can destroy the declarative properties of the satisfaction techniques by giving solutions that depend on the order of satisfying the constraints. Note also that numeric techniques only work for continuous variable domains. Finally, compilation is not possible.

Graph-based techniques use global information about the structure of the constraint graph to make them more efficient. However, they are not as fast as local techniques. Recently, Lamounier [1996] described a hybrid approach to solve both geometric constraints on degrees of freedom and algebraic constraints on dimensions. Both constraint types are represented uniformly by equations which are solved using a combination of local propagation and graph matching to handle loops.

All these techniques determine single values for the constraint variables. The generalization of CSPs into tolerance CSPs allows sets of values to be determined. However, in practice they are difficult to implement.

The techniques for satisfying CSPs with infinite domains, presented in this section, are general purpose. They are, in general, too weak to handle specific types of problems that require knowledge about the problem domain. One such

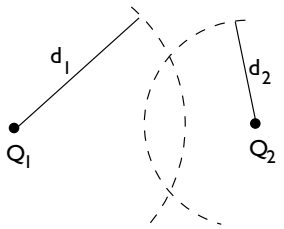


Figure 2.9: An example of a problem involving geometric constraints.

domain is the geometric domain. For example, consider a simple example in which a point is subject to two distance constraints, see Figure 2.9. A point has to have a fixed distance $\mathbf{d_1}$ to point $\mathbf{Q_1}$, and a fixed distance $\mathbf{d_2}$ to point $\mathbf{Q_2}$. Propagation of known states cannot be applied because there is no unique compound label for any constraint. If this problem is solved numerically, coupled nonlinear equations have to be solved. In the geometric domain, however, a solution may be constructed by intersecting two circles, finding two solutions. The next section deals with constraint satisfaction techniques for constraints on geometry. Some of these techniques are variants of satisfaction techniques for infinite CSPs, adapted for use in the geometric domain.

2.4 Geometric domain CSPs

This section gives an overview of constraint satisfaction techniques for geometric constraints. All these techniques exploit knowledge of the geometric domain in the satisfaction process. Figure 2.10 on the next page depicts the different categories of geometric satisfaction techniques. We distinguish a number of classes of such techniques, ranging from local geometric propagation techniques, to global geometric reasoning techniques.

Local geometric techniques

Local geometric techniques only use locally available geometric information, and propagate this information throughout the graph. They are similar to the propagation techniques discussed in Section 2.3, but the propagated information is different. Local geometric techniques do not propagate simple numeric values, but instead they propagate values for geometric parameters such as positions and orientations. These values are represented either *explicitly* as 2D or 3D geometric primitives, or *implicitly* by geometric operations.

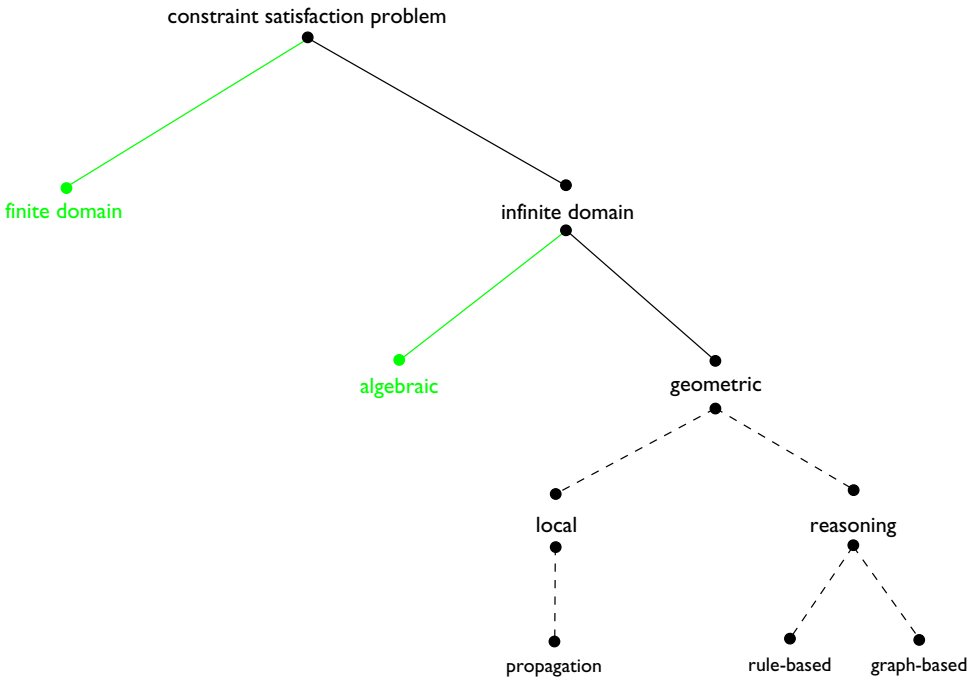


Figure 2.10: Solving techniques for geometric CSPs.

An example of an explicit local approach is the *quantum approach* [Veltkamp and Arbab 1992]. A quantum represents a partial solution set for a variable, resulting from one or more geometric constraints. Quanta are Boolean combinations of geometric primitives representing loci of points. Quanta can be considered a special case of the tolerances discussed in Section 2.3. When a constraint is added to the existing constraint set, quanta are computed for the variables of this constraint and intersected with quanta resulting from previous constraints on the same variables. This results in new quanta, which are propagated through the constraint graph. In this way, the algorithm achieves arc-consistency, cf. Section 2.2. A strong point of this approach is that in underconstrained situations all solutions are determined. The approach has been implemented for 2D constraints; for 3D constraints the quanta may become 3D also, which increases the complexity of the intersection computations enormously, thus limiting the usefulness of the quantum approach.

Another class of local geometric techniques propagates geometric *operations* or *transformations*. In this *operational* approach to geometric constraint satisfaction, geometric constraints are viewed as specifications of geometric operations. Constraints are satisfied by carrying out the associated operations. They are satisfied

incrementally, i.e. in a sequential manner, independent of other constraints, making use of the results of previous computations.

Rossignac [1986] describes an operational approach to constraint satisfaction, in which geometric constraints on CSG primitives are satisfied by one or more rigid motions. The constraints are satisfied incrementally in the order of specification. Rigid motions are defined procedurally, and initialized to identity. Because of these default values, which may be overruled in the constraint satisfaction process, underconstrained situations cannot occur.

Arbab and Wang [1991] describe an incremental approach that they call operational transformation planning. This planning approach determines single solutions. Operational transformations are used to satisfy constraints on geometric objects like circles and lines. Every transformation reduces the number of degrees of freedom of objects. These degrees of freedom are expressed implicitly as a set of constraint preserving operations. When a constraint needs to be satisfied, the associated operations are propagated through the constraint graph. In the propagation process, the set of allowed constraint preserving operations for a variable may be reduced. Loops detected in the propagation process may be broken by performing a meta-operation on the graph, such as deletion of a redundant constraint, or rewriting the constraints in the loop as a system of equations to be solved by relaxation. The satisfaction operations are put into a plan, to abstract from the dimensional properties of the objects. After the planning stage, the plan is executed, using the values of the dimensions of the objects.

Van Emmerik [1989] developed the modeling system GeoNode, which provides hierarchical structuring of geometric models, and direct manipulation on parameterized 3D objects. A geometric constraint fixes one or two degrees of freedom of two coordinate systems with respect to each other. Operational constraint satisfaction is done incrementally, in the predefined order of the geometric hierarchy [Van Emmerik 1991]. The GeoNode constraint solver determines a single solution. In case of multiple ways to satisfy a particular constraint, the solution is chosen that minimally disturbs the model configuration. Constraint loops are prohibited. Extensions to GeoNode have been made, to improve the specification of geometric models by means of higher-level constraints [De Kraker et al. 1994]. Such constraints, for example, a volume constraint, are specified on CSG primitives or faces of CSG primitives, instead of on coordinate systems. They are compiled to a sequence of geometric constraints on coordinate systems, which is then solved.

Fa et al. [1993] developed a constraint-based modeling system that determines the degrees of freedom of a constrained object operationally in terms of allowed rigid motions of the object. For each new constraint on the object, the

allowed motions are reduced using table look-up. There is no report of how constraint loops are handled.

Local geometric techniques are efficient, because they only use local information, and are usually incremental: to satisfy a constraint, they do not require to solve the complete constraint graph. However, due to their locality, the techniques are not powerful, particularly if the constraint graph contains loops. This requires techniques that apply a more *global* approach. Such techniques can be considered to *reason* about geometric aspects of a model.

Geometric reasoning

In geometric reasoning, symbolic solutions are determined, irrespective of the actual values of the constrained object properties. These symbolic solutions are plans that can be executed with the actual values, giving actual solutions. Two main classes are distinguished. Rule-based approaches use a general-purpose resolution system, which is fed with geometric knowledge; graph-based approaches apply geometric knowledge to solve the geometric constraint graph by decomposing it into smaller components.

Rule-based approaches

Brüderlin [1986] has developed a constraint satisfaction system that is based on geometric construction rules expressed as rewrite rules. The system has been implemented in Prolog. The constraint variables (in this case, characteristic points) are projected onto a 2D plane, in which the constraint set is solved. First a symbolic solution plan is determined, which can later be executed using real values of objects. His system has been extended to support implicit constraint specification, i.e. automatic definition of constraints after certain direct manipulation actions [Sohrt and Brüderlin 1991].

Suzuki et al. [1990] use First Order Predicate Logic to represent both geometric models and associated geometric constraints, and to perform geometric reasoning. Topologic constraints are used to define and maintain the topology of the geometric model. The dimensional properties are expressed as geometric constraints on geometric elements. Their reasoning system consists of an inference engine and a logical database. The inference engine executes IF-THEN production rules to propagate dimensional constraints, employing forward reasoning. The logical database is an Assumption-Based Truth Maintenance System (cf. Section 2.2). It maintains facts about the geometric model and the constraints, and also information about the reasons for certain inferences made. Suzuki et al. use this reasoning system in a parametric modeling system. It can keep track of

the dimensional dependencies between variables. In overconstrained situations it shows the conflicting constraints. In underconstrained situations specific default rules can be executed. These rules, which were not specified explicitly by the user, assume new facts if these are not in contradiction with the existing facts. In this way, default rules can be used to infer default properties of a geometric model.

Graph-based approaches

Graph-based approaches exploit the structure of the constraint graph to make the solving process more efficient. Such techniques identify subgraphs that are solved first, using geometric knowledge. Then, these solved subgraphs are combined to obtain the final solution. The decomposition and combination steps provide a symbolic plan to construct the model specified by the constraints. To solve the actual model geometry, the values of the model dimensions are used.

Owen [1991] describes a technique that recursively decomposes the constraint graph into biconnected components. The configuration of the nodes that connect the components are determined using algebraic equation-solving techniques. If a component cannot be split any further, the nodes in the component themselves are solved. Then, the solved components are combined to obtain the final solution. The equations that need to be solved are equivalent to standard geometric construction methods that use ruler and compass. Owen's technique can handle under- as well as overconstrained models. The algorithm described is 2D, a 3D version of the constraint solver is commercially available.

Bouma et al. [1995] describe a 2D geometric solver that, in contrast to Owen's, does not work top down but bottom up. By satisfying a constraint between two geometric elements, they combine geometric elements by construction steps into clusters that are rigid, i.e. have no internal degrees of freedom. A cluster is extended as much as possible, and then placed relative to one or two other clusters according to the constraints between elements of two different clusters. The construction steps are explicitly coded, and depend on the combinations of constraints that may occur. It is unclear whether this approach can be extended to 3D geometric constraints, particularly because the construction steps are much more complex for 3D constraints. Recently, an extension was published that can handle underconstrained models by combining the approach with a top-down decomposition scheme [Fudos and Hoffmann 1997].

Kramer [1992*b*] developed a 3D constraint solving approach that reasons about the degrees of freedom of objects. The approach is quite similar to that of Bouma et al. [1995], as it also combines elements into rigid structures in a bottom-up fashion. It differs in the way the constraints are satisfied. Here, specialized procedures are used to satisfy a constraint on an object such that previ-

ous constraints on this object are not violated. For this, the degrees of freedom of the object are considered, instead of explicitly enumerating all possible combinations of constraints. This approach is used in this work, and in Section 8.1 it is described in much more detail.

Hsu and Brüderlin [1997] describe a geometric constraint solver that works in two phases. In the first phase, the constraint graph is transformed into a directed dependency graph by analyzing the number of degrees of freedom that are fixed by constraints. This directed graph forms a symbolic solution of the constraint graph. For underconstrained models, the remaining degrees of freedom are determined. In the second phase, the directed graph is evaluated to find an actual solution, using a combination of construction steps and iterative techniques for cyclic dependencies.

Comparison

Local geometric propagation techniques are, like the local numeric propagation techniques, fast and efficient. They employ local information and incrementally solve the constraints. However, they have difficulties in dealing with constraint loops; these are usually handled by calling an external numeric solver. Some geometric propagation techniques use a predefined or user-defined order of evaluating the constraints, which renders them rather inflexible. Furthermore, it means that a solution may not always be found, as it depends on the correct order of propagation.

Geometric reasoning techniques determine symbolic solutions. From these symbolic solutions, actual solutions can be determined using the actual values of the dimensions. For computation of the symbolic solution, rule rewriting or graph-based techniques are used. Both classes of techniques are slow, compared to local propagation techniques, but they are much more powerful.

Rule-based systems use a general-purpose reasoning engine, and specify geometric knowledge as rules. The geometric knowledge can be easily extended, for example, to include new types of constraints. They do not take into account the structure of the constraint satisfaction problem that is to be solved.

Graph-based systems, on the other hand, analyze the constraint graph, and employ special-purpose reasoning techniques that use geometric knowledge. This facilitates a structural approach to a solution, but limits the extensibility of such systems. However, graph-based geometric constraint satisfaction techniques provide the best means to handle 3D geometric constraints, because they exploit the structure of the constraint graph, using special-purpose 2D or 3D geometric knowledge where appropriate.

2.5 Discussion

In geometric modeling, constraint techniques play an important role. They facilitate declarative specification of a geometric model, and enable checking and maintenance of its validity. In this chapter, a number of constraint satisfaction techniques has been discussed, both for general constraint satisfaction problems and for geometric constraint satisfaction problems.

General CSPs were classified on the basis of the domain cardinality of the constraint variables: finite or infinite. For finite domain CSPs, the possible values for a variable can be examined one by one. These techniques, however, are of little use if the domain sizes are large or even infinite, or if the number of variables can change due to a labeling action. In highly underconstrained problems, the number of solutions can be large. In infinite domains CSPs, numeric techniques and graph transformation techniques determine single values for variables. For sets of values, a generalization of infinite CSPs to tolerance CSPs was discussed.

The techniques for solving CSPs with infinite domains are general purpose. For the geometric domain, adaptations of infinite domain techniques can be used, e.g. local propagation of geometric transformations, but it is also possible to exploit knowledge of the problem domain in the satisfaction process. This is done in geometric reasoning techniques, which solve geometric constraint problems symbolically. For 3D geometric constraint problems, geometric reasoning technique are best suited, particularly those that are graph based.

3

Feature validation approaches

Features have been defined in the introduction of this thesis as representations of shape aspects that are mappable to a generic shape, and that are functionally significant. Since features have functional significance, they have a well-defined, functional meaning that is application dependent. For generic features, or *feature classes*, this intrinsic meaning is expressed by rules. These rules specify the conditions under which a feature is *valid*, i.e., exhibits a behavior that is in accordance with the meaning of the feature class. Such conditions involve relations within the feature, and relations with other features. In feature modeling, the *specification* of validity conditions is important for obtaining useful feature classes.

Besides the specification of feature validity conditions, their *maintenance* is another important aspect of feature modeling. When a feature model is edited, the validity of the features in the resulting model has to be ensured by the feature modeling system. This *feature validation* may be done by monitoring the validity of the features while the edit action is carried out, or by checking the validity of each feature after the edit action has been completed.

Two main approaches to feature modeling are distinguished that differ mainly in the way the validity conditions are specified and maintained. In *procedural* feature modeling, the validity conditions are expressed in terms of predefined procedures, which are executed automatically when a feature is edited. In *declarative* feature modeling, the validity conditions are expressed as *constraints* that should hold, leaving the actual maintenance of these constraints to a constraint solving system.

There is a strong relationship between constraint satisfaction, geometric modeling, and feature validation. This chapter first discusses the impact of constraints on geometric modeling as it has been in the 1980s, and then discusses important aspects of feature validation, followed by an elaboration on the feature validation approaches in procedural and in declarative feature modeling.

3.1 Constraint-based geometric modeling

The development of geometric modeling techniques made it possible to represent the objects that were being designed, as computer models, for easy modification, analysis and visualization. However, modification of a geometric model could easily lead to models that no longer reflected the intent of the designer. It was necessary to extend the geometric model with some representation of the design intent, specified by constraints.

In constraint-based geometric modeling, a model is built by specifying relations between geometric entities, which are to be maintained by the modeling system. Many relations can be specified declaratively as geometric constraints on these entities, e.g. a constraint declaring two faces to be parallel. Besides geometric constraints, algebraic or *engineering* constraints are used, e.g. to analyze material stress or flow.

Before we discuss constraint-based geometric modeling in more detail, let us first define some terminology. Model *variables* describe the shape and size of a geometric model. The variables depend on the representation of the model: e.g. in a boundary representation, point coordinates can be used. *Dimensions* are defined as the nominal values of geometric properties of an object. A dimension specifies a size, or a distance or an orientation between a pair of model entities. Dimensions may be defined on the basis of the model variables. An object is completely specified by a number of *parameters*, such as dimensions. The modeling system takes care of the translation from parameters to model variables.

In the past decade, two main constraint-based geometric modeling paradigms have emerged: parametric and variational modeling. These paradigms address both the way the model relations are specified in terms of constraints, and the way the modeling system maintains the specified relations when the model is edited [Chung and Schussel 1989]. In *parametric* modeling, geometric and engineering constraints are maintained separately by special-purpose solvers, whereas in *variational* modeling, geometric and engineering constraints are maintained simultaneously by a general-purpose constraint solver.

In *parametric modeling*, an object is defined by specifying the values of a restricted number of parameters. These parameters may be dimensions, but they may also be values without geometric meaning that are used to compute dimensions. Objects are defined on a generic level, and instantiated by specifying parameter values. Constraints are used to define the model parameters in terms of the model variables, and to express relations between model parameters. These constraints are satisfied sequentially, using specific geometric knowledge. Engineering constraints are specified separately, and solved decoupled from the geometric constraints. In many systems, the constraints are unidirectional de-

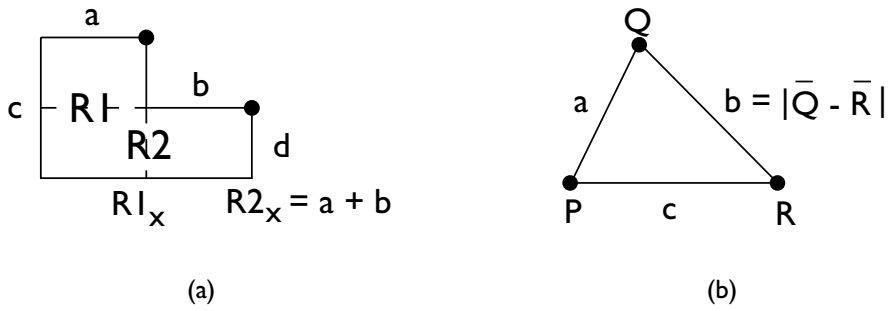


Figure 3.1: Direct (a) and indirect (b) definition of model variables in terms of dimensions [Juster 1992].

pendencies, i.e., they specify how one parameter depends on other variables or parameters. Solutions are thus determined that depend on the predefined order of satisfaction of the constraints. In other parametric modeling systems, the order of satisfaction can be altered by the user, but is sequential nonetheless. Constraint satisfaction methods that are employed in parametric modeling systems can thus be considered local propagation techniques; the values propagated may be simple numeric values, or geometric transformations.

Variational modeling [Lin et al. 1981] uses simultaneous equations to express constraints on the model variables. These equations are typically nonlinear, and are solved numerically with a general-purpose solver. Besides equations to define model dimensions, also engineering constraints may be specified; this is facilitated by the uniform way of expressing all constraints by equations.

If the variational model variables have been defined *directly* in terms of the object's dimensions, altering a dimension by the user easily results in an update of the model variable values by the modeling system. However, Juster [1992] and others believe that it may obstruct the user's train of thought in the modeling process if he has to think in terms of the model variables instead of the dimensions. Dimensions may therefore also be attached after the model has been defined, which in effect means that model variables are defined *indirectly* in terms of the dimensions. This may lead to computationally difficult equations that are hard to solve. However, it is more convenient from the user's point of view.

Figure 3.1 shows an example of each of the two types of parameterizations. In Figure 3.1(a), the model consists of two overlapping rectangles **R1** and **R2**. The model variables are the coordinates of the upper right hand corners of the two rectangles, whereas the model dimensions are chosen as **a**, **b**, **c** and **d**, as depicted. The model variables are defined directly in terms of the dimensions,

e.g. $\mathbf{R2_x = a + b}$. On the other hand, in Figure 3.1(b), the dimensions have been attached to the model afterwards. The point coordinate model variables are thus defined indirectly in terms of the dimensions, e.g. by $\mathbf{b = ((Q_x - R_x)^2 + (Q_y - R_y)^2)^{\frac{1}{2}}}$.

Parametric modeling has the advantage that geometric constraint satisfaction can be done more efficiently making use of special geometric knowledge. It provides better techniques for checking for ill-defined or degenerate solutions. On the other hand, specific algorithms are needed for each constraint type, thus new algorithms are required when the set of constraint types is extended. This disadvantage may be reduced when a formalized solution framework is used, and the specific algorithms can be easily reused [Keirouz et al. 1992]. Another disadvantage is that it may not always be possible to solve the constraint set in a predefined order, particularly if there are loops in the constraint set.

In variational modeling, there is no clear relation of the equations to the geometry, therefore it is hard to predict the effect of changing the value of a model variable. For underconstrained variational models, one of the multiple solutions is found. This solution, although valid with respect to the constraints, may be meaningless from the geometry point of view. This is due to the general-purpose nature of the solving method, which does not incorporate any knowledge of the application domain, i.e. geometric knowledge. Furthermore, numerical solving techniques are computationally expensive, not very robust since convergence is not guaranteed, and sensitive to initial guesses. On the other hand, variational modeling tools are more flexible, since they allow constraint loops, and support easier modification of design specifications by the user, because the specification order is not related to the solving process. Another strong point of variational modeling is its support of coupling engineering constraints and geometric constraints.

In constraint-based geometric modeling, a user still has to specify a model in terms of low-level geometric entities. Such entities are often not compatible with the more abstract level of thinking of the user. Instead of considering a model in terms of edges and faces, he may want to consider it as a collection of related high-level objects with specific functions and parameters, i.e. features. Specification of a model in terms of features provides the user with modeling entities closer to his way of thinking; it also provides a powerful means to ensure the validity of the specified model. The following sections discuss feature validity, and two main approaches to ensure it.

3.2 Feature validation aspects

The meaning of a feature involves a number of feature properties, which concern the function of the feature. Directly related to the *geometry* of the feature are properties such as the feature shape geometry, its dimensions, and its position and orientation. Furthermore, the feature shape topology is a property of a feature. Other properties concern the *interaction* or intersection of features; these are related to the feature geometry, but may have a very different functional meaning, depending on the application.

Feature validation means ensuring that all feature instances in the model are in a state that is in accordance with their intended meaning. Therefore, it is ensured that the feature conforms to the conditions for the properties as specified in the *feature class*, and to any additional constraints applied to the feature instances *after instantiation*.

Since feature validation involves several different feature properties, different types of validity conditions have to be considered. Shah and Mäntylä [1995] distinguish validation of dimension limits, location limits, feature attachments, and feature interactions; in addition, we distinguish validation of the feature shape.

Shape validation concerns the internal consistency of the feature shape geometry and topology. An example is a restriction on the sides of a slot to be parallel.

Dimension limits concern the size of a feature. An example is a maximum on the depth of a blind hole feature that is manufactured with a particular type of machine.

Location limits concern the position and orientation of a feature. An example is a restriction that the pockets in a pocket array must be mutually parallel.

Feature attachment validation involves the compatibility between the model entities on which the feature is instantiated and the feature entities involved. In a *feature attachment* the feature parameters that depend on the model are determined; these include feature dimensions and feature location. The following section gives examples of such dependencies. Attachment validation ensures, for example, that a rectangular slot feature is attached to a planar model surface. If a feature is attached to the model by attaching feature faces to model faces, the mutual compatibility between the attachments, and also between the attachments and the feature shape has to be checked. For example, for a cylindrical hole that is used to put a bolt through in some assembly connection, this bolt has to be fixed with two nuts. Therefore, it should be checked that the two attachment faces are parallel, see Figure 3.2 on the next page. In this example, the functional requirements of the feature involve attachment compatibility.

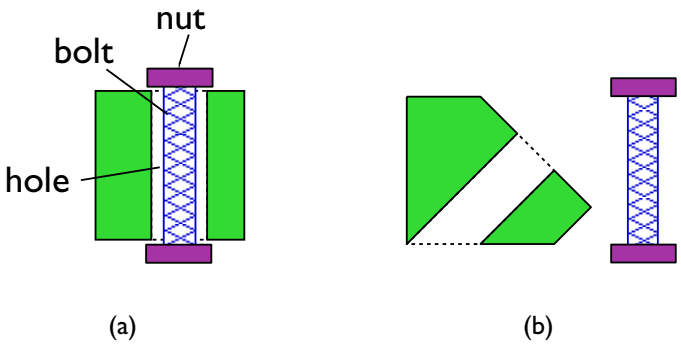
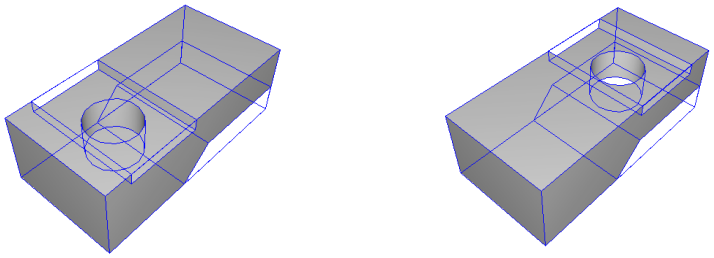


Figure 3.2: The function of the bolt hole requires compatibility of its attachments. In (a), the hole is attached to two parallel faces; in (b) it is attached to two nonparallel faces, which makes it impossible to fix a bolt in the hole.

Feature interaction validation involves the boundary and volumetric intersection of a feature with other features in the model. Such intersections may cause a feature to get into a state that is no longer in accordance with its intended meaning, because the topology of the feature model is changed. Figure 3.3 on the facing page depicts an example of such a situation. Consider the blind hole that is attached to the slot in (a). The function of the blind hole is dependent of the presence of a bottom face, for example, if it is used as a pin-blocking face in an assembly connection. If the slot is moved, the attached blind hole moves along (b). We see that the blind hole bottom face has disappeared, which is due to a boundary or volumetric interaction. Thus, the feature shows a behavior that is no longer in accordance with its intended functional meaning, i.e., it is not a valid blind hole anymore. Shah [1991] gives examples of feature interactions; Bidarra and Bronsvoort [1996] give a systematic analysis of feature interaction classes.

The different types of feature validity specifications are formulated in a language. Many languages are used for this, ranging from general-purpose programming languages to highly-tailored feature specification languages. The disadvantage of general-purpose languages is the strong dependency of the feature specifications on the modeling system implementation. Therefore, special-purpose feature specification languages have been developed that decouple the feature specifications from the modeling system implementation. Such feature specification languages may be quite expressive, e.g. if they embed feature validity condition specifications as constructs in the language.



(a) (b)

Figure 3.3: A change in the position of the slot in (a) causes the bottom of the attached blind hole to disappear (b) due to a feature interaction [Bidarra, Dohmen and Bronsvoort 1997].

3.3 Approaches to feature validation

When considering previous approaches to feature validation, the *specification*, the *representation*, and the *maintenance* of validity conditions are important. From the user's point of view, feature validity specification is likely to be the most important aspect. This involves the kind of validity conditions that can be specified, and also the specification method that is used. Validity representation methods concern the internal storage of the validity conditions, and are thus related to the way the validity conditions are maintained. These aspects will be discussed for the two main approaches to feature validation, procedural and declarative feature validation.

Procedural validation

In procedural feature validation, feature classes are specified by predefined procedures. These procedures specify methods for instantiating a feature, creating its geometry, modifying it, and maintaining its validity.

A feature is instantiated by attaching it to one or more *parent* features. A feature has a local coordinate system, to specify a position and orientation when it is instantiated. This is done relative to one of its parents; in this way, a hierarchy of coordinate systems is formed that reflects the structure of the feature model. The feature procedures specify how the feature parameters are computed.

Two examples of procedural feature modeling systems are the ASU Features Testbed [Shah et al. 1990] and GeoNode [Van Emmerik 1991]. In the ASU Fea-

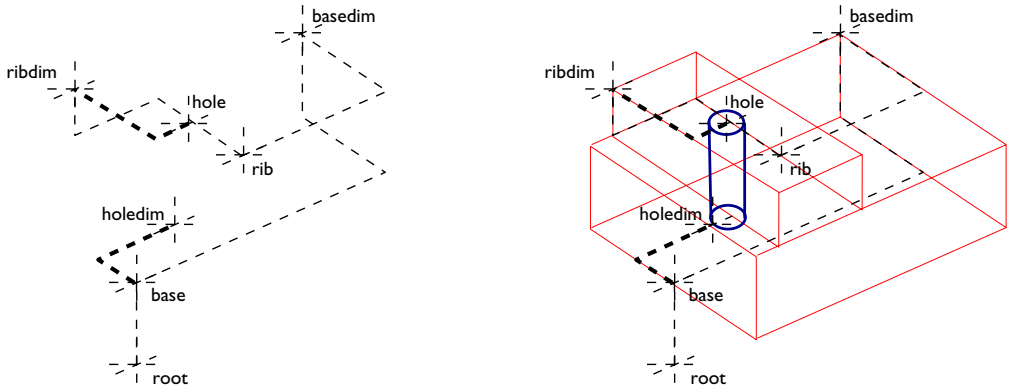


Figure 3.4: Attachment of a through-hole feature in GeoNode, using a hierarchy of coordinate systems. To two coordinate systems in the hierarchy (a), a through-hole feature is attached (b).

tures Testbed, a feature is attached to a single parent feature; using rules, its parameters are derived explicitly from other parameter values, possibly of the parent feature, and thus a parameter hierarchy is formed.

In GeoNode, a feature is attached to coordinate systems that have been created by the user. These coordinate systems are related by geometric constraints; the positions of the coordinate systems are used to compute the feature parameters. This provides the user with more flexible feature attachments.

For a GeoNode example, see Figure 3.4. First a hierarchy of coordinate systems is built (a), which is used to attach a base block and a rib feature, after which a through-hole feature is attached to coordinate systems relative to the base and the rib (b). For each of the features, one coordinate system specifies the feature location, and another specifies the dimensions of the feature. The radius of the through-hole is determined by constraints that specify the 2D position of its dimensioning coordinate system **holedim** relative to its locating coordinate system **hole**.

In procedural feature modeling systems, feature validity is maintained by evaluating the dependencies between a feature and its parents in the feature model hierarchy. Parameter evaluation is thus done in a unidirectional way along the hierarchy, using the predefined procedures incorporated in the feature, and thus in a predefined order. In this process, the validation rules on parameters are also evaluated. Feature attachment validation requires evaluation of special-purpose

attachment rules that differ for each feature class and that possibly depend on the class of the parent feature. Feature interaction validation, if at all possible, also requires special-purpose rules for particular combinations of interacting features.

In the procedural approach to feature validation, specification and maintenance of the validity conditions are not separated, but rather tightly coupled. The unidirectional dependency relations between features specified by the user are used directly in the process of maintaining the validity conditions, and do not allow a flexible validity maintenance scheme. On the other hand, an advantage is that evaluation can be done in a simple and efficient way, which facilitates, for example, validation during graphical user interaction. For validation of feature attachments and interactions, special-purpose rules have to be provided that depend on the classes of the features involved, and which are therefore not generic.

Towards the end of the 1980s, feature modeling systems emerged that supported a more declarative approach to feature validation. Such systems enabled the user to focus on which validity conditions were needed for a particular feature model, and not on how these conditions are maintained. The next section elaborates on this declarative approach.

Declarative validation

In declarative feature validation, feature classes are specified declaratively as a combination of geometric entities and constraints that specify validity conditions. Geometric entities may be volumetric primitives, or they may be boundary elements such as edges and faces. Hierarchical model relations are also expressed as constraints. The constraints are embedded in the feature class, which ensures that when a feature of a certain class is instantiated, instances of its constraints are created automatically. The validity conditions are maintained by a constraint solver that is separated from the feature specifications. These aspects of declarative feature validation will now be elaborated, by discussing some implementations of this approach.

In the feature modeling system EXTDesign, a description of feature geometry is combined with a declaration of embedded constraints [Laakko and Mäntylä 1993]. The constraints specify algebraic relations between feature parameters, or linear distance relations between feature elements. Constraints are stored in a constraint graph; parent-child relations between features are stored in a separate feature graph. Constraint specification is supported by a tool for translating high-level constraints into groups of connected low-level constraints [Lagus et al. 1994]. This translation is done at instantiation time, after which only the low-level

constraints are considered. For solving the low-level constraints, the SkyBlue constraint solver [Sannella 1992] is used.

Vieira [1995] and Brunetti, Ovtcharova and Vieira [1996] propose constraints that concern the shape description of features, and constraints that concern the semantic description. To describe the shape of a feature, geometric constraints between feature elements are used. These are represented in an entity-relationship graph [Brunetti et al. 1995]. Geometric constraints are represented internally by equations. The semantic description of a feature is specified by three types of feature validation constraints: on the feature location, on parameter values, and on the feature function and its allowed interaction with other features. The third type includes constraints on allowed intersections, constraints for attachments, and constraints on the covering of boundary feature elements by other feature elements. The approach is to be incorporated in the feature modeling system SINFONIA; no implementation details are provided on how the feature validity conditions are maintained.

Shah, Balakrishnan, Rogers and Urban [1994] propose a declarative approach for the second implementation of their ASU Features Testbed modeler. In the declarative approach, explicit parameter hierarchies have been abandoned. Instead, features and feature parameters are specified by declaring geometric constraints between feature elements. They propose to represent the feature model by a constraint graph. When a feature is instantiated, an associated constraint graph containing the embedded feature constraints is instantiated and merged with the constraint graph that represents the feature model. For constraint solving, they propose to map the constraint graph onto a graph that is solved by a geometric constraint solver [Kramer 1992*b*].

Salomons [1995] distinguishes three types of constraints, used in the feature modeling system FROOM. First, constraints between feature elements for internal relations (e.g. adjacency of the elements of a feature), and for elements with an external entity (e.g. relative position of a feature). These are implemented by constraints on nominal geometry, and solved by degrees of freedom analysis [Kramer 1992*b*]. Secondly, constraints specified on dimensions, which are implemented by algebraic constraints on parameters, and solved by simulated annealing [Thornton and Johnson 1993]. Thirdly, tolerance constraints, which are implemented as variational geometry constraints, and solved by a technique based on work by Clément and Rivière [1993]. The three solvers work independently of each other. This has the drawback that a dimension change causing a geometry change, requires re-solving the geometric constraints, and vice versa. Changes in tolerances do not affect nominal geometry, nor dimension values. For the representation of features, conceptual graphs are used. These are bipartite graphs containing concepts (e.g. feature faces) and conceptual relations (e.g.

parallel). Constraints of the three types are stored in three separate conceptual graphs.

The previously mentioned approaches focus on geometric and algebraic constraints for feature validation, and do not adequately address the issue of changes in feature model topology due to interaction of features.

Bidarra and Teixeira [1994] define interaction validity conditions for a feature by expressing its meaning with *semantic constraints* that specify for each feature face the extent to which it has to lie on the product boundary. They specify these constraints as logical predicates that can be verified. Also, they have made a classification of feature intersection types, on the basis of the changes to feature elements [Bidarra and Bronsvoort 1996]. Furthermore, validation techniques for different intersection types have been investigated [Bidarra, Dohmen and Bronsvoort 1997].

Mandorli et al. [1997] describe, what they call, self-validating features, which incorporate validation rules. Such rules check the state of the topologic entities the feature comprises; an example is a rule checking that the entrance face of a cylindrical hole has a convex edge loop. Each feature considers locally the state of the low-level geometric entities in the B-REP-based geometric model that correspond to its feature topology entities. An advantage of this approach is that interaction validity conditions can be fine-tuned for a particular feature class. The limitations of this approach have to do with the B-REP based representation, and with the locality of the validation. Firstly, when boundary entities disappear due to a feature instantiation, it is difficult to check interaction validity since there is no longer a direct correspondence with the feature topology entities. Secondly, due to the local feature validation that is performed, not all types of feature interactions can be detected, such as one that results in a disconnection of part of the model. Such interactions can only be detected if the whole model is considered. Thirdly, it is difficult to specify appropriate reactions of the system when some feature interaction is detected, as only local information is available.

Chen and Hoffmann [1995*b*] focus on flexible feature attachments. For feature attachments, they have made operational definitions that specify the effect of attaching a feature to the model in terms of Boolean operations. In a feature specification, feature geometry is defined by geometric operations such as sweeps. Validity conditions are specified by geometric constraints and algebraic constraints, which are represented uniformly, and maintained by a 2D geometric constraint solver [Bouma et al. 1995]. Furthermore, semantic constraints are specified [Hoffmann and Juan-Arinyo 1997] similar to those of Bidarra and Teixeira [1994].

3.4 Discussion

In this chapter, we first presented two main approaches to constraint-based geometric modeling. Both consider the specification and maintenance of geometric models by constraints on geometric entities. In parametric modeling, the user is responsible for specifying the constraints in such a way that the modeling system is able to determine a geometric configuration of the constrained entities. The system satisfies the constraints sequentially, often in the order of specification. This rather inflexible scheme is relaxed in the variational modeling approach, in which all constraints are solved simultaneously, regardless of their specification order. A drawback of this approach is that no geometric knowledge is used in the solving process, which may lead to unexpected and possibly geometrically meaningless models.

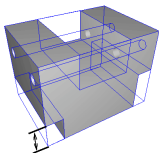
Some researchers argue that a flexible modeling system should incorporate both parametric and variational modeling techniques. Such a system should provide parametric geometry to specify shape, and variational modeling tools for specifying and solving engineering constraint problems. However, such constraint-based geometric modeling systems still require the user to model in terms of low-level geometric entities, which seriously limits the suitability for design.

Features provide a means for the required abstraction from the low-level geometry. They have a well-defined meaning expressed by feature validity constraints. In the modeling systems discussed, various constraint types were encountered, including geometric, algebraic and semantic constraints. To ensure that the features in the model are valid with respect to their intended meaning, feature validation is performed by the modeling system. Feature validation concerns conditions for its shape, limits on its dimensions and location, on its attachments and on its interactions with other features. Two main approaches to feature validation were discussed. In procedural validation, the maintenance of the validity conditions is tied to the way the conditions were specified by the user. This limits both the expressiveness of the specifications, and the possibilities of maintaining the validity conditions. Declarative validation separates feature validity specification and maintenance. The separation facilitates specifying validity conditions as local feature properties, embedded in feature class specifications.

A separation between specification and maintenance of the feature validity conditions is required for flexible and expressive feature validity specification combined with powerful feature validity maintenance. In such a separated validity maintenance scheme, the constraints of different types may be solved in a coupled way by representing them uniformly, e.g. as equations. Alternatively, they may be solved decoupled with specialized solvers.

The separation of validity specification and maintenance ensures that the system can handle the validation of feature attachments and feature interactions in a generic way. The next chapter presents a new declarative approach to feature modeling and validation that realizes this.

Part II Feature validation



4

Multiple-view feature modeling

In a concurrent engineering environment, multiple interpretations of a product can coexist. Such interpretations, or views, are application dependent and specified in terms of features. The views need to be maintained simultaneously, to truly support concurrent engineering. It is required that a new view of a product can be invoked, for example, when a new product life cycle phase has to be considered. Simultaneous view maintenance facilitates that changes to the product that are made in any of the views are reflected in the other views.

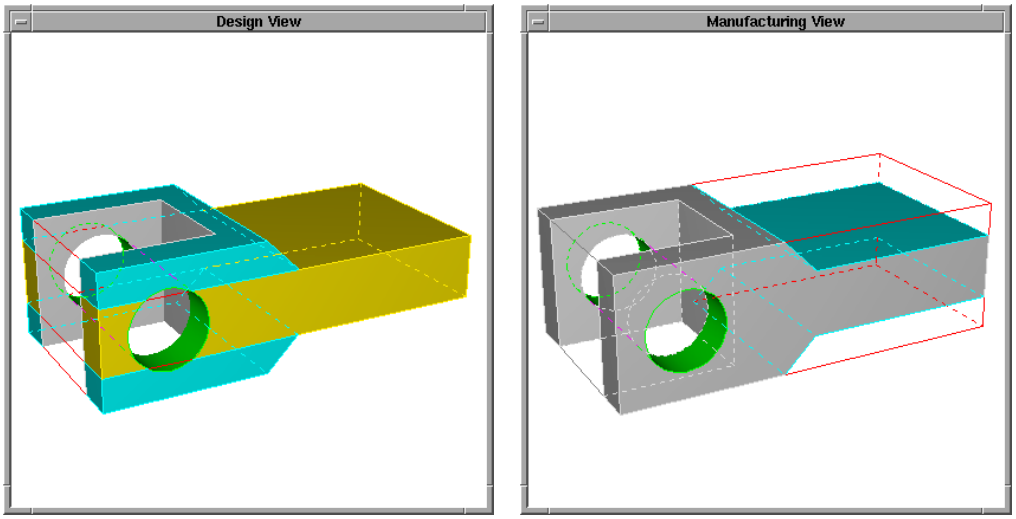
To give an example, Figure 4.1 on the following page shows two views of a part. The part is represented in the design view by a base block, two ribs with a slanted face, a through-hole and a through-slot. In the manufacturing view, the part is represented by a stock which is modified by two steps with a slanted face, a through-hole and a through-slot. The design view reflects the way the product has been designed using both additive and subtractive features; the manufacturing view consists only of a stock and subtractive features that correspond to manufacturing operations.

This chapter first presents our approach to multiple-view feature modeling. Then, it elaborates the concept of a product model to support this. Finally, it elaborates how this approach is implemented in a prototype modeling system.

4.1 Multiple views

For simultaneous maintenance of multiple feature views, a coupling between the views is required. Approaches have been developed by others to make a direct coupling between views, for example from a design view to a manufacturing process planning view. To prevent having to provide direct couplings between all views, a coupling through some central representation is desirable.

As was already stated in the introduction, in this research the focus is on features that represent geometry. The different views share the product geometry; therefore, the central representation contains the geometry specified in the



(a) (b)

Figure 4.1: A design view (a) and a manufacturing view (b) of a simple part.

views [De Kraker et al. 1995]. From this central geometry a feature model is derived when a view is invoked or *opened*. The derivation of a feature model has been completed successfully when all views are *consistent*, i.e. when they represent the same geometry. This is the subject of the work of De Kraker [1997].

When a view is *edited*, i.e. when features or constraints are added or removed, or when feature parameters or constraint parameters are changed, the changes have to be propagated to, and thus reflected in, the other views. Therefore, we propose to link the feature models of the views by constraints, which reflect the fact that the views represent the same product geometry.

Finalizing, in our approach we have multiple views of a product that each contain a feature model. A representation that is central to all the views contains both the product geometry and the constraints specified in the views plus the constraints between the views, see Figure 4.2 on the next page.

An edit view operation may be in conflict with another view. To resolve such conflicts, each view is assigned a *priority*, which is used to decide whether the edit operation is executed or not [De Kraker et al. 1997*b*]. If the edit operation is executed, it is propagated to the other view. If not, the edit operation is canceled, which is reported in the corresponding view. Examples of such situations are given in Section 5.4.

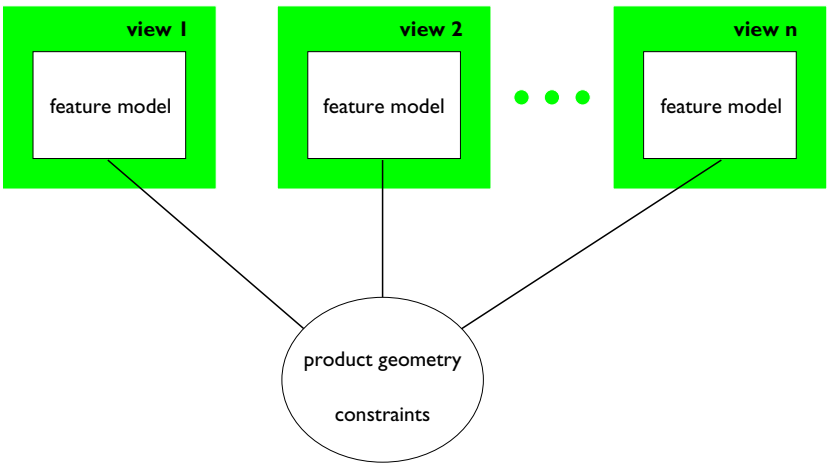


Figure 4.2: Multiple feature views of a product are coupled through a central representation.

To propagate feature add and remove actions from the edit view, the other views are reopened incrementally [De Kraker et al. 1997*b*]. To propagate feature parameter changes, and constraint add and remove actions, *link constraints* between feature models of different views are used [Dohmen et al. 1996]. These are elaborated in Section 5.3.

In the previous chapter, the use of a declarative approach to feature modeling was advocated, because it separates specification and maintenance of the feature validity conditions. In our approach, features are specified declaratively in a general-purpose object-oriented language. The object-oriented paradigm has been found to be very suitable for feature modeling [Sreevalsan et al. 1990]. An object-oriented language contains a class-object mechanism to facilitate both the specification of feature and constraint classes, and the specification of instances of these classes. The classes are specified by attribute types and methods, the objects by attribute values. The next section discusses a representation to reflect the separation of validity specification and maintenance.

4.2 Product model

A *product model* contains the different views and their feature models that describe one product. The representation of the product model consists of two different levels: specification and maintenance, see Figure 4.3. At the specification

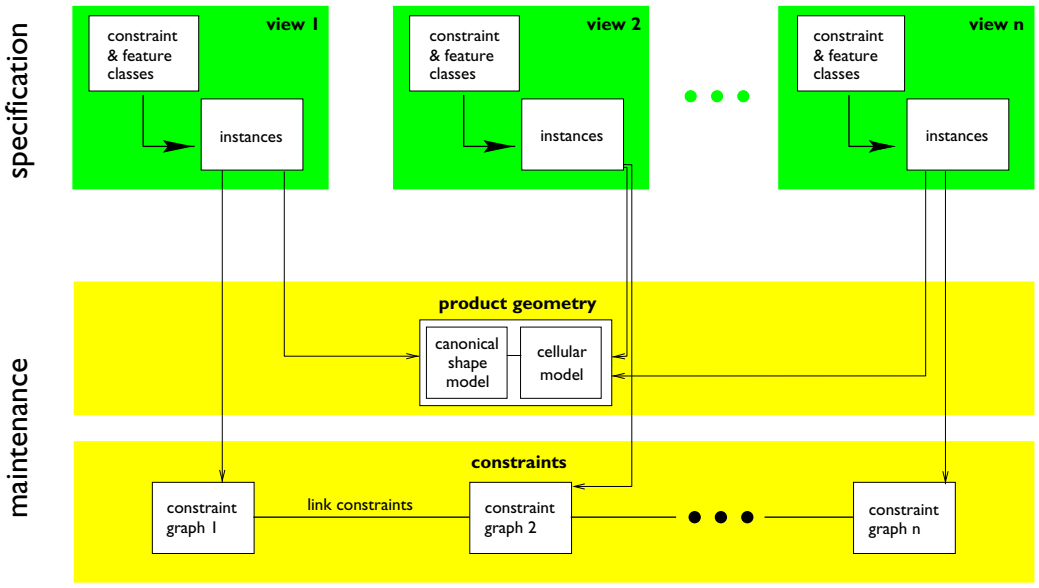


Figure 4.3: Product model representation.

level, views and features and their properties are specified, such as application-specific parameters and attributes, and constraints to specify feature validity conditions, including relations with other features. At this level, both classes and instances of features and constraints are specified. At the maintenance level, feature geometry and feature validity are maintained.

Several considerations have been made on how to enable feature validity specification which integrates feature geometry and constraints. Symbolic handles must be available for geometry access and constraint specification. These handles are provided by feature elements. For example, the closure top and bottom faces of a through-hole can be used to attach the hole to a model, using constraints. Geometry handles must not only be available in a newly to be added feature, but in the feature model as well.

In the history-based approach, a new feature is attached to topologic entities in the B-REP which results from previous modeling operations. This approach is applied in many feature modeling systems, e.g. [Parametric 1993, Chen and Hoffmann 1995*b*], however, it raises topologic naming problems. A general property of the B-REP is that entities may be destroyed, split, or merged as a result of a modeling operation. *Persistent topologic naming* is defined as identifying and tracking topologic entities when a geometric model is modified [Kripac 1995]. Chen and Hoffmann [1995*a*] show some typical problems that occur when fea-

tures are attached to a B-REP. Their examples demonstrate naming problems and the associated ill-defined semantics of the feature model when it is edited; for example, when a round feature is attached to part of a split B-REP edge, and later the split edge is reconnected, it is not clear what should be done with the attached round

To resolve naming problems and the ill-defined feature model semantics, we propose *attachments of canonical feature elements*. Canonical feature elements are elements of *canonical*, or generically defined, shapes, i.e. elements of topological or geometrical parameterized shapes. They are persistent, and thus can be used as constraint variables. Following this scheme, a new feature is attached with its topologic elements to topologic elements of other features.

When changing feature parameters, canonical feature elements and shapes have another attractive property. Whereas in history-based modeling, a feature parameter change requires a (partial) re-evaluation of the feature model geometry, our canonical feature shapes allows to directly change parameters of one feature.

Currently, only coplanar attachments have been implemented. Other, oblique and nonplanar, attachments must be investigated further.

Summarizing, by using attachments of canonical feature elements, we can specify a feature model using constraints on these topologic elements. Furthermore, we avoid naming problems, and the feature model semantics are well-defined when the model is edited.

At the specification level, views and features are specified in LOOKS, a strongly typed object-oriented imperative programming language [Peeters 1993, Peeters 1995], which resembles the JAVA™ language [Niemeyer and Peck 1996]. An interpreter for LOOKS is used to process view, feature and constraint specifications.

In a view specification, a view is assigned a library of feature and constraint classes. The feature model is specified by instances of these feature classes. For a feature class, three types of feature constituents are available: *shape*, *validation constraints* and the *nature* of a feature. Since only volumetric features are considered in this thesis, a shape is represented by a parameterized solid. To specify a compound feature, multiple shapes are combined. To specify feature validity, different constraint types that operate on feature elements and feature parameters are available. These constraints are embedded as attributes of a feature class. The feature nature is either additive or subtractive, specifying whether the feature adds or removes material to or from the product. The inheritance mechanism of LOOKS is used to create feature hierarchies. In Chapter 5, example feature specifications with validation constraints are given.

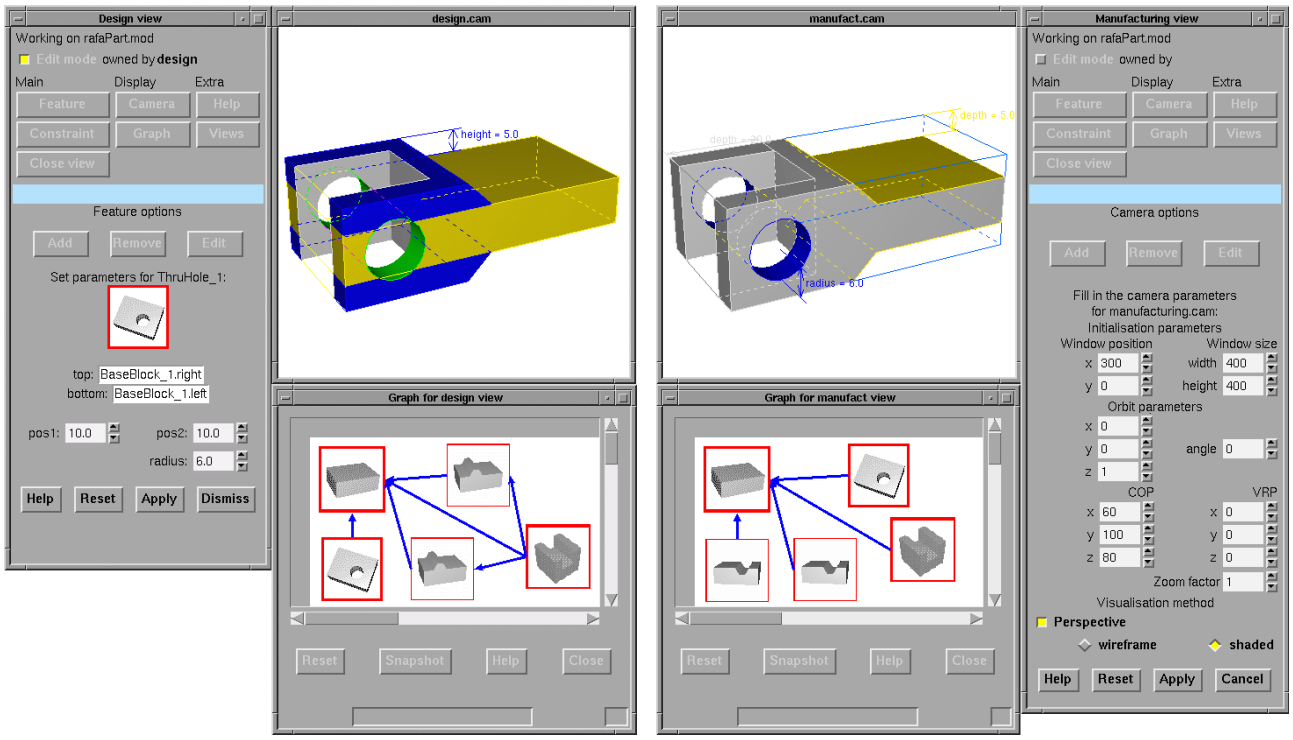


Figure 4.4: The graphical user interface of SPIFF.

At the maintenance level, the specified feature shapes and constraints are stored and maintained. For the feature shapes, a *canonical shape model* and a *cellular model* are used, which represent the product geometry. The canonical shape model is a CSG tree structure; it implicitly contains the order of instantiation of the shapes. Cells in the cellular model are volumetric and represent all feature intersections; they can have overlapping boundaries, but they cannot have overlapping volumes. Because they reflect all feature intersections, they can have any shape. With each cell, for each view, it is stored to which features the cell belongs. For information on how the cellular model is created and maintained, see [Bidarra, De Kraker and Bronsvoort 1997].

The validation constraints are stored in a *constraint graph*, which contains subgraphs, one for each view. The maintenance of this constraint graph is the subject of Chapter 6.

4.3 Implementation

A prototype multiple feature-view modeling system, SPIFF[1], has been implemented. Figure 4.4 on the facing page shows the Graphical User Interface (GUI) of SPIFF [Vergouwe 1996]. The user interface comprises a set of windows for each view. Here, a design view (left) and a manufacturing view (right) of the part shown in Figure 4.1 on page 50 have been opened. For each view, a control panel is available that is used to edit the feature model. Also, the user interface provides for each view cameras that visualize the feature model using newly developed feature visualization techniques [Versluis et al. 1997]; visualization of feature closure faces, feature interactions and feature parameters are included. Furthermore, the GUI provides a graph window that displays the structure of the feature model; in this window, feature instances, feature dependency relations such as attachment and relative location, and model validation constraints are depicted.

The architecture of SPIFF is shown in Figure 4.5 on the following page. Rectangular boxes represent functional entities, and rounded boxes represent data structures. The architecture consists of three functional levels that have a corresponding implementation in three different languages. These levels are: user interaction, feature model specification, and feature validity maintenance.

The *user interaction level* has been implemented in TCL/TK. The View Manager maintains the set of open views. The GUI stores a set of windows for each view and processes edit operations in a view. The View Manager and the GUI generate LOOKS commands that contain feature model specifications in a generic way, using command template files. They pass on these specifications to the feature model specification level.

The *feature model specification level* specifies a feature model for each view as instances of LOOKS feature and constraint classes. These LOOKS objects are stored in the Feature Manager, which passes their C++ counterparts to the feature validity maintenance level.

The *feature validity maintenance level* contains a Constraint Manager and a Feature Geometry Manager, which together maintain feature validity in C++. The Constraint Manager maintains a constraint graph deploying several solvers, see Chapter 6. Currently, a geometric and algebraic constraint solver have been implemented. The Feature Geometry Manager stores all feature shapes in the Canonical Shape Model. The Canonical Shape Model is evaluated into the shared Cellular Model. The C++ data structures have been implemented using the functionality of the NIH [Gorlen et al. 1991] and LEDA [Mehlhorn and Näher 1995] class libraries, and the ACIS™ [Spatial 1996] geometric modeling toolkit.

[1] Named after Spaceman Spiff, interplanetary explorer *extraordinaire* [Watterson 1995].

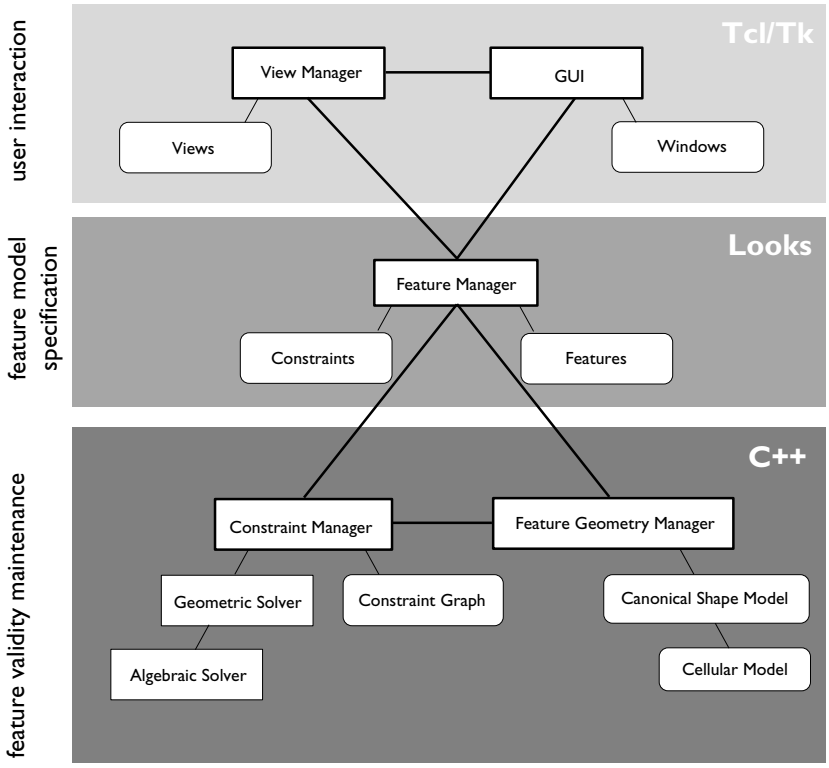


Figure 4.5: SPIFF system architecture.

The system architecture presented provides a clean functional separation of the system into layers. The modularity of the system architecture enables extending a layer with new functionality, without effecting the other layers. Each layer is implemented in a programming language that best suits the layer's requirements. For the GUI, TCL/TK is a language tailored to specifying and prototyping user interfaces. LOOKS is a strongly typed object-oriented language for specifying extensible sets of feature and constraint classes, and for implementing the Feature Manager functionality. Finally, C++ is used for implementing validity maintenance, which requires fast computations for constraint solving and geometry operations. For validity maintenance, the functionality of external class libraries is used.

Interface functions facilitate the communication between the managers in the different layers. Requests and answers are sent between managers in adjacent layers, or are sent within the same layer. The GUI sends modeling actions as LOOKS statements, which are executed in the LOOKS layer as member functions of the Feature Manager, features, or constraints. The Feature Manager coordi-

nates the managers in the C++ layer. The GUI queries the Feature Manager for the results of the modeling actions.

The coupling between the GUI and the LOOKS layers is done through LOOKS statement templates; TCL/TK objects and LOOKS objects have a coupling that uses object identifiers. The separation between the TCL/TK and LOOKS layers enables the GUI to be independent of the feature and constraint specification language. However, this coupling requires much communication; for example, when a feature parameter is changed, all model changes have to be collected, after propagating the parameter change, by querying all feature instances.

The coupling between the LOOKS and the C++ layers is embedded in the LOOKS interpreter, since the instances in LOOKS have C++ counterparts. The separation between the LOOKS and C++ layers enables generic, extensible feature and constraint specifications, and enables validity maintenance by the C++ managers.

Within each layer, modularity is achieved by a functional separation into different managers. This functional separation has the practical advantage that each manager can employ specialized class libraries. Furthermore, the separation facilitates rapid and concurrent development of prototype software.

A modeling session with the implemented system is shown in Chapter 9. It demonstrates both the usability of the concepts presented in this and subsequent chapters, and their successful implementation.

5

Validity specification and maintenance

This chapter presents the constraint types that are used to specify feature validity conditions. It gives examples of their use in feature class specifications, and demonstrates how new constraint types are created. Then, it elaborates how validity is maintained across multiple feature views.

5.1 Constraint types

Attach constraints specify the attachment of a feature to a feature model. They are defined between elements of different features, e.g. faces. An attach constraint in a feature model specifies two kinds of relations simultaneously: a structural relation and a geometric relation.

The structural relation is a hierarchical dependency relation of the attached feature on other features. It is used instead of the parent-child relations between features that are used in many other systems. The advantage is that a feature instance does not necessarily have only one parent feature; instead, it can be attached to feature elements of several features, each of which is called a parent of the feature.

The geometric relation of an attach constraint defines the remaining degrees of freedom of the attached feature element, relative to the other feature element. These degrees of freedom depend on the feature element types and the attach constraint type.

Two types of attach constraints are available, Contact and Align, which both specify coplanar attachment of one feature face to another. A Contact constraint specifies overlapping attachment, an Align constraint specifies nonoverlapping attachment. In the attachments, the normals of the feature faces have either equal or opposite directions, depending on the nature of the features.

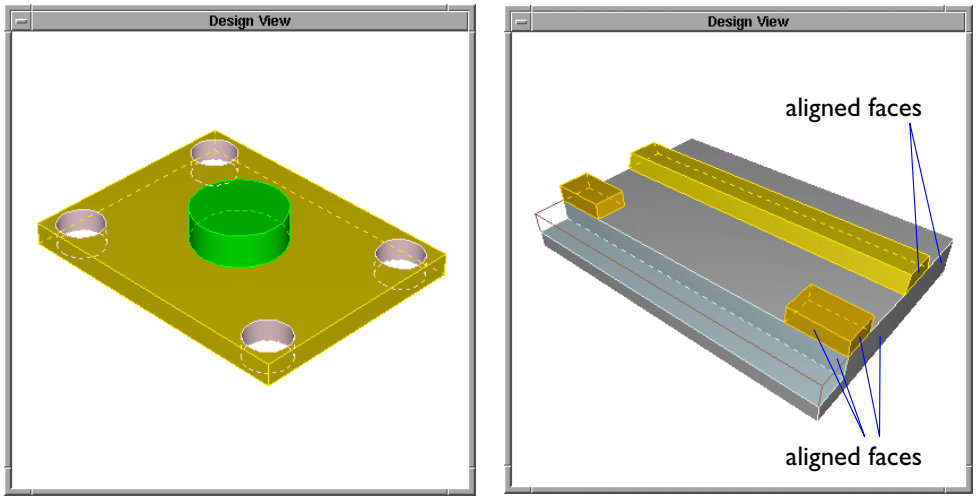


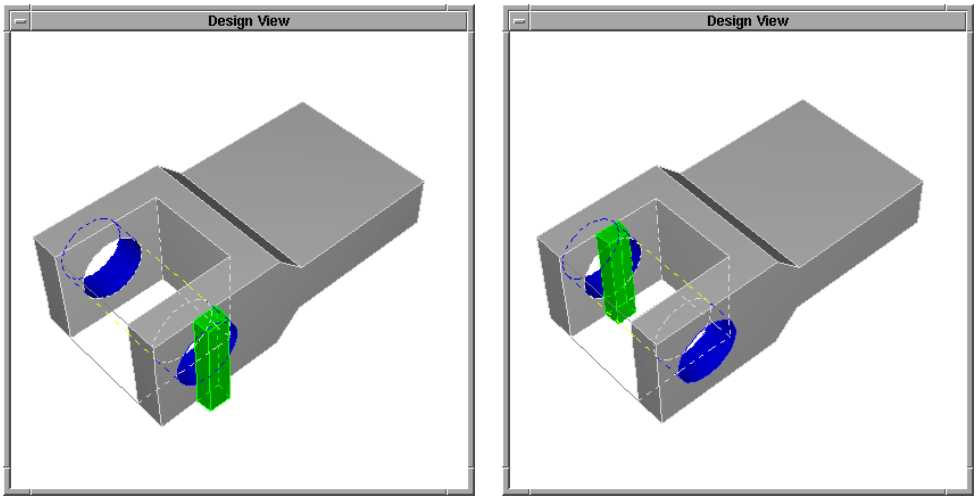
Figure 5.1: Different attach constraint types are available. In (a) Contact constraints are used for overlapping attachment; in (b) Align constraints are used for nonoverlapping attachment.

Figure 5.1 shows some examples of the use of attach constraints. In (a), a Contact constraint on the protrusion bottom face is used to attach the protrusion to the base block. Note that for feature shapes the face normals always point outwards of the feature shape volume. For two additive or two subtractive features, a Contact constraint specifies opposite directions of the attach face normals. The protrusion has three remaining degrees of freedom in the plane of the base block top face: two translational degrees of freedom, and one rotational about the face normal.

Similarly, in (a) Contact constraints are used to attach each of the through-holes to the base block. For a combination of an additive and a subtractive feature, a Contact constraint specifies equal attach face normal directions.

Figure 5.1(b) shows a model in which a rib feature has been attached with one Contact and two Align constraints. For two additive or two subtractive features, an Align constraint specifies equal attach face normal directions.

Also, corner protrusions have been attached, using, for each of them, one Contact and two Align constraints. They align the corner protrusion with the side of the base block and the back of the step. For a combination of an addi-



(a) (b)

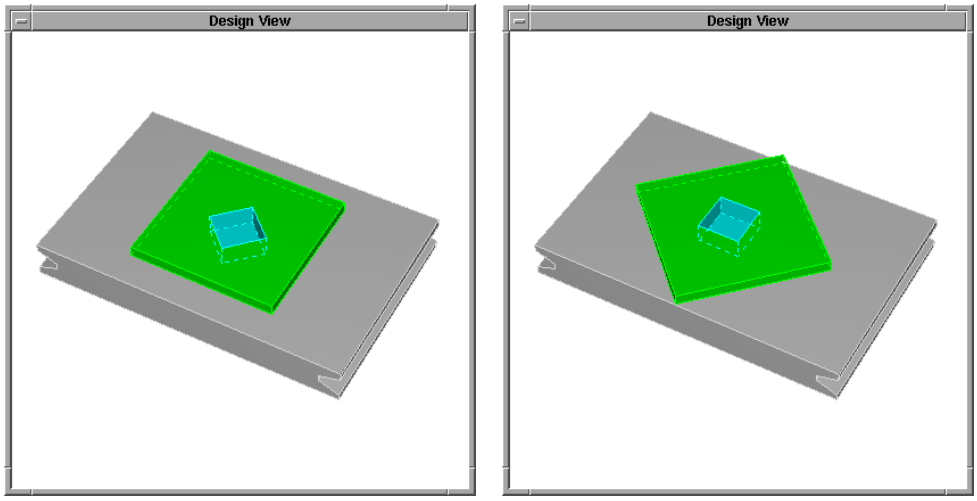
Figure 5.2: Semantic constraints prevent undesirable behavior. In (a) a NotOnBoundary constraint on the through-hole top face is violated, whereas in (b) a NoVolumetricIntersection on the through-hole volume is violated.

tive and a subtractive feature, an Align constraint specifies opposite attach face normal directions.

Semantic constraints specify topologic properties of feature elements. In SPIFF, semantic constraints are used similar to those defined by Bidarra and Teixeira [1994]. For a vertex, edge or face, a semantic constraint specifies the extent to which the element must lie on the product boundary. For a volume, a semantic constraint specifies the extent to which the volume is allowed to be intersected by other feature volumes instantiated later.

Consider the part of Figure 5.2. The through-hole feature shape is a cylinder that is subtracted. The cylindrical side face must at least be partly on the boundary, and the top and bottom face must not be on the boundary. When a rib is placed over the hole, the semantic constraint on the top face is violated, as shown in Figure 5.2(a).

If a through-hole is involved in a pin-hole connection feature in an assembly view [Van Holland and Bronsvoort 1997], a semantic constraint on its volume may be imposed that declares that no additive feature instantiated later may intersect the hole. For example, Figure 5.2(b) shows how the volumetric semantic constraint is violated by a rib placed in the through hole. However, in both Fig-



(a) (b)

Figure 5.3: Geometric constraints specify the position and orientation of the pocket relative to the protrusion. If the location of the protrusion in (a) is altered, the attached pocket will move accordingly (b).

ures 5.2(a) and (b), the through-hole is intersected by a through-slot, which is allowed.

Semantic constraints on feature faces are of types OnBoundary and NotOnBoundary. Each constraint type has a parameter that is equal to either completely or partly. This parameter expresses the extent to which the feature face must or must not lie on the product boundary. Such constraint parameters enable fine-tuning of feature classes, as will be shown in Section 5.2.

Geometric constraints specify geometric relations between feature elements. For faces, geometric constraints specify a relation involving the face origins and normals. The location of the origin of a face is predefined; it depends on the type of shape. For example, for a face of a block shape, the origin is located in a corner, whereas for a cylindrical shape, top and bottom face origins are located at the center of the face.

Figure 5.3(a) shows a model which contains a thin protrusion to which a rectangular pocket is attached. The position of the pocket is specified, in this case, by distance constraints from one corner point of the pocket relative to two faces of the protrusion. An angle constraint is used to specify the orientation of the pocket entrance face about the protrusion top face normal. Other available

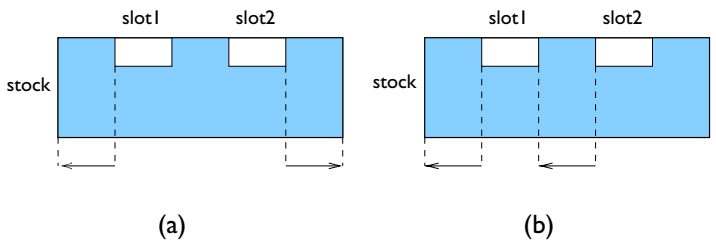


Figure 5.4: Slot2 is positioned relative to its parent feature, a stock (a). Model validation constraints are used to position slot2 relative to slot1 (b). In this way, the dimensioning scheme of the model is changed.

geometric constraints specify, for example, coplanarity and parallelism between faces. In the model of Figure 5.3(a), it is convenient to specify the pocket location relative to the protrusion, which is the parent feature of the pocket; if the location of the protrusion is altered, the pocket will move accordingly (b). In general, however, it is not required to locate a feature with respect to its parent features only, i.e. geometric and attach constraints can be used independently of each other.

Algebraic constraints specify equations containing feature and constraint parameters. They can be used, for example, to specify a certain ratio between length and width of a pocket. Also, algebraic constraints are used to define feature parameters in terms of shape parameters. For example, for a stepped hole feature, a total depth parameter can be defined as the sum of the depths of the comprising cylinders.

Dimension constraints specify an interval for the value of a feature parameter. An example of a dimension constraint in a manufacturing view, is a constraint on the width of a slot, declaring it to be within some specified range, because slots with smaller or larger widths cannot be manufactured. In this way, several slot classes can be specified, each derived from the same slot base class.

Constraints of the presented types can be used as validation constraints for feature classes. Such *feature validation constraints* are embedded as attributes of feature classes. Furthermore, geometric and algebraic validation constraints may be defined between feature instances. In this case, we call them *model validation constraints*. They are specified as constraint objects that are not embedded.

With geometric model validation constraints the position or orientation of a feature instance is specified, possibly in a different way than specified in the

feature class. This can be used, for example, in the model of Figure 5.3(a); after instantiation of the pocket relative to the protrusion, geometric model validation constraints may be instantiated that specify the location of the pocket relative to the base block. Thus, the pocket will no longer move in accordance with the protrusion.

Figure 5.4 on the page before shows how model validation constraints are used to change a dimensioning scheme for a model. In (a) two slots are each positioned relative to the side of a stock. With model validation constraints, in (b) slot2 is positioned relative to slot1, giving a different dimensioning scheme.

5.2 Feature and constraint specification

Feature classes can be constructed using the feature constituents. A feature class can embed them as attributes, or can inherit them from a base class, see for example Figure 5.5 on the facing page, containing LOOKS code. The class RoundedPocket inherits the subtractive nature constituent. Furthermore, it inherits from class Compound, which contains a list of shapes. The feature shape for class RoundedPocket is composed from a block shape and two cylindrical shapes, which are declared as attributes of the feature. Figure 5.6 on page 66 shows how the shapes are combined. Other attributes are constraints, which specify its validity conditions. A Contact constraint is used for attachment of a RoundedPocket instance to the model. Geometric constraints specify coplanarity of the planar cylinder faces to the top and bottom of the block; in this way, the cylinder height is dependent on the block height. Other geometric constraints specify the position of the cylinders by a distance relative to the block; the values of the distance constraint parameters are defined by algebraic constraints. A new feature parameter is defined for the total length of the RoundedPocket, using an algebraic constraint to define it in terms of the shape parameters. On the width and the total length of the pocket a Dimension constraint is specified. For access to selected private parameters of the feature shapes, attributes of type UserDefinedParameter may be added to the RoundedPocket class; they are connected to the private shape parameters with algebraic constraints of type Equal.

Every feature has an initialization method, which takes as arguments feature parameter values, the faces to which the feature is to be attached, and the reference faces used for specifying its location. In the implementation of this method, here omitted, attributes are specified as constraint variables for the validation constraints. Also, constraint parameters are set; for example, for an OnBoundary constraint the extent is specified, i.e. partly or completely. Multiple initialize methods may be defined, to provide different ways of locating the feature with respect to the model.

```
class RoundedPocket inherit Subtractive, Compound
private
   // feature shapes
   b1:Block;
   c1:Cylinder;
   c2:Cylinder;

   // attach b1 to external face
   attachTop : Contact;

   // locate c1 & c2 relative to b1
   coPlanTopC1, coPlanBotC1, coPlanTopC2, coPlanBotC2 : CoPlanarEqual;
   topDist1C1, topDist2C1                             : DistPointFace;
   topDist1C2, topDist2C2                             : DistPointFace;

   // algebraic constraints to define radius of c1 & c2 as half of b1.width
   ratio1, ratio2: Division;

   // algebraic constraint totalLength = c1.radius + b1.length + c2.radius
   plus:Plus3;

   // algebraic constraints to set relative position of c1 & c2
   pos1, pos2     : UserDefinedParameter;
   ratio3, ratio4 : Division;

   // semantic constraints
   semBottomB1, semBottomC1, semBottomC2 : OnBoundary;
   volB1, volC1, volC2                   : NoVolumetricIntersection;

   // dimension constraints
   dimensionWidth       : Dimension
   dimensionTotalLength : Dimension;
public
   // positioning constraints for feature
   topDist1, topDist2 : DistPointFace;
   topAngle           : AngleFaceFace;

   // user defined feature parameter
   totalLength : UserDefinedParameter;

   // initialization method
   Init(topAttach,distFace1,distFace2:Face; p1,p2,alpha,l,w,d:real):void;
implementation
endclass;
```

Figure 5.5: Rounded pocket feature class specification.

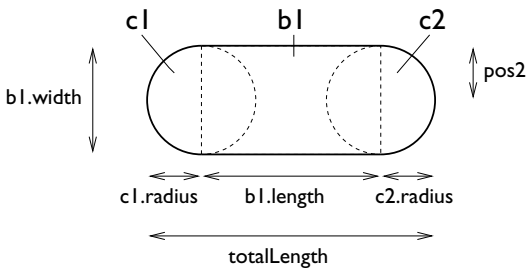


Figure 5.6: A RoundedPocket feature comprises a block shape and two cylindrical shapes, which are interrelated by geometric and algebraic constraints.

```
class Plus3
private
      plusA:Plus;
      plusB:Plus;

      temp: UserDefinedParameter;

public
      Init(a,b,c,d: ValueVariable):void;
      Init(a,b,c: ValueVariable; d:real):void;

implementation

endclass;
```

Figure 5.7: Compound algebraic constraint class specification.

Inheritance can be used to create a hierarchy of generic features. In each class, additional feature validation constraints can be declared. It is possible to modify parameters of inherited constraints, e.g. to strengthen a semantic constraint from partly to completely.

New constraint classes are specified in a similar way, see for example Figure 5.7. An algebraic Plus3 specifies the sum of three parameters to be equal to a fourth parameter, or to a real value used as a constant. The Plus3 constraint has two initialization methods, one method taking arguments of class ValueVariable only, the other taking a fourth argument of type real. Class ValueVariable is a base class both for a feature shape Parameter class and for a UserDefinedParameter class.

The class Plus3 has two private constraint attributes of type Plus, which are connected by a private UserDefinedParameter. These two constraint attributes are initialized in the implementation of the Plus3 initialization methods. The first constraint attribute specifies that the sum of the ValueVariable arguments a and b is equal to the value of the private attribute temp. The second constraint attribute specifies that the sum of the attribute temp and the third argument c is equal to the fourth argument d.

The Plus3 constraint combines two algebraic constraints into one. It is an example of a *higher-level*, or compound, constraint. Higher-level constraints can also combine constraints of different domains. For example, constraints for assembly connections can be specified by a combination of geometric and algebraic constraints. In this way, higher-level compound constraint classes can be easily created from the available predefined simple constraint types.

5.3 Link constraints

When all views of a product are consistent, edit actions may be done in the edit view. One of those edit actions is a feature parameter change, see for example Figure 5.8 on the next page. Two consistent views are shown of a product. If a dimension of the protrusion in the design view is changed, the change is propagated to the manufacturing view. In the latter view, the appropriate dimension of the step is automatically updated.

During the parameter propagation process, the features in all views are validated. Here, we assume that no constraint conflict occurs; the next section elaborates how such conflicts are detected and resolved.

To propagate parameter changes from the edit view to the other views, the views have to be connected. This could be done by coupling feature parameters of different views with algebraic constraints; such an approach would require automatic derivation of these parameter relations from the product model. The main disadvantage is that it is far from trivial to determine a consistent and well-specified set of parameter relations. Therefore, we take a different approach that does not use an explicit coupling of feature parameters of different views.

In our approach, the views are linked on the basis of the geometry. The constraint subgraphs of different views are connected by inter-view constraints called *links*. A link restricts degrees of freedom of a feature element of another view with respect to a feature element of the edit view, and is thus unidirectional. In this way, each of the other views is linked to the edit view. In each of the other views, the feature parameters are determined from the edit view, using the link constraints with the edit view, and the attach constraints in the view.

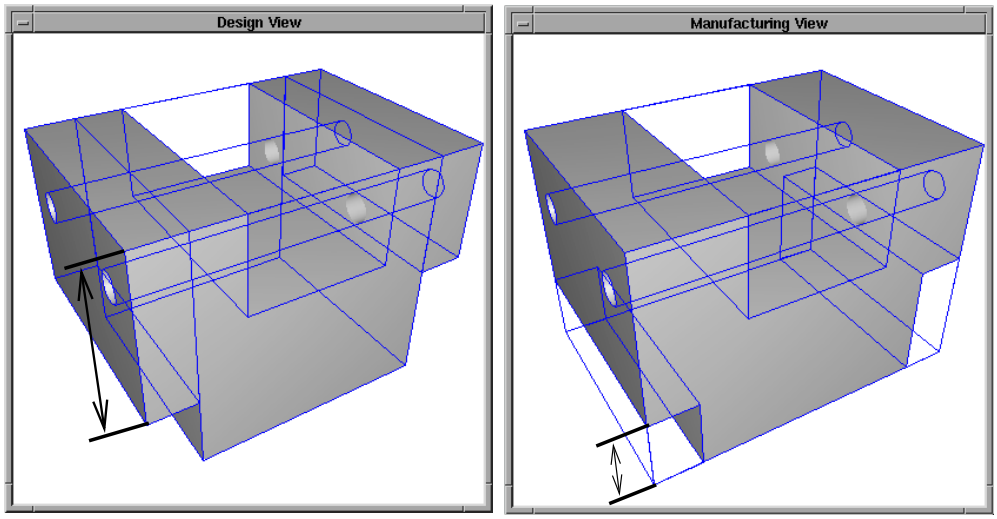


Figure 5.8: Propagation of a parameter change between two views. A change of a protrusion dimension in the design view (a) leads to a change of a step dimension in the manufacturing view (b), and vice versa.

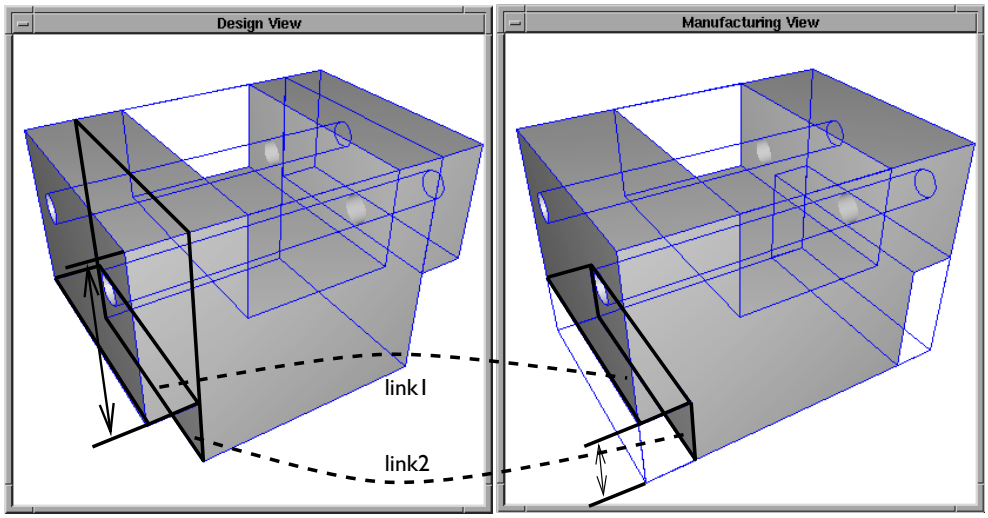


Figure 5.9: Links between two views. The parameters of the step in the manufacturing view are determined using the four attach constraints of the step plus the two links indicated.

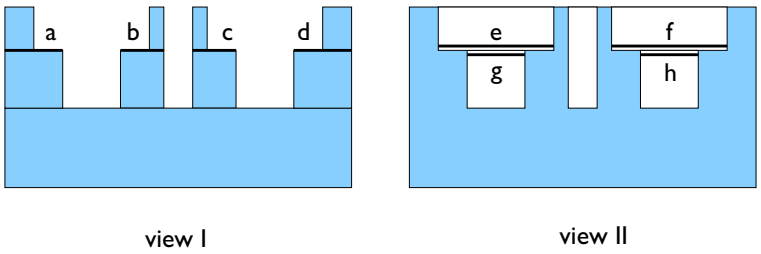


Figure 5.10: Links are created between overlapping on boundary faces.

For example, Figure 5.9 on the facing page depicts the links that are used to propagate the parameter change of Figure 5.8. Among other feature elements, the two step faces indicated are linked to the protrusion face and the base block face indicated. The step is attached on four sides. These attach constraints, plus the two link constraints indicated, fully determine the location of the step faces and thus its location and dimensions. The other features in the manufacturing view are linked in a similar way.

Different views will have feature elements that overlap geometrically. Because the only relationship between consistent views is that they represent the same geometry, feature elements are linked if and only if they overlap and are at least partly on the product boundary. To illustrate the criteria for selecting elements for linking, Figure 5.10 shows two consistent views of a product. In view I, the model consists of a base block with eight protrusions, in view II it consists of a stock with five slots. Let us assume that view II is the edit view, and that we link faces of features only.

The overlap criterion is a necessary condition; a coplanarity criterion only is not sufficient, since it would allow linking a face in the edit view to a face in another view that just happens to be coplanar with it, which would introduce a too strong coupling between the views. In Figure 5.10, for partly on boundary faces e and f, candidate faces for linking in view I are the faces that are coplanar with e and f, i.e. a, b, c and d. If e and f would be linked to all four of them, this would limit the editing possibilities in view II, because it would introduce an undesired equality of the depths of the two upper slots in view II. However, linking face e to a and b, and face f to c and d does give the desirable behavior.

Faces are only linked if they are partly on the product boundary. To determine the parameter values in a view, the link constraints on the linked faces, together with the attach constraints on the other faces in the views are used. We might want to use link constraints only, and not consider the attach constraints at all. This would require linking not only of on boundary faces, but also of not on boundary faces. In Figure 5.10, faces g and h are such faces. Candidate faces for

linking in view I, based on coplanarity only, are, again, faces a, b, c and d. Linking g and h to all four of them would introduce a similar undesirable coupling as mentioned before. However, no faces of view I are overlapping with g and h, and thus cannot be linked. Therefore, for the faces that are not on the boundary, the relations that are specified by the attach constraints in each view are required to determine the feature parameters in the non-edit views from the edit view.

A link constraint for two planar faces specifies coplanarity with equal or opposite orientation for the face normals, depending on the nature of the features; it thus leaves two translational and one rotational degree of freedom between the linked faces. A link constraint between two cylindrical faces specifies that the cylinder axes induced by the cylindrical faces must be equal; it thus leaves one translational degree of freedom between the linked faces. The degrees of freedom that are left are specified by attach and geometric constraints.

Links are automatically created by the modeling system when all views are consistent, after a view has been chosen for editing. For this, the product geometry in the cellular model is queried for overlapping feature faces that are on the product boundary [De Kraker et al. 1997*b*].

5.4 Constraint conflicts

Edit operations are propagated from the edit view to the other views using constraint techniques. In this process, conflicts may arise between views. To resolve these, each view is assigned a priority that is used to decide whether the edit operation will be carried out and propagated. If propagation is successful, the feature models in the other views are adjusted accordingly, and thus the edit view determines the feature models in the other views. A distinction is made between two types of edit operations: changing a feature parameter, and adding or removing a feature.

When a feature parameter is changed in the edit view, this change is tried to be propagated to the other views. In the propagation process, one of three situations occurs:

1. No conflicts are caused, and the parameter change is propagated to the other views using link constraints.

2. The parameter change in the edit view causes a conflict with a constraint in another view.

3. The parameter change in the edit view causes a conflict in the sense that another view gets a different topology.

An example of situation 1 was shown in Figure 5.8 on page 68. Both conflict situations (2 and 3) are detected using the link constraints with the edit view and the validation constraints in the other view. Conflicts are resolved by reopening the view that conflicts with the edit view, which is either successful or not, and by using the priorities specified for the views [De Kraker et al. 1997*b*].

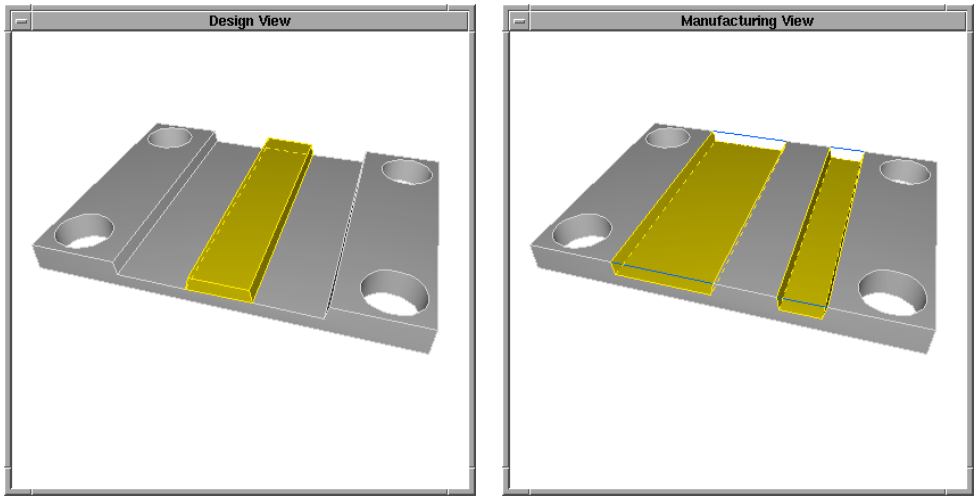
When a feature is added or removed, this always causes a conflict, since the product topology is changed from the edit view, and the other views still represent the original product topology. These edit operations are propagated to the other views in the same way as in the conflict situations above, by reopening the other views.

Reopening of a view is performed incrementally. Using the previous interpretation as a starting point, possibly some features are removed, and, possibly, new features are added. The new feature instances are found by reasoning about the cellular model geometry. Model validation constraints on the removed feature instances are also removed; if required, feature class specifications can be extended with these validation constraints, to ensure their automatic instantiation in the reopen process. In this way, a new interpretation of the edited product in terms of features is determined [De Kraker 1997].

If reopening is successful, and thus a feature model for the reopened view that is consistent with the edit view has been obtained, the edit operation has successfully been propagated. Reopening may also be unsuccessful. In this case, a valid feature model for the reopened view does not exist. For a manufacturing view, for example, this means that the product cannot be manufactured with the given feature library. In this case, another type of manufacturing view may be tried.

If reopening a view is not successful, and that view is of a higher priority than the edit view, then the edit operation is canceled. In this way, an allowed edit operation in a lower-priority view, for example a manufacturing view, can be blocked by a higher-priority view, for example a design view. If the unsuccessfully reopened view is of a lower priority than the edit view, then the conflict is presented to the users of the conflicting views. The edit operation may be canceled, or it may be confirmed, after which the edit operation is carried out and the conflicting view is discarded until it is reopened explicitly by a user. In this way, views that can no longer be maintained are discarded, because there is no feature model that represents the product.

Some examples are now given to clarify the conflict detection and handling. Figure 5.11 on the following page depicts two consistent views of a part which has a wide slot in which a rib was placed in the design view (a), and two smaller slots in the manufacturing view. In the manufacturing view, the user has added an algebraic constraint declaring the slots to have equal widths, since this facilitates



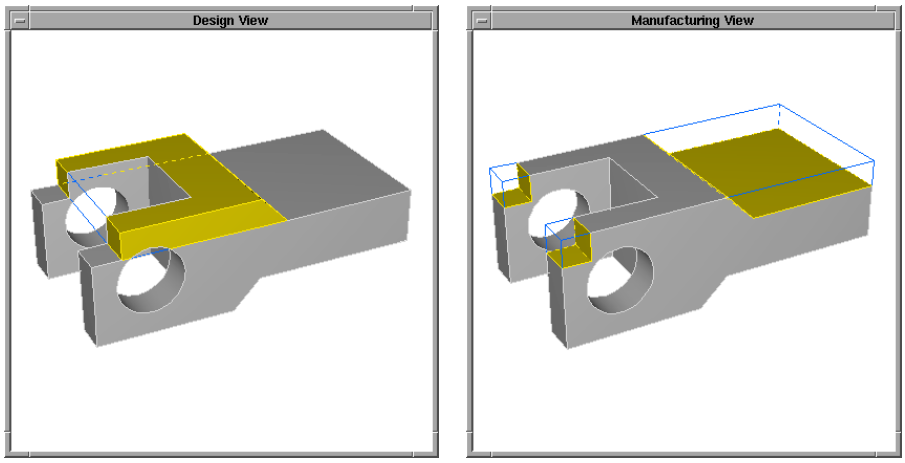
(a) (b)

Figure 5.11: Changing the rib position in the design view (a) violates an equal slot widths constraint in the manufacturing view. The manufacturing view is reopened, resulting in the model shown in (b).

cheaper manufacturing of the part. If the designer changes the position of the rib in the design view, this change will cause a violation of the algebraic constraint in the manufacturing view. If we assume that the design view has a higher priority than the manufacturing view, the latter view will be reopened, and the algebraic model validation constraint is removed, and two slots with a different widths are instantiated (b).

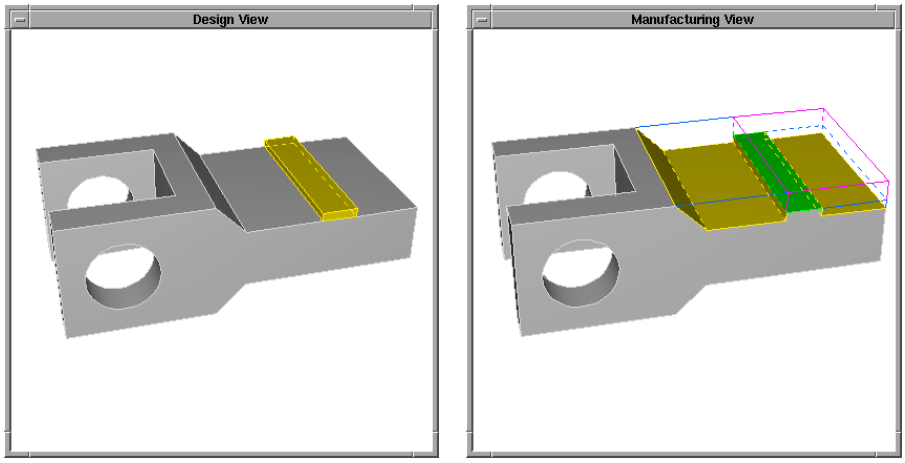
Figure 5.12(a) depicts a feature model in the design view of the part already encountered in Chapter 4. The user has repositioned the upper rib (a). This causes a change in the product topology, which is detected as a conflict between two link constraints: the left rib face and the left base block face are each linked to the same face of the block stock in the manufacturing feature model, but they are no longer coplanar. The manufacturing view is reopened, leading to the addition of two step features and a dimension change of the upper step (b).

Figure 5.13 shows how a feature addition is propagated. In the upper step of the previous feature model a protrusion is instantiated (a). Again, this causes a change of the product topology. After the manufacturing view has been reopened successfully, the step has been replaced by a slot and two steps on top of each other (b).



(a) (b)

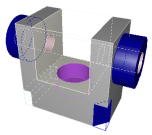
Figure 5.12: Changing the rib position in the design view (a) causes a change of product topology, detected as a conflict between link constraints. The manufacturing view is reopened and two steps are instantiated (b).



(a) (b)

Figure 5.13: Adding a rib in the design view (a) causes a change of product topology. The manufacturing view is reopened, leading to instantiation of a slot and two steps on top of each other (b).

Part III Constraint satisfaction



6

Constraint management

Constraint management is maintaining the set of constraint instances that result from modeling actions. This chapter first gives an overview of how the constraint management is done. Then, it elaborates the constraint graph data structure used for storage of the constraint instances. Next, it explains how the constraints of different types are maintained, and then, it presents some optimizations of the constraint maintenance scheme. Finally, the constraint management approach is viewed from a broader perspective.

6.1 Overview

Constraint management is done when a user in one of the views performs a feature model edit action. The Constraint Manager is responsible for maintaining the validity of the feature models of the different views. It has two main tasks. Firstly, it stores the set of constraints that result from all modeling actions, and it can be queried about this set of constraints. Secondly, it maintains the validity conditions specified by these constraints.

Figure 6.1 on the next page depicts an overview of the constraint management. All constraints specified in the views are stored in a constraint graph data structure. The constraint graph provides efficient functions for updating and querying. As was explained in Chapter 4, the validation constraints are specified in LOOKS. Their C++ counterparts, or *constraint instances*, are stored in the constraint graph. The constraint graph contains a symbolic representation of the product in terms of constraints specified by feature models in the different views.

The validity conditions specified by the constraints are maintained by alternatingly deploying specialized constraint solvers. Each of them receives the constraints to be solved in a *primitive constraint graph* that is a *map* of the main constraint graph. This map is maintained by the Constraint Manager. In the map maintenance, primitive constraints are added or removed from the primi-

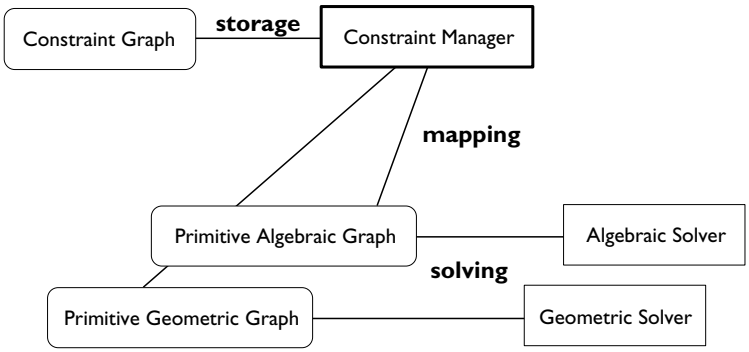


Figure 6.1: Overview of constraint management.

tive graphs, and primitive constraint parameters are changed. More details of the mapping process are provided in Chapters 7 and 8.

After a primitive constraint graph has been solved, the constraint manager uses the results to update the main constraint graph. The other primitive constraint graph is updated accordingly. In this way, the interdependence of the constraints is handled by the Constraint Manager, while the primitive graphs are solved efficiently.

The relation between shape geometry and shape parameters is specified in the constraint graph by *shape constraints*. A shape constraint controls how the shape changes when a shape parameter changes, and vice versa. A number of shape constraint types are available which each corresponds to a shape type. This use of shape constraints has the advantage that feature shape validity is preserved in the constraint management. A shape constraint is instantiated automatically when a feature shape is instantiated; therefore, a user making a feature class specification does not have to include many geometric constraints.

A distinction is made between *explicit* and *implicit* shape parameters. A shape parameter is called explicit if its value is specified by an algebraic expression, e.g. a value. It is called implicit if its value is not specified in this way, but is determined by means of attach and geometric constraints. For example, a blind hole feature and a through-hole feature both have a cylindrical shape. For the blind hole, the cylinder height parameter is explicit, but for the through hole it is implicit and determined by the attach constraints of the cylinder top and bottom faces. For

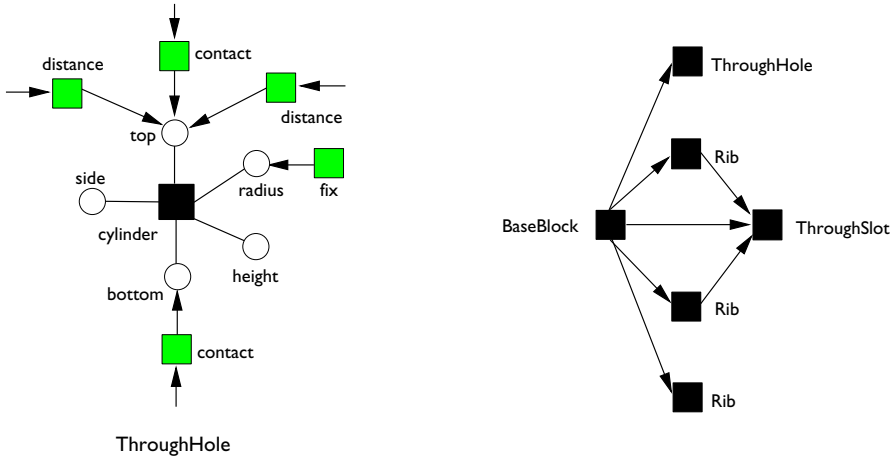


Figure 6.2: Constraint graph for a through-hole feature (a), and shape graph for the design view (b).

each shape parameter that is explicit for a feature class, a *fix value* constraint is declared in the feature class specification.

For specification of the position and orientation of the part being modeled, a *fix global location* constraint is available. It fixes the position and orientation of a feature element with respect to the world coordinate system. Typically, base features are fixed in this way, whereas all other features in the part are located relative to feature elements.

6.2 Constraint graph

All specified constraint instances from all views are stored in a bipartite constraint graph, in which two types of nodes represent constraints and variables, and edges connect constraints to associated variables [Dohmen et al. 1996]. Constraint variables are feature elements, shape parameters and user-defined parameters. Refer to Section 2.1 for an overview of constraint representations.

Hierarchical relations are represented by unidirectional constraints, nonhierarchical relations by multidirectional constraints. Multidirectional constraints have a direction that may vary, depending on the direction of connected con-

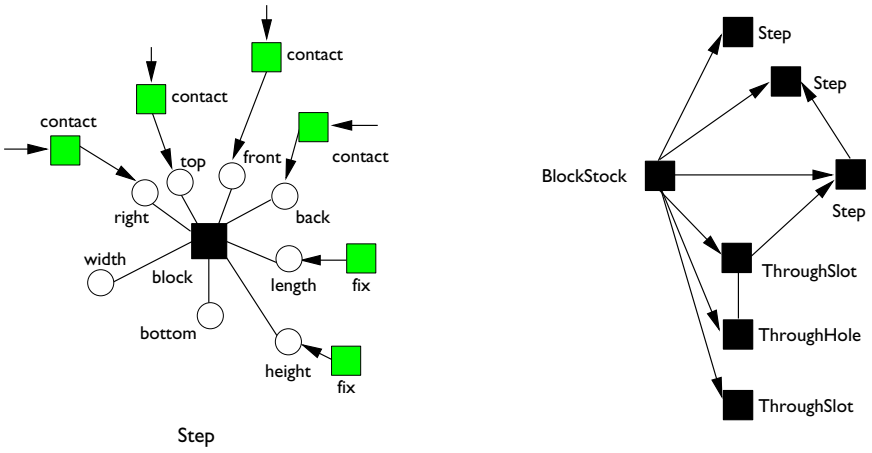


Figure 6.3: Constraint graph for a step feature (a), and shape graph for the manufacturing view (b).

straints. The direction of a constraint is stored in the constraint graph edges that connect the constraint with its variables. Feature validation constraint instances between shapes of different feature instances, such as attach constraints, are unidirected. Model validation constraint instances, and feature validation constraint instances between shapes of a compound feature, are all multidirected.

The constraint graph consists of subgraphs, one for each view. In a constraint subgraph, the main entities are the shape constraints. A shape constraint connects the elements and shape parameters of a feature shape. A fix value constraint is connected to one shape parameter only. Feature elements are connected to elements of other features with unidirected attach and geometric constraints.

In Figure 5.13 on page 73, two views of a product were shown. These are now used to illustrate the structure of the constraint graph. In the design view, the model consists of a base block to which two ribs with a slanted face, a through-slot, a through-hole and a third rib have been attached. Figure 6.2(a) shows the complete constraint graph for a through-hole feature. Figure 6.2(b) shows a simplified constraint graph for the whole design feature model in which the shapes are depicted as nodes, and multiple constraints are depicted as a single, possibly directed, edge. In shape graph drawings, edges that represent

only multidirectional constraints are depicted undirected. Such a *shape graph* is used throughout this chapter to show the structure of the constraint graph for a feature model. In the model of Figure 6.2(b), the through-slot is attached to two ribs, and positioned relative to the base; thus, the through-slot parameters are determined by the base and the two ribs, which is depicted as three incoming directed edges for the through-slot.

In the manufacturing view, the feature model consists of a block stock to which three steps, two through-slots and a through-hole have been added. Assume that, in this example, after instantiation of the through-slot with the slanted face, the user has instantiated a model validation constraint to specify a relative position between this slot and the axis of the through-hole. Figure 6.3 shows the complete constraint graph for a step feature, and the shape graph for the manufacturing feature model. In the shape graph, the model validation constraint is shown as an undirected edge, to stress that a change of position of the slanted through slot influences the position of the through-hole, and vice versa.

The two views are consistent. If, for example, the design view is chosen for editing, link constraint instances are created which are directed from elements in the design view to elements in the manufacturing view. Link constraints are stored in the constraint graph in the same way as the other constraint instances. In this way, the constraint subgraphs of all views are connected.

The constraint graph has been implemented in C++ as a container data structure, i.e. it stores pointers to constraints and constraint variables that are externally created objects. The constraint graph class is derived from the graph template class in the LEDA class library, which provides the basic node and edge objects in a double-linked structure. Figure 6.4 on the next page depicts the different data structure entities. The arrows in the picture denote pointers to objects. Nodes contain a pointer to either a constraint or a constraint variable, whereas edges contain a pointer to edge direction information. The constraint graph class provides methods for addition, removal, modification and querying of the data.

To go from a constraint to its variables, or vice versa, the node-edge linked list is used of the graph containing the object. A constraint does not contain direct pointers to its variables, nor vice versa. This ensures that modification of the constraint graph topology is handled in a safe way. Such modifications occur, for example, when a feature instance is positioned relative to other features than at instantiation time. To prevent expensive graph searching for a particular constraint or variable, two sorted index arrays provide fast random access to the data in the graph; the arrays use a unique identifier for each constraint and each variable.

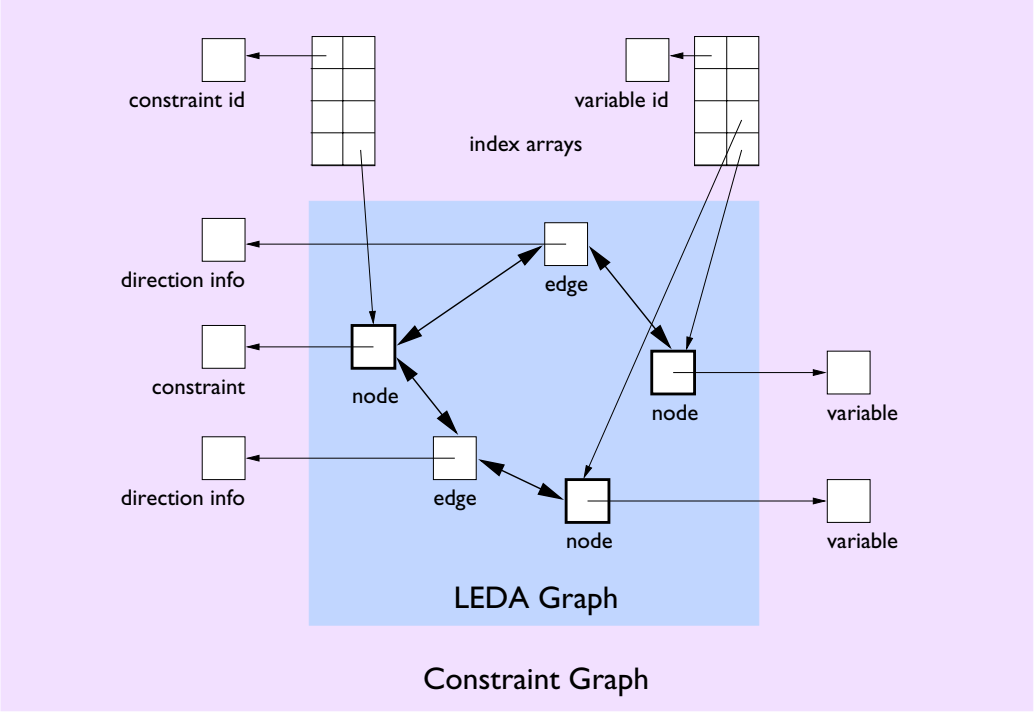


Figure 6.4: Constraint graph data structure.

6.3 Constraint maintenance

When a feature model is edited in one view, constraint maintenance is done in order to maintain feature model validity in all views. This section first presents the general constraint maintenance scheme for one view, and next presents an extended scheme that includes constraint maintenance across multiple views.

Single view

Constraint maintenance is done by performing several actions consecutively. Figure 6.5 on the facing page depicts the main steps in this process.

An edit action implies changes in the main constraint graph, which can be either changes in its topology, or changes in the values of some constraint variables. The first step in the constraint maintenance process is updating the map of the algebraic constraints, which involves mapping newly added algebraic constraints, and passing changed feature parameter values to the algebraic constraint graph.

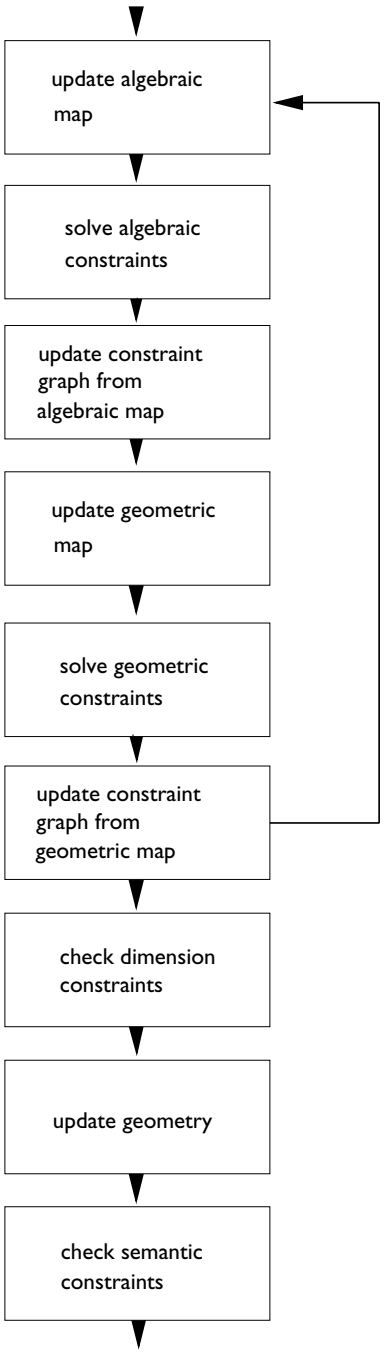


Figure 6.5: Overview of constraint maintenance process.

This is followed by calling the algebraic solver to solve the algebraic constraint graph. In order to determine the parameter values, the values in the solved primitive algebraic constraint graph are mapped back to the main constraint graph. Next, to reflect the solved parameter values in the geometric constraint graph, and to account for newly added geometric constraints, the map to the geometric constraint graph is updated. Following, the geometric solver is called, and the results of this solving action are mapped back to the main constraint graph in order to update the location of the feature elements. From the geometric configuration of the feature shape elements, all feature shape parameters can then be derived.

Now, the feature parameter values are checked against the dimension constraints. Then, the product geometry, i.e. the cellular model, is updated, which enables finally checking the semantic constraints. If a conflict is detected during solving or checking of constraints, the constraint maintenance process is stopped, and the conflict is handled, as described in Section 5.4.

Algebraic constraints are specified for feature parameters, which are either shape parameters or user-defined parameters. Algebraic constraints can only specify values for explicit shape parameters; for implicit parameters, a value determined from an algebraic constraint would always be overruled by the geometric solver. However, implicit shape parameters can be used as input values in algebraic constraints, by which other parameters are determined. For example, in an assembly view, the length of a pin in one part could be declared to be equal to the depth of a through-hole in another part. The Constraint Manager handles this situation in two iterations. Firstly, it solves the algebraic constraints that do not involve implicit parameters. Then, it solves the geometric constraints, after which the implicit parameters are known. Then, the algebraic constraints on these implicit parameters are included in a second algebraic solving step. This is depicted as the backward arrow in Figure 6.5. This can cause a value change of other, explicit parameters, and therefore the geometric constraints are re-solved.

Two iterations are sufficient, since we assume that the constraints in the primitive geometric graph fully specify the model geometry. After algebraic solving, the feature parameter values are reflected in the geometric graph; the parameter values are thus used as inputs for the geometric solver. Conflicts between constraints in the algebraic graph and constraints in the geometric graph are detected as conflicts between primitive geometric constraints by the geometric solver. Chapter 8 elaborates this.

Semantic constraints are checked using queries to the cellular model [Bidarra, De Kraker and Bronsvoort 1997]. For a semantic constraint on a shape face, the boundary presence of the cellular faces that together represent the shape face is

considered. For volumetric semantic constraints, the natures of the volumetric cells shared by intersecting feature shapes are considered. This functionality is provided in SPIFF by the Feature Geometry Manager.

Multiple views

For multiple views, the constraint maintenance scheme has been extended to establish and maintain a geometry-based coupling between the edit view and the other views. Section 5.3 presented link constraints, which provide a coupling between elements of features of different, consistent, views. The link constraints are used to propagate feature parameter changes, which is done in several steps.

The view that is chosen for editing determines the feature parameter values in the other views, some directly via link constraints, others indirectly via link constraints and attach constraints. Therefore, in the non-edit views, all shape parameters are implicit. In each non-edit view all constraints are de-activated that would determine feature parameter values which are to be determined directly by the link constraints; these constraints need not be considered in the constraint maintenance scheme until this view is chosen for editing, and they are re-activated by the Constraint Manager. The de-activated constraints are of the algebraic, geometric and fix value type. The attach and shape constraints remain active.

Next, the feature geometry manager is queried for feature elements of non-edit views that are part of the product boundary, and overlap with feature elements in the edit view. Between these pairs of overlapping feature elements, link constraints are established, directed from the edit view to each of the non-edit views.

Next, the map to the geometric constraint graph is updated. The de-activated geometric constraints are removed from the geometric graph. The link constraints are also mapped to the geometric graph. The algebraic and geometric graph together now contain a symbolic representation of the product geometry as specified in the edit view, with each non-edit view determined by the edit view.

Now, the constraint maintenance process is continued by calling the dedicated solvers. After solving, the parameter changes have been propagated to the non-edit views, and the algebraic validation constraints in the non-edit views are checked for violation. Then, the dimension and semantic constraints are checked.

When changes are tried to be propagated, conflicts may be detected, and some views may need to be reopened, as was explained in Section 5.4. The view and feature geometry managers perform the interpretation of the new product geometry in terms of the features in the particular view. In the interpretation pro-

cess, feature attachments relations and feature dimension values are determined; the location of the features is determined in global space.

The interpretation process has been completed, when the location parameters of the features with respect to other features have been determined. This requires that for each feature, reference model faces are determined relative to which the feature is to be located. For this purpose, information is used of the feature class, such as its shape type, the faces used for positioning and the attach faces. For each feature, a set of model faces is determined that can serve as location reference faces. From this set, a suitable subset of model faces is selected. This selection can be based on different criteria; in the implementation of SPIFF, the model faces that have the smallest distance to the feature are selected. After the view has been reopened, the engineer may choose other reference faces if desired.

6.4 Optimizations

The presented general scheme for maintaining the constraints of the different types is fine-tuned to the characteristics of the different edit actions. In the remainder of this thesis, we will use the term *geometry-related constraints* to refer collectively to shape, attach, geometric and link constraints.

When a feature is added, only the location and dimensions of the shapes comprising the new feature have to be determined. The algebraic and geometry-related constraints of the previous feature model are not violated by the addition of the new feature. Therefore, only the new algebraic and geometric constraints are solved. After all parameters of the new feature have been determined and checked against its dimension constraints, the new feature shapes are inserted into the feature model geometry. Then, the semantic constraints are checked of the new feature and all features it intersects.

When a feature is removed, the Constraint Manager checks whether there are features that are dependent on the feature to be removed. This is done by considering the direction of the constraints of this feature. If there are constraints directed from this feature towards other features, it has dependent features and therefore, it is not removed. Constraints directed towards this feature, and multidirected constraints with other features are removed automatically together with the feature. When the feature shapes have been removed from the cellular model, the semantic constraints are checked of all its previously intersecting features.

For feature parameter changes, the general constraint maintenance scheme can be optimized by considering only the features that directly or indirectly are changed. After the primitive algebraic and geometric graphs have been solved,

the product geometry has been determined. Due to the parameter changes, some features have changed location or dimensions. Constraint checking is only required for the algebraic, dimension and semantic constraints for these changed features, together with the semantic constraints of any feature that intersected those features before or after the change.

These optimizations require computation of the set of semantic constraints that need to be checked. This is done using the cellular model to determine the set of features that intersect a particular feature. The amount of time spent to find the intersecting features is roughly equivalent to the time spent on checking its semantic constraints. Therefore, in worst cases, i.e. if the complexity of the model is high in terms of feature intersections, it does not really pay to determine this set of intersecting features, but it would be cheaper to simply check all semantic constraints. However, for feature interaction validation [Bidarra, Dohmen and Bronsvoort 1997], the time required for interaction checking is large compared to the time required to determine the set of intersection features; therefore, in this case, it always pays.

Another optimization involves pruning the constraint graph in order to solve only a part of it. This pruning concerns the geometry-related constraints only. Consider a product on which a number of consistent views exist. Let us assume that one or more parameters have been changed in the edit view. Furthermore, let us assume that the algebraic constraints have already been solved, and that therefore all explicit feature parameters are known in the edit view. Now, before the geometry-related constraints are solved, the main constraint graph is analyzed to determine subgraphs that will not be disturbed by the parameter changes. In such subgraphs, the shapes have a fixed relative position and fixed dimensions; therefore, we call them *rigid subgraphs*.

For example, consider the model of Figure 6.6(a) on the next page. In Figure 6.6(b) the corresponding shape graph is depicted. If the depth of the blind hole is changed, only the subgraph of the constraint graph containing the blind hole has to be solved. However, if the depth of the through-slot is changed, then a larger subgraph consisting of the through-slot and the blind hole has to be solved. In the latter case, the geometric solver regards the blind hole as a rigid body, since only its location in global space may change, but not its dimensions.

A pruning algorithm has been implemented that operates on the shape graph for a model; it performs a modified breadth-first graph search. The shape graph is traversed going from a shape constraint node to a connected successor shape constraint node, following the direction of the edges. The goal of the algorithm is to give each shape constraint a *group number* that indicates the rigid subgraph it is part of. Shape constraint nodes with equal group numbers are in the same rigid subgraph. The group number -1 is reserved to indicate that the shape constraint

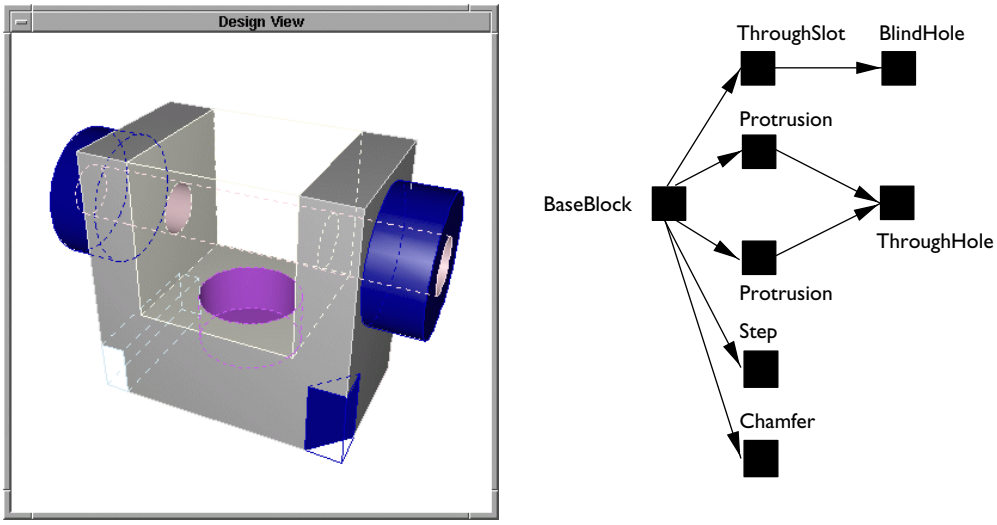


Figure 6.6: A design view of a part (a) and the corresponding constraint graph (b).

node is not part of a rigid subgraph, i.e. that it is nonrigid. Group number 0 indicates a rigid subgraph which is fixed in global space. After the group numbers of all shape constraint nodes have been determined, they are accounted for when the constraint manager updates the map of the main constraint graph to the primitive geometric graph. More details on this are provided in Chapter 8.

In the shape graph, there are one or more shape constraints that each have a fixed global location; typically, these shape constraints correspond to base features. These shape constraint nodes are called *root nodes* of the graph. From the root nodes, all shape constraints can be reached that correspond to the features that comprise a model part. The shape constraint nodes which are reachable from the root nodes of a model part together form a connected shape graph component. Traversing the shape graph from each of the root nodes, the pruning algorithm determines the connected components that each completely form a rigid subgraph, because they represent a model part that does not contain shape constraints with changed parameters. These rigid subgraphs are fixed in global space, and, therefore, are assigned the group number 0.

While traversing the shape graph, the group number for the currently reached shape constraint node is determined by looking back and considering the group numbers of its preceding shape constraint nodes. For a shape constraint node,

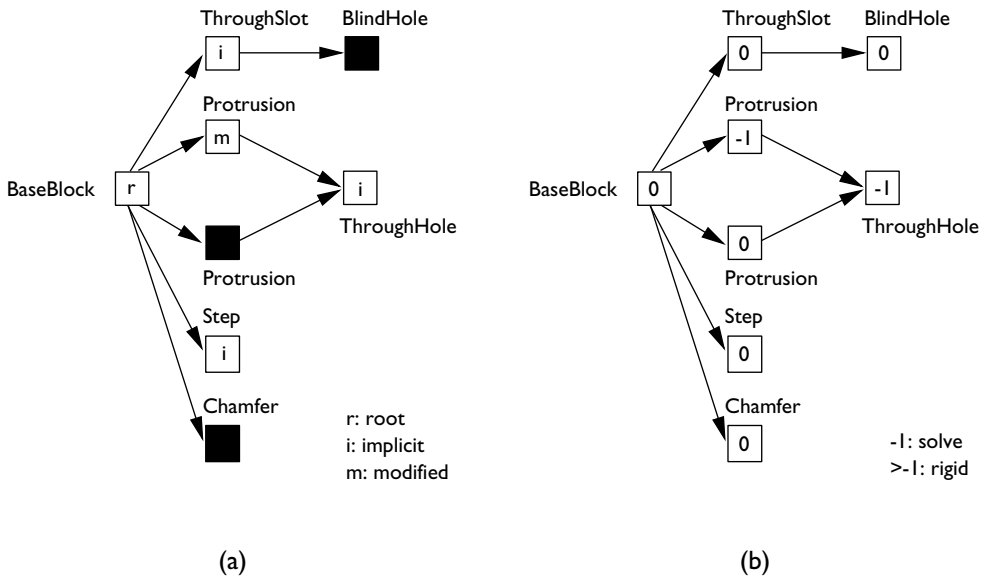


Figure 6.7: Shape graph before (a) and after pruning (b), if a protrusion is modified.

the preceding nodes are its adjacent nodes connected with undirected edges or with edges directed towards it. If all preceding nodes are in the same rigid subgraph, i.e. they have the same nonnegative group number, the current node is also assigned this group number. If not all preceding nodes have an equal nonnegative group number, a distinction is made between current shape constraints with implicit parameters and those without: a shape constraint node with implicit parameters is marked as not belonging to a rigid subgraph, whereas a shape constraint node without implicit parameters is assigned a new group number, since it is the root of a new rigid subgraph.

Two examples are given to illustrate the pruning. Figure 6.7 shows the shape graph for the previous model. In this example, one of the cylindrical protrusions is modified by the user. The shape constraints for the through-hole, through-slot and step features have implicit dimensions. Figure 6.7(b) shows the result of the pruning algorithm. The modified protrusion and the attached through-hole need to be solved by the geometric solver.

Figure 6.8 shows the case for modification of the base block. The through-slot, through-hole and step are nonrigid, and have to be solved for location and dimensions; the other shapes are rigid, but their location has to be determined by the geometric solver.

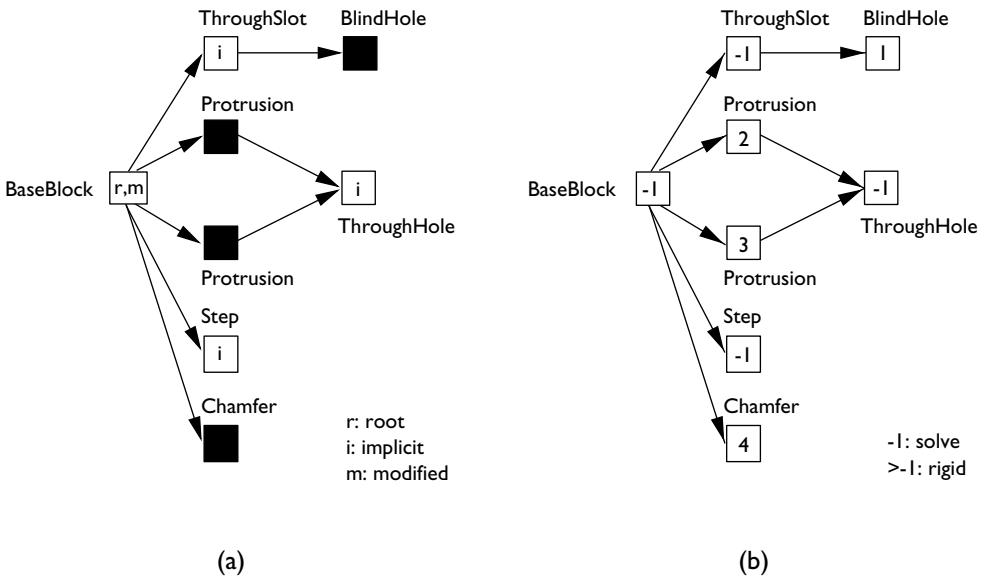


Figure 6.8: Shape graph before (a) and after pruning (b), if the base block is modified.

In the implementation, the shape graph is computed from the main constraint graph, every time the pruning has to be done. In this way, topology changes of the main constraint graph are always reflected in the shape graph.

The analysis of the constraint graph is done solely on the basis of the directions of the connecting edges. No reasoning about degrees of freedom is involved, nor is it considered which dimension is actually modified. It is therefore an inexpensive way to prune part of the constraint graph before solving, but it may not always provide an optimal pruning. For example, consider again Figure 6.7. In this example, one of the cylindrical protrusions was modified, resulting in resolving the attached through-hole. This is correct if only the height of the protrusion is changed; however, if only the radius of the protrusion is changed, the attached through-hole could be, but will not be pruned. To achieve this, the graph analysis would have to be done at the level of degrees of freedom of shape elements, instead of at the level of complete shapes; this would require considerably more computational overhead, for example, to deal with degenerate cases. Fortunately, the suboptimal pruning technique discussed here, in practice gives very satisfactory results.

6.5 Discussion

The constraint management approach presented in this chapter has several interesting aspects. If we limit our scope to a single view, the approach deals with constraints of different types, which interact with each other. Widening our scope to multiple views, we may consider how the views are coupled, and what role the constraints play.

Within one view, constraints of different types interact with each other. The coupling between constraints that specify parameter values and constraints that specify geometry relationships has been addressed often in the past. The first category of constraints is sometimes called functional constraints, or engineering constraints; we will just call them algebraic constraints. The second category are the geometry-related constraints, i.e. the shape, attach, geometric, and link constraints.

A common approach is to express both algebraic and geometry-related constraints uniformly in one domain, usually as coupled equations. Chapter 2 of this thesis discussed several satisfaction techniques for such constraints, such as iterative techniques, graph-based techniques, propagational techniques, and combinations of these. Henderson [1996], for example, uses an approach in which only a single constraint type is available to uniformly represent both algebraic and geometry-related constraints. A constraint of this type is defined by one or more satisfaction procedures. A local propagation algorithm is used to select a procedure for each constraint in order to solve the constraint graph.

The main advantages of a uniform representation are that the coupling between the algebraic and geometry-related constraints is preserved, and that well-known solving techniques are available. However, the representation of the geometric relationships as equations causes a loss of geometric domain-specific information, which may lead to a less efficient and less powerful constraint solving scheme, and which may even result in meaningless models.

A different approach expresses the coupling of the constraints at the constraint solving level, instead of at the constraint representation level. Separate constraint representations are maintained, and are solved with different dedicated constraint solvers. The coupling between the algebraic and geometry-related constraints is accounted for explicitly in the solving process by alternating between the solvers; such approaches differ in the way the alternation process is done.

For example, Hoffmann and Juan-Arinyo [1997] developed an approach that alternatingly solves part of the algebraic graph and part of the geometric constraint graph. Geometric constraints are satisfied by performing geometric construction steps; when values for dimensions are required to perform a geometric

construction step, they are computed from algebraic constraints, or from configurations of geometric elements. Thus, the algebraic and the geometric solver are more tightly coupled.

The approach presented in this thesis has two steps. Firstly, it solves the algebraic constraints, and next the geometry-related constraints. If after that there are unsatisfied algebraic constraints left, the two steps are repeated. In this way, the dedicated solvers can work independently. Such an approach has the advantage of modularity, making it possible to deploy other constraint solvers. A limitation of the current implementation is that a strong coupling between the two constraint graphs is not supported; it is not possible to alternate between the two solvers after having solved a part of a graph. This is due to the assumption that the geometric graph is well-constrained. However, the current approach will be extended to incorporate such a strong coupling.

In all these approaches, only the values of constraint variables are communicated between the different constraint domains. By using information about dimension dependencies in the geometric graph, it may be possible to derive algebraic relationships between dimensions that can be used by the algebraic solver directly. In such a way, not only values, but also structural information about dimension dependencies is communicated; this would give an even more powerful constraint management scheme.

Much research has been done on maintaining simultaneous views, in the context of concurrent engineering. Molina et al. [1995] give a review of concurrent engineering support systems. Most of these systems assume a particular combination of views, and lack a more generic approach.

Related, but different approaches, for example by Henderson [1996], do not actually support different views of a product, but rather have a distributed product model to support collaborative design. In such systems, different parts of a product can be designed in physically different places. Constraints are used to couple parameters explicitly, and to propagate parameter values between the different parts.

Others do use the generic notion of a view of a product, and support different views. For example, Bowen and Bahler [1992] specify views in terms of accessible aspects of objects. They developed the general-purpose constraint programming language Galileo4, in which design decisions are specified as constraints on objects. The different views are not connected by constraints, but instead the objects themselves are shared by the views. There is no representation of the product geometry, nor are geometry-related constraints supported. Conflicts between constraints of different views are detected and explained, thus providing information on the basis of which the engineers of the views involved can negotiate.

Some approaches support coupling of the geometry of the views in some central geometric representation, but do not express this coupling in the constraints, see for instance [Brunetti, De Martino, Elter and Falcidieno 1996]. This limits the feature validation capabilities, since feature model changes cannot be propagated easily between the views.

In the approach presented in this thesis, the coupling between the views is geometry-based, to reflect that all views represent the same product geometry. The constraint variables that correspond to feature elements are not shared, but connected with link constraints in order to couple only the required degrees of freedom of the feature elements. There is no direct coupling between feature parameters of different views, for example by algebraic constraints.

Instead, first the geometry is determined using the algebraic constraints and geometry-related constraints from the edit view. Next, from the geometric configuration of the feature elements, the feature parameter values are determined in the non-edit views. In these views, the validation constraints are checked. If a violation is signaled, the priorities of the involved views are used to decide whether to overrule the violated constraints, or to cancel the modification.

The presented constraint management approach facilitates a powerful scheme for multiple-view feature validation. The link constraint coupling between consistent views is created easily using the cellular geometric model. It enables propagation of changes between the views, validation of the different feature models, and detection of conflicting views.

7

Algebraic constraints

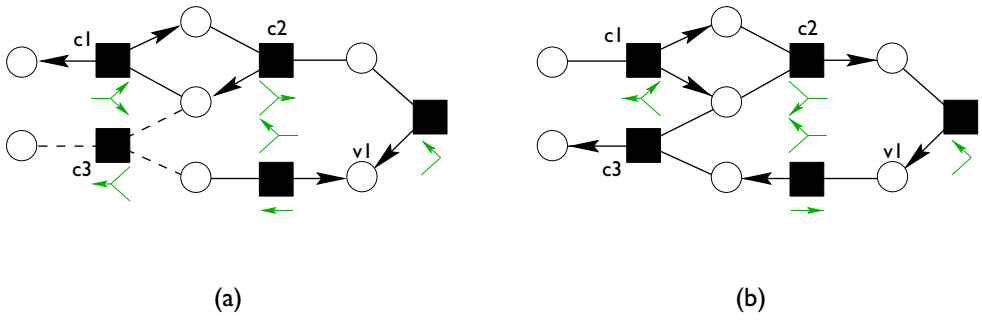
This chapter presents how the algebraic constraints are maintained. First, it presents an overview of the SkyBlue solver that was incorporated as the dedicated primitive algebraic solver. Next, it gives details about the mapping of the algebraic constraints to the primitive algebraic constraint graph, and about how this map is maintained. Then, implementation details are provided. Finally, the chosen approach is evaluated.

7.1 SkyBlue

The public domain SkyBlue solver [Sannella 1992, Sannella 1993] is an example of a local propagation constraint solver; local propagation techniques have been discussed in Section 2.3. SkyBlue has been used, for example, in applications for user interface construction in which 2D geometric and arithmetical algebraic constraints are used.

A SkyBlue constraint is defined by one or more procedures, or *methods*, that each satisfy the constraint. For each method, the variables of the constraints are divided into two groups: the *input variables* and the *output variables*. Each satisfaction method computes the values for the output variables of the constraint on the basis of the values of its input variables. A SkyBlue constraint in general has several, alternative, methods, corresponding to different combinations of input and output variables. A SkyBlue constraint has no inherent direction. SkyBlue does not make any assumptions about the value or type of variables of a constraint. All SkyBlue constraints and variables are stored in a SkyBlue constraint graph.

SkyBlue constraint types have been implemented for the basic arithmetical operations addition, subtraction, multiplication and division. Each of them takes three SkyBlue variables, and has three satisfaction methods that each take a different variable as output variable. Also, a SkyBlue constraint has been implemented for equality of two SkyBlue variables.



(a) (b)

Figure 7.1: Method graph with conflicts (a) and method graph for same constraints without conflicts (b).

The goal of the SkyBlue algorithm is to select a satisfaction method for each constraint; thus, it assigns a direction to each constraint. SkyBlue works in two phases. Firstly, from the SkyBlue graph, the solver constructs a directed *method graph* by setting the direction for each constraint according to its selected method. In this way, a symbolic solution to the SkyBlue constraints is determined. Secondly, each selected constraint method is executed, and thus values are propagated in the constraint graph.

For an example, see Figure 7.1, after [Sannella 1993]. In (a) a method graph for a constraint graph is shown; squares denote constraints, and circles denote variables. For constraints, directed edges indicate the selected method. Below each constraint, its alternative methods are depicted. The method graph contains a conflict for variable v1, because it is determined by two constraints. Furthermore, it has an unsatisfied constraint c3, and a directed cycle between constraints c1 and c2. For the same constraints, in (b) an alternative method graph is shown, in which all constraints are satisfied. How SkyBlue determines these graphs is explained later.

Two operations are provided for a SkyBlue graph, which are adding a SkyBlue constraint to the graph, and removing a constraint from it. After each of them, the method graph is updated.

To set the value of a SkyBlue variable, a Stay constraint is temporarily imposed on the variable. This constraint ensures that the connected variable is not changed in the propagation process. The Stay constraint is added and the method graph is updated, then the value is propagated, and finally the constraint is removed; now, the variable is free to be changed again.

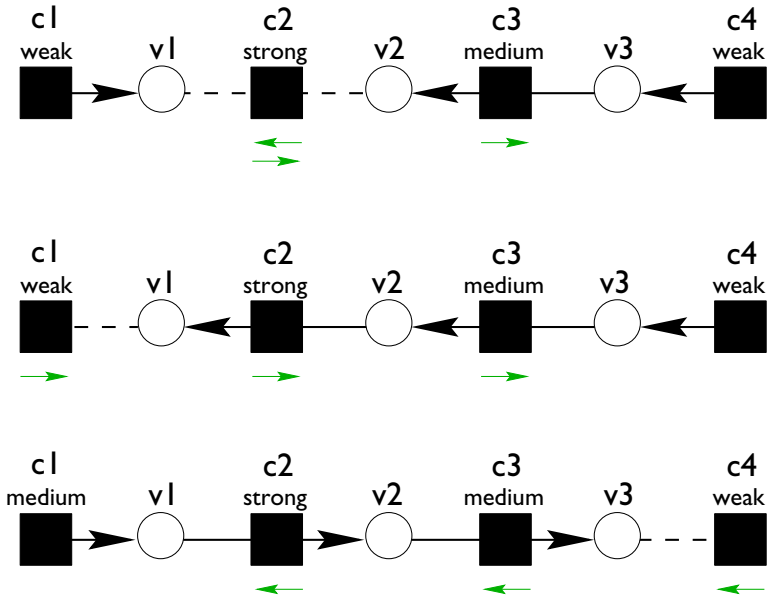


Figure 7.2: Three method graphs.

SkyBlue works incrementally, i.e. after addition or removal of a SkyBlue constraint, it takes the directions of the constraints in the previous method graph as the starting point for the algorithm, and typically changes the constraint directions only in a small subgraph. Only for constraints of which input values change, or that were previously unsatisfied, the selected method is executed.

SkyBlue only uses local information in the satisfaction process; it cannot handle circular dependencies of variables in the method graph. However, such directed cycles are detected by the algorithm, and provisions have been made for calling an external, more powerful, solver to handle such cycles.

SkyBlue constraints each have a priority, or *strength*, associated, indicating how important it is to satisfy the constraint. A constraint is said to be *weaker* than a constraint of a higher priority, which is called *stronger*. The highest strength is 'required', the lowest 'weak'. An arbitrary number of strength levels may be defined. If the SkyBlue constraint graph has conflicting constraints, SkyBlue will always determine a solution such that no unsatisfied constraint could be satisfied by making a weaker constraint unsatisfied.

For an example, see Figure 7.2. In the first method graph, the strong constraint c2 has just been added. The second method graph shows a possible method graph without conflicts; in order to satisfy c2, weak constraint c1 is left

unsatisfied. Another solution would be to leave c4 unsatisfied instead of c1. Since c1 and c4 have equal strength, SkyBlue will arbitrarily choose one of these solutions. The third method graph shows a solution when the strength of constraint c1 has been set to medium. Now, c4 is left unsatisfied.

7.2 Mapping

The SkyBlue solver has been incorporated in the constraint maintenance scheme of Chapter 6 as the solver of the primitive algebraic constraints. All algebraic constraints in the main constraint graph are mapped onto SkyBlue constraints. Variables of algebraic constraints are mapped onto a SkyBlue variable containing a floating point value. A constant value used in an algebraic constraint is mapped onto a SkyBlue variable with a required Stay constraint on it to prevent its change by the solver. The algebraic constraints presented in Section 5.1 are each mapped to one of these SkyBlue constraints.

In Chapter 6 it was explained that before solving the primitive algebraic constraints, i.e. the SkyBlue constraints, the map is updated by the Constraint Manager. This update involves two activities.

For algebraic constraints that are active and that were not yet mapped, SkyBlue constraints are created, and added to the SkyBlue graph. All SkyBlue constraints created get the default strength 'required'. The SkyBlue constraints are kept in the SkyBlue graph, until the constraint is either de-activated by the Constraint Manager or removed from the main constraint graph. In both cases, the SkyBlue constraints are removed from the SkyBlue graph.

For parameter values that were set by the user in the edit view, a required Stay constraint is created for the corresponding SkyBlue variable and added to the SkyBlue graph. They are kept until the constraint solving process is finished, after which their strength is set to 'weak'. Thus, in subsequent solving actions, SkyBlue will try to leave these variables unchanged, unless they are overridden by another, stronger, SkyBlue constraint. In this way, Stay constraints are used to control the propagation process.

As was explained in Section 6.3, algebraic constraints on implicit parameters are discarded in the constraint maintenance until the implicit parameters have been determined by the geometric solver. This is done by first de-activating the involved algebraic constraints, and mapping and solving the remaining algebraic constraints. When the geometric solver has solved the primitive geometric graph, and the results have been used to update the values in the main constraint

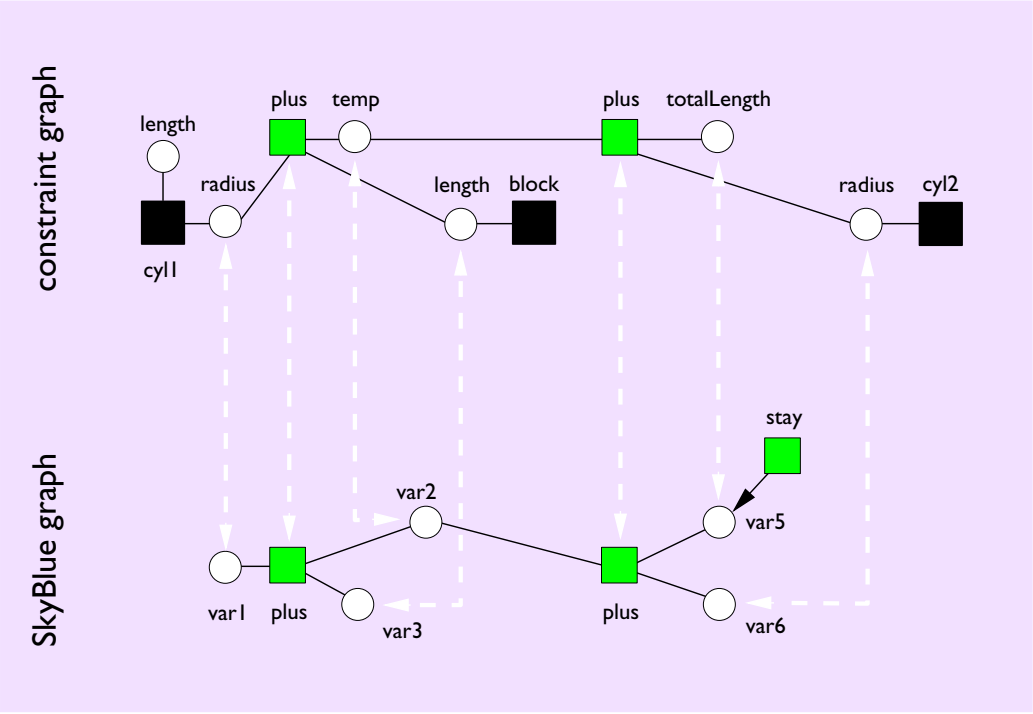


Figure 7.3: Map of algebraic constraints onto SkyBlue constraint graph.

graph, the implicit parameters are known. Then, a required Stay constraint is defined on these parameters, and the algebraic constraints on them are activated, mapped and solved.

The working is illustrated with the RoundedPocket feature of Figure 5.6 on page 66. Its initialization method Init() takes three real number arguments. Two of them specify a value for the width and depth of the block shape comprising the feature. The other is used to specify either the total length of the rounded pocket or the length of the block shape; only one of these is set in the Init() method. For the three parameters that are set in the Init() method, the Constraint Manager creates Stay constraints when mapping the algebraic validation constraints to the SkyBlue graph.

Figure 7.3 depicts a part of the constraint graph for the RoundedPocket, and its map onto the SkyBlue graph. Notice that parameters without algebraic constraints, e.g. cyl1.length, are not mapped onto the SkyBlue graph. The SkyBlue variable var5 corresponding to the user-defined parameter totalLength has a Stay

constraint imposed, because it was set in the Init() method of the RoundedPocket feature.

The priority of the Stay constraints for the RoundedPocket is first set to the strength 'required', in order to propagate their values to the other variables, for example those that correspond to the radius of the cylinders. In this way, the values for the private feature parameters are determined. When the algebraic constraints have been solved, the strength of the Stay constraints for the feature parameters is reduced to 'weak'. The value of these parameters can be changed by algebraic constraints that are added later.

Now, algebraic model validation constraints could be specified, for example one that makes the total length of this RoundedPocket instance equal to that of another RoundedPocket instance. When the algebraic constraints are solved again, SkyBlue will arbitrarily choose a total length parameter of one of the rounded pockets, and propagate its value to the other one. However, the user can specify in which direction the propagation is done, by re-specifying the total length value for one of the two RoundedPockets. He can also specify an Equal constraint on one of the two total length parameters, to give it a constant value. This approach ensures that the response of the modeling system to edit actions is as desired.

7.3 Implementation

A public domain implementation of SkyBlue in C++ has been used in SPIFF. The SkyBlue solver is requested by the Constraint Manager to update the SkyBlue graph by addition or removal of SkyBlue constraints. The Constraint Manager maintains a list of the Stay constraints that are created for parameters that are set by the user.

The implemented algebraic validation constraint types are Plus, Minus, Times, Division and Equal. They are derived from a base class AlgebraicConstraint. The map has been implemented in C++ by means of object pointers. Each AlgebraicConstraint object maintains a list of SkyBlue constraint objects that it has been mapped to.

The implemented feature parameter types are Parameter, AngleParameter and UserDefinedParameter. The first two are used as feature shape parameters. All three are derived from a base class ValueVariable, see Figure 7.4. Each ValueVariable stores a pointer to the SkyBlue variable object that it has been mapped to.

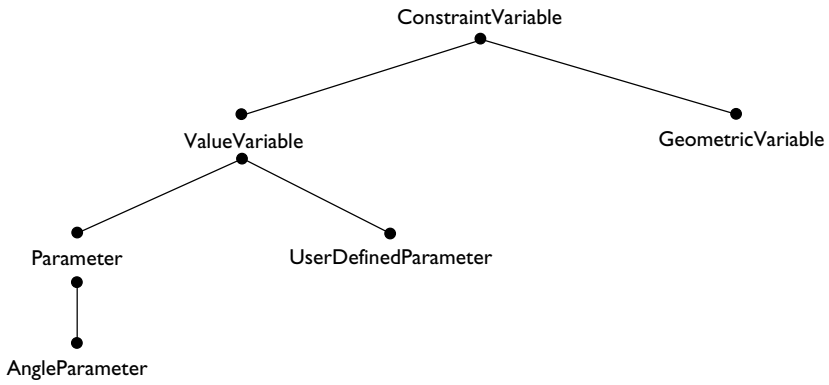


Figure 7.4: C++ class hierarchy for the constraint variables used in algebraic validation constraints.

7.4 Discussion

The constraint management approach presented in the previous chapter included a dedicated solver for the primitive algebraic constraints. The SkyBlue solver is used for this purpose. It is used to solve and maintain primitive algebraic constraints. These constraints are the result of mapping the algebraic validation constraints and the algebraic constraints that specify user-defined parameters.

One of the advantages of SkyBlue is that the propagation process can be controlled using the priorities of the constraints. Parameter values that were previously specified are kept as much as possible, unless they are overridden by stronger constraints. Therefore, SkyBlue propagates feature parameter changes in such a way that the model is minimally disturbed. Although not currently implemented, the SkyBlue priority mechanism could be used to handle priorities for algebraic constraints in the main constraint graph. The user would then be able to change priorities of algebraic constraints, to examine alternative product configurations.

Another advantage is that the set of primitive constraints can be easily extended, because SkyBlue does not make any assumptions about the meaning of the constraints. Useful extensions would be constraints for trigonometric expressions.

The SkyBlue solver is well-suited for the propagation of parameter values between the algebraic and the geometric graph, which is used for the algebraic constraints on implicit parameters. Future work will include a more direct propagation between the graphs, by alternating between the solvers whenever variable values from the other domain are required. This is where the incremental nature of SkyBlue will also be advantageous.

The SkyBlue solver is a general-purpose constraint solver, and therefore it is not very powerful, in the sense that it cannot apply domain knowledge needed to handle more complex combinations of constraints, such as circular dependencies. To overcome this, an extra algebraic solver could be incorporated to solve circular dependencies; this could be done, for example, by an iterative solver. Instead of multiple algebraic solvers, a single algebraic solver may be incorporated that can handle cycles directly, e.g. a solver based on simulated annealing, see Section 2.2.

Another disadvantage is that SkyBlue cannot handle inequalities; therefore, dimension constraints cannot be used in the propagation process, but can only be checked afterwards. Some solvers, e.g. simulated annealing solvers, are capable of solving equalities as well as inequalities. Incorporating such a solver would give an earlier detection of dimension constraint violations.

Again another disadvantage of SkyBlue is that it does not signal constraint conflicts explicitly; it simply leaves conflicting constraints unsatisfied. Therefore, after solving the SkyBlue graph and updating the main constraint graph, the Constraint Manager checks all algebraic constraints for violation.

8

Geometric constraints

This chapter presents how the geometry-related constraints are maintained. First, it presents an overview of the degrees of freedom analysis approach that was adopted for the dedicated geometric solver. Next, it gives details about the mapping of the geometry-related constraints to the primitive geometric constraint graph, and about how the map is maintained. Next, it discusses the concept of a dependency graph, which is used to handle over- and underconstrained geometric graphs. Then, implementation details of the geometric solver and the dependency graph are provided. Finally, the chosen geometric solving approach and its implementation are discussed.

8.1 Degrees of freedom analysis

Degrees of freedom analysis is a constraint satisfaction technique for 3D geometric constraint problems. Developed by Kramer [1992*b*], it was devised to describe the kinematic behavior of mechanical linkages, such as used in gear boxes and transmissions. It has been shown to be applicable in other domains as well, such as assembly and feature modeling [Shah, Ali and Rogers 1994, Salomons 1995].

Linkages consist of rigid *links* connected by *joints* that transform motion from one link to the connected link. A joint permits a certain relative motion, described by translational and rotational degrees of freedom. For example, a revolute joint allows a single relative rotation, i.e. it has one rotational degree of freedom. A helical joint allows a translational and a rotational motion that are coupled, i.e. it has a single, *coupled*, degree of freedom. Only a restricted set of kinematic joint types is required to model all possible kinematic linkages.

In order to describe the kinematic behavior of a linkage, it is formally described in terms of geometric constraints. For each joint, constraints specify the relative degrees of freedom, or DOFs, of two objects that represent the two

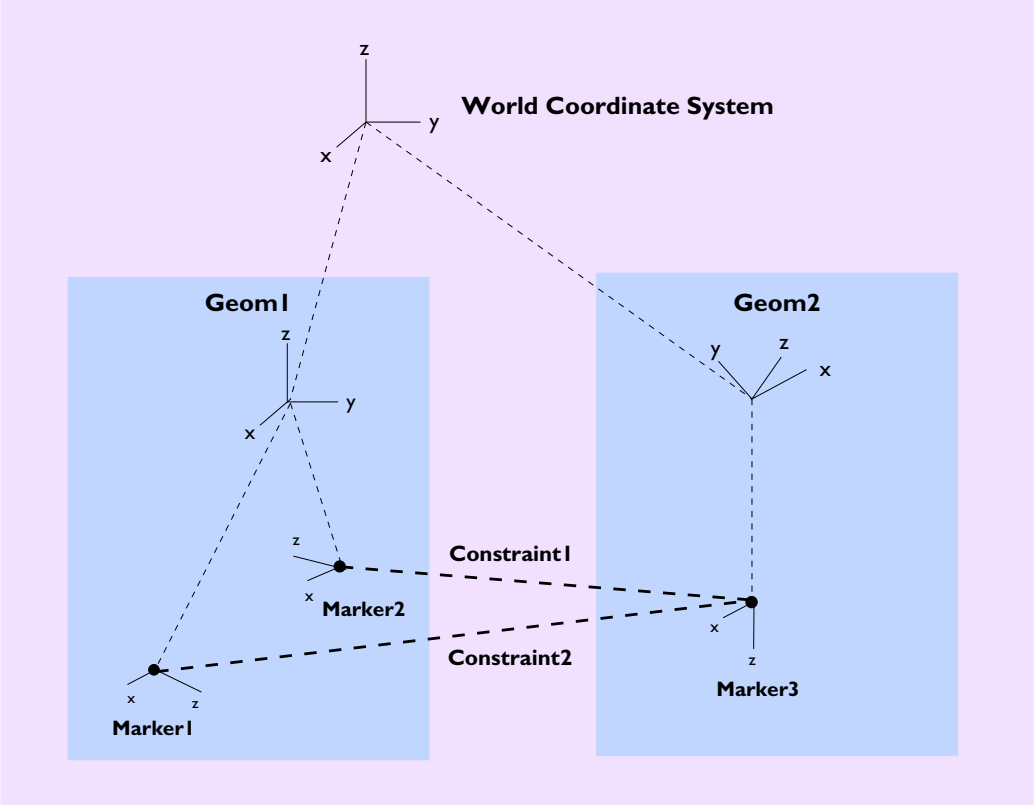


Figure 8.1: Object hierarchy used in degrees of freedom analysis.

links involved. Only the degrees of freedom of the links are considered, not their shape or size.

In the linkage domain, only those combinations of geometric constraints are needed that correspond to the kinematic joint types. In the domain of 3D geometric models, however, many more combinations are meaningful. This has implications for the degrees of freedom analysis, as will be shown later. In the remainder of this section, degrees of freedom is described in some detail, under the assumption that the domain is the general 3D geometric domain.

In degrees of freedom analysis, a model is described in terms of geometric constraints between pairs of *geoms*. A geom is an abstract geometric object that has three translational and three rotational degrees of freedom; the shape and size of a geom are irrelevant. A geom has a coordinate system defining the position and orientation of the geom with respect to a world coordinate system.

A geom may have one or more *markers*, which are embedded coordinate systems that are fixed with respect to the geom coordinate system. Markers are the variables on which the geometric constraints operate. See Figure 8.1 for an example.

Four types of geometric constraints between two markers M1 and M2 are used:

coincident positions of M1 and M2 are equal

in-line position of M1 is on the line through M2 and parallel to M2's z-axis, the *marker line*

in-plane position of M1 is in the plane through M2 and perpendicular to M2's z-axis, the *marker plane*

parallel directions of z-axes of M1 and M2 are equal.

The in-line and in-plane constraints are not symmetrical, i.e. the semantics of these constraints depends on the order of the two marker variables. This should not be confused with the direction of the constraints. None of the four constraints has an inherent direction; in the satisfaction process a direction is determined.

Besides these four constraint types, a few others are available: dist-point-plane and dist-planes. These specify a perpendicular distance from a marker or marker plane to another marker plane. In a graph preprocessing step, these constraints are translated by the geometric solver to in-plane constraints with markers that have been translated relative to their respective geoms.

The constraints and geoms are stored in a binary constraint graph; nodes represent geoms, edges represent one or more constraints. The term *joint* refers to the set of constraints between two connected geoms.

The goal of degrees of freedom analysis is to determine a configuration, i.e. positions and orientations for all geoms, such that all constraints are satisfied. If a model contains conflicting geometric constraints, or if a model is underconstrained, degrees of freedom analysis will detect this.

The main degrees of freedom analysis algorithm alternates between solving all rigid joints and restructuring all loops. A joint between two geoms is *rigid* if the constraints in the joint determine all DOFs of one of the geoms relative to the other one. When a rigid joint has been solved, the relative position and orientation of the two geoms have been determined. Therefore, the two geoms are combined into a *macrogeom* which embeds the two geoms; the embedded geoms are called *subgeoms*. In subsequent solving steps, the macrogeom is regarded as

a normal geom; macrogeoms embed other macrogeoms, and *basis geoms*, which are the geoms of the constraint graph when the solving process started. A loop is restructured by rewriting constraint information of the joints in the loop, such that some of the joints of the loop become rigid.

Two techniques are used: action analysis and locus analysis. *Action analysis* acts on a single joint; it satisfies all constraints in the joint, and if the joint is rigid, the two involved geoms are combined into a macrogeom. *Locus analysis* considers a group of three geoms that are connected by two or three nonrigid joints; it attempts to make two of the nonrigid joints stronger, by transferring constraint information of the third joint to the other two joints. Rigid joints are solved with action analysis, whereas for loop restructuring both action and locus analysis are used.

Degrees of freedom analysis is a structured, graph-based approach. It solves a constraint graph bottom-up, by first solving and then combining subgraphs, using geometric knowledge. In the analysis, the structure of the constraint graph is continuously altered; pairs of geom nodes and their joint are replaced by a single macrogeom node. For a well-constrained model, degrees of freedom analysis ends when the constraint graph has been reduced to a single macrogeom node that embeds all geoms of the original graph. The next two sections elaborate action analysis and locus analysis.

Action analysis

In *action analysis*, the constraints in one joint are satisfied incrementally, i.e. without disturbing an already satisfied constraint in that joint. At the beginning of action analysis for a joint, one of the two involved geoms is considered to be *grounded*, i.e. fixed in global space, the other one is allowed to be moved. The DOFs of the free geom are gradually removed by satisfying the constraints. Action analysis keeps track of the DOFs that remain for the free geom after a constraint has been satisfied. For each constraint, a transformation is computed for the free geom using its available DOFs, such that the constraint is satisfied. If the free geom has no DOFs left, after the constraints in the joints have been satisfied, the joint was rigid. The free geom then has the correct position and orientation relative to the fixed geom; therefore, the two geoms can be combined into a macrogeom.

For each constraint to be satisfied in a joint, action analysis uses a table with satisfaction procedures called *plan fragments*. It is a threedimensional table, indexed with the number of translational DOFs, rotational DOFs and the constraint type. A plan fragment is a procedural description; it performs degeneracy tests, computes transformations of the free geom, and computes the resulting DOFs of the free geom.

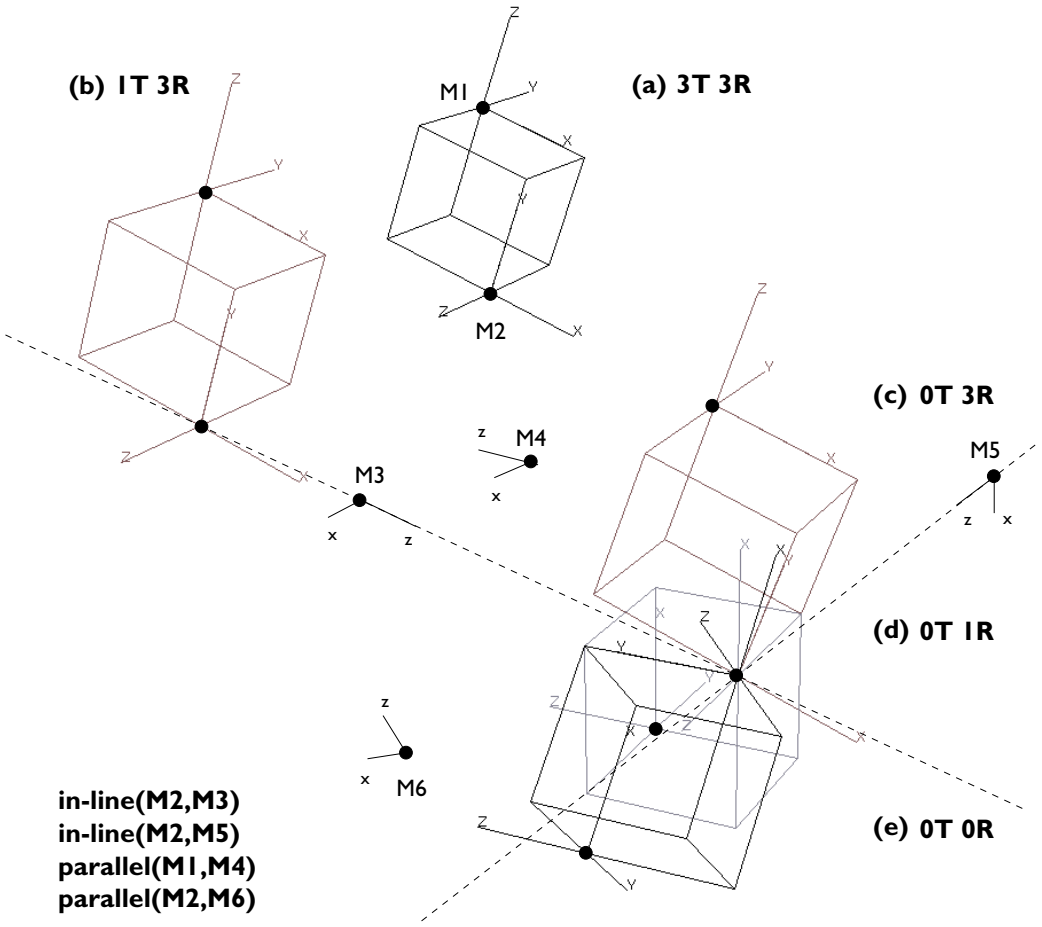


Figure 8.2: Action analysis reduces the DOFs of the geom in (a) by satisfying the constraints incrementally until no DOFs are left (e).

The state of a geom is represented in terms of number of DOFs, including vectors for translation directions and rotation axes. This information is used in the plan fragments for the degeneracy tests and for the computation of the transformations.

For an example, see Figure 8.2. A geom, embedding two markers M1 and M2, has two in-line and two parallel constraints imposed, as listed in the figure. The other markers M3-M6 are embedded in one grounded geom, which is not depicted. The free geom is depicted as a cube, but the actual shape of the geom is irrelevant for degrees of freedom analysis. The initial configuration of the geom

is (a). The first in-line constraint is satisfied by translating the geom such that M2 is on the line through M3 (b); this removes two translational DOFs. The second constraint is satisfied by translating the geom along the line through M3 to the intersection point of this line and the line through M5 (c); this removes the third translational DOF. The geom can still freely rotate about the marker M2. To satisfy the third constraint, the geom is rotated such that the z-axis of M1 has the direction of the z-axis of M4 (d); this removes two rotational DOFs, leaving one rotational DOF about the line through M2 and parallel to the z-axis of M4. To satisfy the last constraint, the geom is rotated about this line, such that the z-axis of M2 has the direction of the z-axis of M6 (e); this removes the third rotational DOF, and thus the position and orientation of the geom have been determined.

Action analysis determines a path in the plan fragment table, ending in a state with no DOFs left. The path is determined by the order of satisfaction of the constraints in the joint. A path can always be found, provided that plan fragments are available for all geom states that can be reached; action analysis is therefore said to be *complete*. Furthermore, the implementation of the plan fragments is such that, whichever order is chosen, the end state will always be the same; action analysis is therefore said to be *canonical*. In [Kramer 1992*b*], these two properties are proven for the domain of mechanical linkages, using an exhaustive case analysis. For the 3D geometric domain, this proof has not been given.

A joint may contain redundant constraints, which are not required to fully determine the DOFs of the free geom. For these redundant constraints, plan fragment are available that check whether the location of the free geom obeys the constraint.

A plan fragment may find that, given the location of the free geom and its available DOFs, there exists no transformation to satisfy the constraint. This implies that this constraint is conflicting with another, already satisfied constraint. Such an overconstrained and inconsistent situation is signaled; Section 8.3 explains how such a situation is handled.

Action analysis satisfies the constraints in one joint by determining the DOFs of one geom relative to the other geom. If a joint is not rigid, constraint information of its adjacent joints can often be used to make the joint rigid. This is the goal of locus analysis, elaborated in the next section.

Locus analysis

In *locus analysis*, three geoms connected by two or three joints are considered at the same time. The constraints in one joint are rewritten to constraints in the other joints; in this way, the configuration of the three geoms is specified

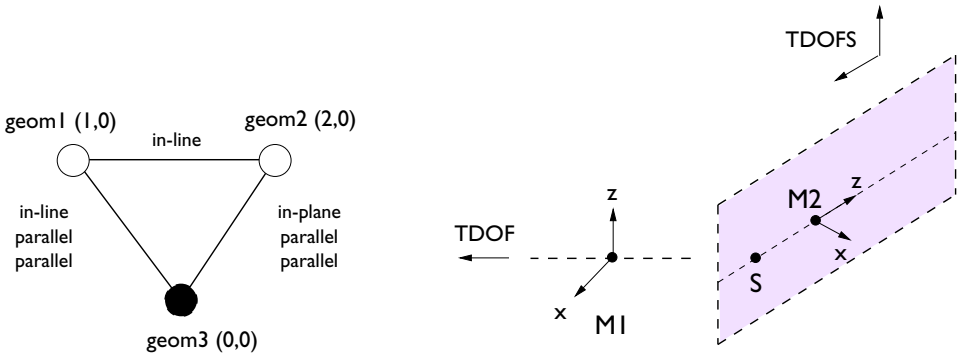


Figure 8.3: Application of locus analysis to a loop with three geoms.

differently, without loss of information, but in such a way that previously nonrigid joints may become rigid, and can therefore be solved.

A *locus* is the region of space that a geometric primitive can reach, when it moves according to its DOFs. In locus analysis, one constraint is considered at a time; loci are computed for the two constraint markers, according to the DOFs of the respective geoms, the constraint type, and the markers' location to the respective geoms. Locus analysis interprets a locus of a constraint marker as the region where the other constraint marker should be in order to satisfy the constraint. The two loci are intersected, and the intersection is used to create new markers and new constraints which are added to the constraint graph. This will be illustrated with an example.

Figure 8.3 (left) depicts a loop of three geoms. Geom3 is taken to be grounded. After action analysis on the two joints on geom3, geom1 has one TDOF left, whereas geom2 has two TDOFs left. The available DOFs are indicated between parentheses. Locus analysis is applied to the in-line(M1,M2) constraint between geom1 and geom2, with M1 a marker of geom1, and M2 a marker of geom2. The line is determined by the position and z-axis of marker M2. In this example, the two TDOFs of M2 are assumed to be dependent of the z-axis direction.

The point locus for M1 is a line with the direction of the TDOF of M1, as depicted in the right half of the figure; the line locus for M2 is the plane created by sweeping the line through M2's z-axis along the two TDOFs of M2. In general,

intersecting these two loci gives a point S; marker M1 must be coincident with this point S to satisfy the constraint, and the line specified by marker M2 must go through point S to satisfy the constraint. Therefore, the in-line constraint is rewritten to two constraints: coincident(M1,S), and in-line(S,M2); marker S is added to the grounded geom geom3, and the two constraints are added to the respective joints. These joints thus become rigid, and will be replaced by macro-geoms later. The newly created constraints remove DOFs of the two geoms in the same way as the original constraint, but via the grounded geom.

In the example, point and line loci were determined. For in-plane constraints, plane loci are determined by sweeping the plane according to the DOFs. For parallel constraints, orientation loci are computed; an orientation locus for a unit vector is represented by a part of a unit sphere, e.g. a circle in case the geom has one RDOF left.

In the previous example, locus intersection gave a single intersection point. If a discrete set of intersections is found, one of them is chosen. If a continuous intersection is found, standard locus intersection is not able to find a solution. Plane loci are often 3D, and will therefore often give a continuous intersection. Two approaches have been examined to prevent these continuous intersections [Noort 1997].

The first approach is to consider the loci for more than one constraint in the joint. Intersecting more than two loci may give a discrete intersection. However, intersecting loci of different constraints is only allowed for markers that have the same global position. Furthermore, plane loci will often be 3D, and intersecting them will not yield a locus with a reduced dimensionality. Therefore, another approach was devised.

This second approach is to reduce the dimensionality of a locus for a constraint, before intersecting it. The approach ignores DOFs which are not affected by the constraint, thus giving a simpler locus. It can be applied to in-line and in-plane constraints. Figure 8.4 shows an example. Consider an in-line constraint between M1 on geom1 and M2 on geom2, and assume that geom1 has one TDOF left, and geom2 has two TDOFs left, as depicted. The point locus for M1 is a line, the line locus for M2 is the whole 3D space. In general, intersecting the loci gives a continuous intersection; however, if the TDOF direction of geom1 is parallel to the line direction of M2, this TDOF will not be removed when the constraint is satisfied. Therefore, this TDOF can be ignored, and the point locus can be reduced to a single point; now, intersecting the loci gives this single solution. Notice, however, that in contrast to the previous example, no coincident constraint is added to the graph, because the TDOF of M1 is not removed by the constraint.

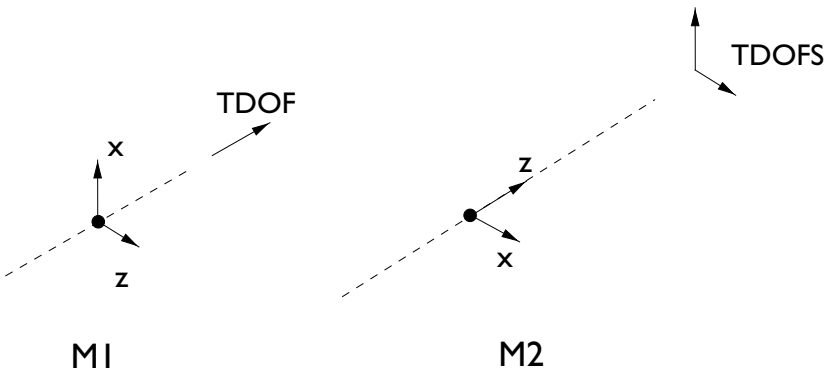


Figure 8.4: Ignoring DOFs to reduce the dimensionality of a locus.

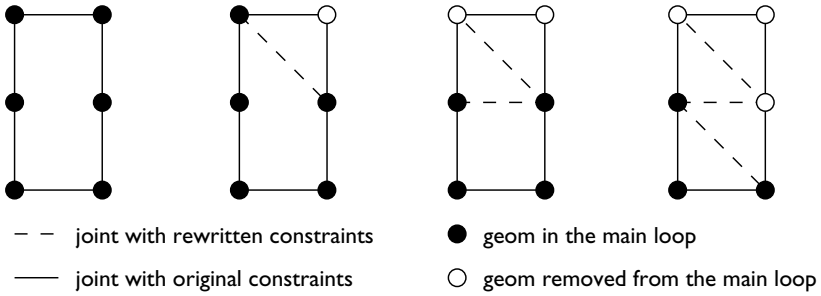


Figure 8.5: A loop is handled by repeatedly applying locus analysis, thus splitting it into loops of three geoms.

Locus analysis is applied to groups of three geoms. Such a group may form a loop; larger loops are handled by applying locus analysis to three consecutive geoms at a time, thus creating joints with rewritten constraints, see Figure 8.5. After locus analysis, constraints in the group of three geoms have been rewritten; the middle geom of the group is removed from the loop. Then, locus analysis is applied to

another group of three geoms in the remaining loop. In this way, a loop is split in several stages into groups of three geoms.

Degrees of freedom analysis solves the primitive geometric constraint graph, which results from mapping the geometry-related constraints in the main constraint graph. The next section elaborates this mapping.

8.2 Mapping

A constraint solver based on degrees of freedom analysis has been incorporated in the constraint maintenance scheme of Chapter 6 as the primitive geometric constraint solver. All geometry-related constraints in the main constraint graph are mapped onto primitive geometric constraints, which we will call *DOF constraints* in the remainder of this thesis.

Each face constraint variable in the main constraint graph is mapped onto a geom. The position and orientation of the geom correspond to the position and orientation of the origin of the face, which depends on the shape type that the face belongs to. The geom embeds several markers, used for the DOF constraints.

Shape constraints are mapped onto a group of DOF constraints between the geoms that correspond to the shape faces. For an example, see Figure 8.6 (left), depicting the map of the cylinder shape. For the three geoms, top, side and bottom, several markers are used, of which the z-axis is depicted. The marker with number 1 is the marker with the same global location as the geom, and is called the *main geom marker*. The constraints that result from the mapping are listed in the figure. The first six constraints specify the shape in terms of relationships between geoms. The last two specify the shape parameters height and radius.

Figure 8.6 (right) depicts the map of a vblock shape. A vblock is a generalized block that has two slanted faces, which can be specified by independent shape parameters. The first six constraints specify part of the shape in terms of relationships between geoms. A vblock is mapped onto two groups of three geoms each; only the constraints for one group are shown. The front and back faces have angle parameters associated; they specify the inner angle from the top face to the front and back faces. For the angle parameters, parallel constraints are used with a marker with rotated z-axis. The last constraint specifies the shape parameter length; similar constraints are used for the height and width parameters.

Attach constraints are mapped onto an in-plane and a parallel constraint. The attach face and attach reference face are assumed to be already mapped. For the parallel constraint, a new marker is created that is coincident with the marker

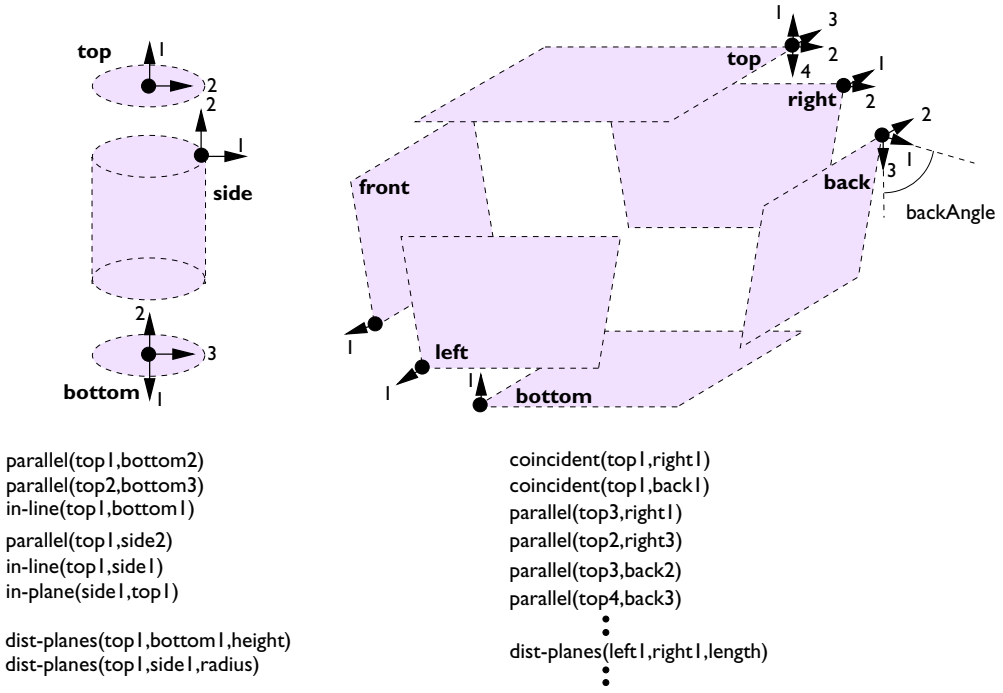


Figure 8.6: Maps of cylinder and vblock shape constraints.

of the geom corresponding to the attach reference face. The orientation of the z-axis of this new marker obeys the semantics of the type of attach constraint involved, as explained in Section 5.1; therefore, the new marker has a direction that is either equal or opposite to the direction of the main geom marker.

Geometric constraints are mapped onto various combinations of DOF constraints. For example, a DistPointFace constraint is mapped onto a dist-point-plane constraint; a DistXYPointFaceConstraint is mapped onto two dist-point-plane constraints, with markers on the reference geom that are rotated in such a way that their z-axes are aligned with either the x-axis or the y-axis of the reference geom. An AngleFaceFace constraint specifies an angle between two parallel faces about the normal of the first face; see Figure 8.7. It is mapped onto a parallel constraint, with markers rotated such that their z-axes are in the respective planes and have the specified relative orientation.

Link constraints between planar faces are mapped in a similar way as the attach constraints. Link constraints between cylindrical faces are mapped onto

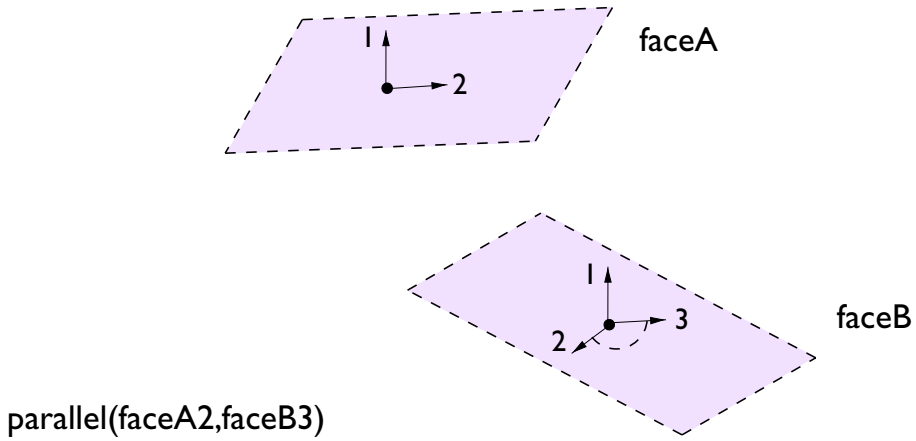


Figure 8.7: Map of AngleFaceFace constraint.

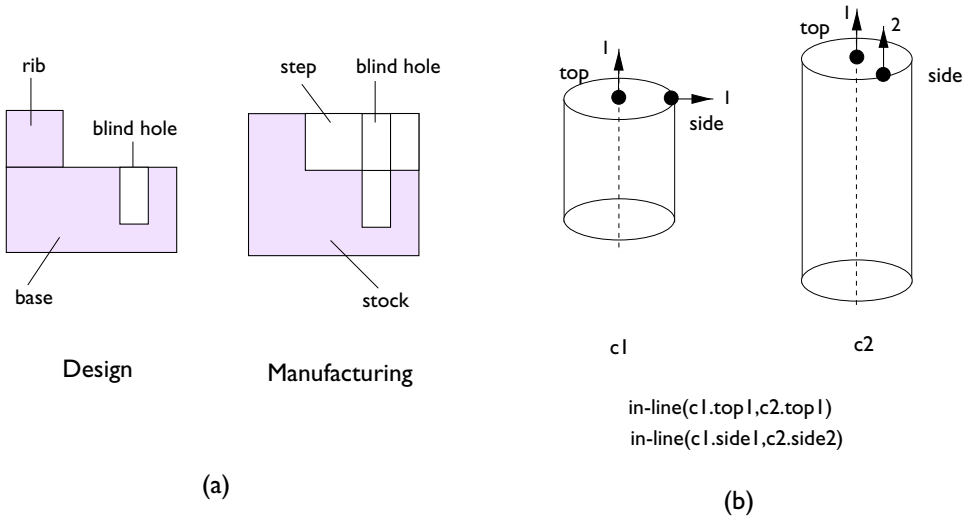


Figure 8.8: Example of two views (a). Map of a link constraint between two cylindrical faces; c1.side in a non-edit view is linked to c2.side in the edit view (b).

in-line constraints, see Figure 8.8. In (a) two views are shown of a product; the two blind holes are coaxial, but do not have the same depth. Assume that the manufacturing view is the edit view. In (b) the map is shown for a link constraint from cylindrical face c1.side to cylindrical face c2.side of the edit view. With the first in-line constraint indicated, the cylinders are made coaxial, and with the second in-line constraint, the side geoms become colinear, effectively coupling the radius of c1 to the radius of c2. The height parameters of the cylinders remain uncoupled, as required.

Before solving the DOF constraint graph, the map is updated by the Constraint Manager. For geometry-related constraints that are active and that were not yet mapped, DOF constraints are created. The DOF constraints for a geometry-related constraint are kept in the DOF constraint graph, until the constraint is either removed from the main constraint graph or is de-activated by the Constraint Manager: the DOF constraints are then removed from the DOF constraint graph. The geoms are kept until their associated variables in the main constraint graph are removed, which happens when their shape constraint is removed.

As was explained in Section 6.1, a distinction is made between explicit and implicit shape parameters. The presence of an active fix value constraint implies that a parameter is explicit. The Constraint Manager uses this information to update the map of a shape constraint by adding or removing the DOF constraints that specify a distance or angle value. Furthermore, for each shape parameter value that was determined in the algebraic constraint solving, the Constraint Manager updates the map of the associated shape constraint by translating or rotating a marker on a geom of the involved shape.

When the DOF constraint graph has been solved, the position and orientation of the geoms are known, and thus of the faces in the main constraint graph. All shape constraints determine their parameter values by computing distances and angles between their faces. Distance and angle parameter values of de-activated geometric constraints are computed similarly.

In this section, it was assumed that the DOF constraint graph is well-constrained, or consistently overconstrained. The next section explains how underconstrained and inconsistently overconstrained graphs are handled.

8.3 Over- and underconstrained models

Degrees of freedom analysis is able to detect underconstrained or inconsistently overconstrained models. To handle such models appropriately, the basic degrees of freedom analysis approach has been extended [Noort et al. 1997]. In the re-

mainder of this chapter, the term *overconstrained* is used to refer to inconsistently overconstrained models only.

The extended degrees of freedom analysis approach has three extra phases, carried out when an over- or underconstrained model is found:

1. identification: the involved constraints and DOFs are determined

2. repair: the constraint graph is modified to reach a consistent and fully determined model

3. re-solving: the modified graph is re-solved.

In the *identification* phase, the geometric solver determines the scope of the problem. A model is found to be overconstrained when during action analysis a constraint could not be satisfied using the available DOFs of the free geom. The solver first identifies the *overconstrained* DOFs, i.e. the DOFs required to satisfy the constraint. Next, it analyzes the constraint graph to determine the *involved* constraints, i.e. the constraints that caused the overconstrained DOFs to be removed.

A model is found to be underconstrained when there are geoms with DOFs left after all constraints have been satisfied. In this case, the solver identifies the relative DOFs that are left between these geoms.

Repairing an overconstrained situation can be done in two ways. Firstly, one or more involved constraints can be removed from the graph, or replaced by other constraints. Secondly, one or more involved constraints can be edited, e.g. by translating a marker relative to its geom. This removes the inconsistency, but not the overconstrained situation; if later, after the feature parameters have been changed, and thus markers have been re-located, the graph is solved again, another inconsistency will be found. Therefore, the first way is adopted.

Repairing an underconstrained situation is done by adding constraints that remove the DOFs that were left.

Re-solving of the modified DOF constraint graph is done incrementally. The part of the constraint graph that needs to be re-solved is determined on the basis of the constraints that are added or removed from the graph.

We conclude that to handle over- and underconstrained models, it is required to find the geometric properties that are determined by each constraint in the solving process. This information is stored in a *dependency graph*, which is elaborated in the next section.

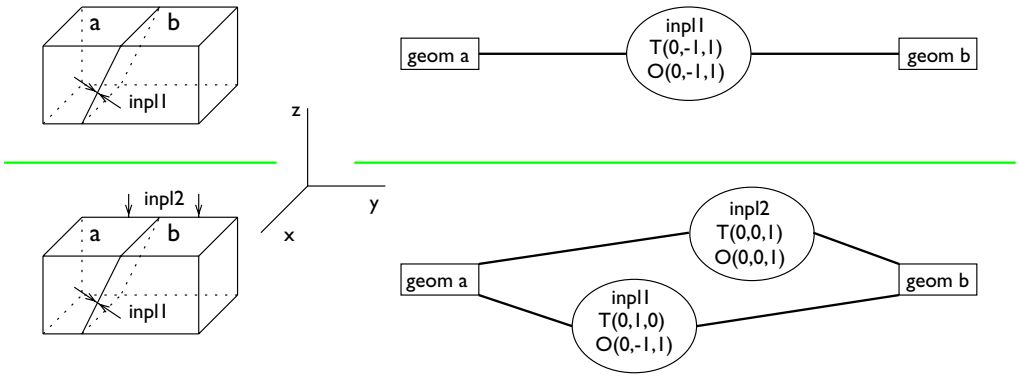


Figure 8.9: A dependency graph stores the DOFs that are removed by a constraint.

Dependency graphs

A dependency graph stores the dependency relations of geoms, as determined by the constraints between them. These relations are not known a priori, but depend on the order of solving the constraints. If the constraint graph is over-constrained but consistent, it contains redundant constraints; these constraints do not remove, or *reduce*, any DOFs, and therefore they do not contribute to the dependency relations.

The dependency graph is a bipartite graph, with two types of nodes that represent basis geoms and the constraints that reduced DOFs in the solving process. With a constraint node, it is stored which DOFs it reduces. Figure 8.9 shows an example. In the top half of the figure, a model containing two geoms a and b depicted as vblocks is shown; they are connected by an in-plane constraint inpl1 between a marker on geom a and one on geom b. In the dependency graph shown on the right, it is stored that the inpl1 constraint reduces the TDOF with direction (0,-1,1), which is equal to the orientation of the z-axis of the plane marker on geom b indicated as O(0,-1,1). The DOFs that a constraint reduces when no other constraint is involved, are called the *initial DOFs*; these only depend on the constraint type, and the orientation of the involved markers.

In the bottom half of the figure, the two vblocks are connected by two in-plane constraints, with initial DOFs that are dependent. If the inpl2 constraint is solved first, it removes TDOF (0,0,1), and thus the inpl1 constraint will remove only the TDOF along (0,1,0); these DOFs are called the *actual* DOFs of the constraint. ‘

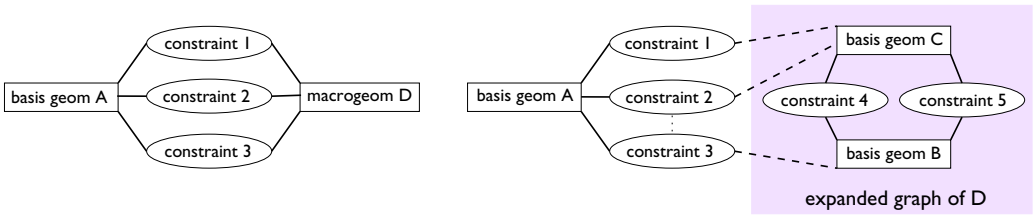


Figure 8.10: Joint dependency graph (left), and its expanded dependency graph (right).

The dependency graph is generated during the solving process. It registers which DOFs are removed by each constraint that is satisfied in a rigid joint. Dependency graphs are built in two steps, see Figure 8.10. Firstly, when the constraints in the joint are being satisfied, a dependency graph for the joint is built, the *joint graph*. It contains only the dependencies between the two geoms in the joint. Secondly, when the two geoms are combined into a macrogeom, the macrogeom nodes in the joint graph are expanded, i.e. they are replaced with the dependency graph that was computed earlier for that macrogeom; thus, the joint graph is transformed into an *expanded graph*. In this way, the dependencies of the basis geoms in the macrogeoms are put into the expanded graph; therefore, an expanded dependency graph contains only geom nodes for basis geoms.

The concept of a dependency graph has been used previously. Two uses can be distinguished. Firstly, a dependency graph can be used to solve a constraint graph. If a constraint graph is undirected, it can be made directed to find the dependencies of constraint variables; when the dependencies have been determined, the graph can be solved. Examples are given in [Serrano and Gossard 1988, Sannella 1992]. Others focus on geometric dependencies, e.g. Hsu and Brüderlin [1997]; their approach is described in Section 2.4. Secondly, a dependency graph can be used to analyze over- and underconstrained models. This is done, for example, by Suzuki et al. [1990] and by Keirouz et al. [1993]; we make similar use of a dependency graph, but store more detailed dependency information in it, to be able to analyze the DOFs involved.

In SPIFF, the geometric constraint graph is solved without use of a dependency graph; during solving, the dependencies are determined and stored in a dependency graph. The dependency graph is used to find the involved constraints for an overconstrained model. For underconstrained models, it is used to find the basis

geoms that are fixed relative to each other. These applications of dependency graphs are elaborated in the next two sections.

Overconstrained models

An overconstrained situation is detected during solving of the constraint graph, either in action analysis or in locus analysis. In both cases, the overconstrained DOFs have to be identified.

During action analysis, the overconstrained DOFs are determined in the plan fragment in which the overconstrained situation was detected. In the plan fragment, a transformation is computed to satisfy the constraint given the DOFs of the free geom; if no such transformation can be determined, the overconstrained DOFs are known.

During locus analysis, an overconstrained situation is found when two loci have an empty intersection. In order to determine the overconstrained DOFs, one of the two geoms for which the loci were determined is given more freedom, i.e. its locus is gradually enlarged, until it gives an intersection with the other locus. The DOFs that were added to the geom are the overconstrained DOFs. For example, consider a coincident constraint, for which one locus is the point (5,5,5), and the other locus is a line with direction (0,0,1) through point (0,0,0). To find the overconstrained DOFs, the second geom is given an extra TDOF that is independent of (0,0,1), say (0,1,0); thus, the second locus becomes a plane with normal vector (1,0,0). Because the loci still have an empty intersection, the second geom is given another TDOF (1,0,0). Now the loci have a nonempty intersection; the overconstrained DOFs are therefore (0,1,0) and (1,0,0).

After the overconstrained DOFs have been determined, the involved constraints are determined using the dependency graph. Because constraint nodes are only added to the dependency graph if they reduce DOFs, the graph contains a loop of constraints that reduce the overconstrained DOFs.

Figure 8.11 shows an example to illustrate the approach. On the left an overconstrained model is shown, consisting of three geoms that each represent a shape. Assume that the overconstrained situation is detected when constraint inpl3 could not be satisfied in action analysis. The overconstrained DOF found in the plan fragment is the translation in direction (0,1,0). The dependency graph is shown on the right. It is searched for the involved constraints; these are the constraints on paths between geom a and geom b that reduce the overconstrained DOF, indicated by the dashed lines. One of these constraints, inpl4, actually reduces a DOF that is different from its initial DOF; this is caused by other constraints that reduce DOFs that are dependent of the initial DOF of constraint

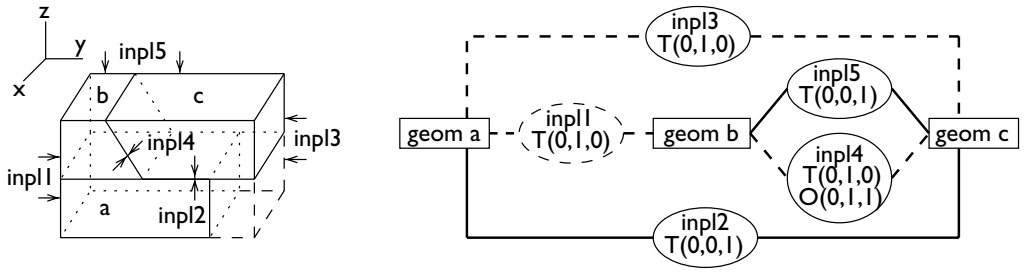


Figure 8.11: Overconstrained model (left), and its dependency graph (right).

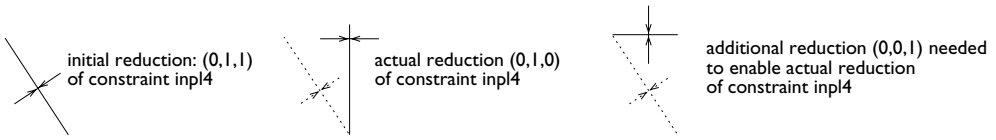


Figure 8.12: Initial reduction may differ from the actual reduction of a constraint.

inpl4, see Figure 8.12. These constraints are found on paths between the geoms of constraint inpl4, and are added to the set of involved constraints. In this example, constraint inpl5 is therefore added to the involved constraints.

In the previous example, the overconstrained situation can be repaired by removing one of the involved constraints, e.g. by removing constraint inpl3. This action would lead to the model configuration shown in the figure. The constraint selected for removal should not reduce more DOFs than the overconstrained DOFs to prevent the model from becoming underconstrained. If no such constraint is in the set of involved constraints, one of them should be replaced by another constraint that frees the overconstrained DOFs without making the model underconstrained.

If several constraints are candidates for removal or replacement, selection criteria can be applied, e.g. based on user-defined priorities. In the current implementation, the user is requested to select a constraint from the set of involved constraints. Since detection of overconstrained models is done when solving the DOF constraint graph, the map of the main constraint graph is traced back to

translate the involved DOF constraints into model validation constraints or feature parameters. The user can remove one of the suggested model validation constraints, or he can de-activate some of the fix value constraints for feature dimensions, or some of the geometric feature validation constraints for feature positions or orientations.

When a DOF constraint has been removed or replaced, part of the DOF constraint graph has to be re-solved. This part is the previously solved part of the model that corresponds to the expanded dependency graph containing the removed or replaced constraint; it can easily be determined in the dependency graph.

Underconstrained models

A model is found to be underconstrained when, after solving, multiple geoms with DOFs are left in the DOF constraint graph. The solver identifies the remaining DOFs between each pair of these geoms. For each pair, there may be several paths between the two geoms in the solved DOF graph; all constraints in these paths are rewritten to one joint between the two geoms, using locus analysis. Then, the relative DOFs of the two geoms are determined with action analysis on this joint.

An underconstrained situation needs to be repaired, in order to give visual feedback to the user about which parts of the model are underconstrained. Repairing is done by adding constraints to the model such that the remaining DOFs are removed. This is done for each pair of geoms, by adding constraints to the associated joint to remove the DOFs. After constraints have been added to a joint, the model is re-solved incrementally, since this joint may also remove DOFs of the other joints. This is repeated for the other joints, until the model has become well-constrained.

Constraints added to an underconstrained model provide values for unspecified feature parameters. These values are currently chosen arbitrarily, to provide the user with an image of the model plus an indication of which DOFs are underconstrained. The dependency graph is used to find the basis geoms in the macrogeoms that are left, which are used to determine the unspecified feature parameters. The user can add model validation constraints, or activate feature validation constraints in order to make the model well-constrained.

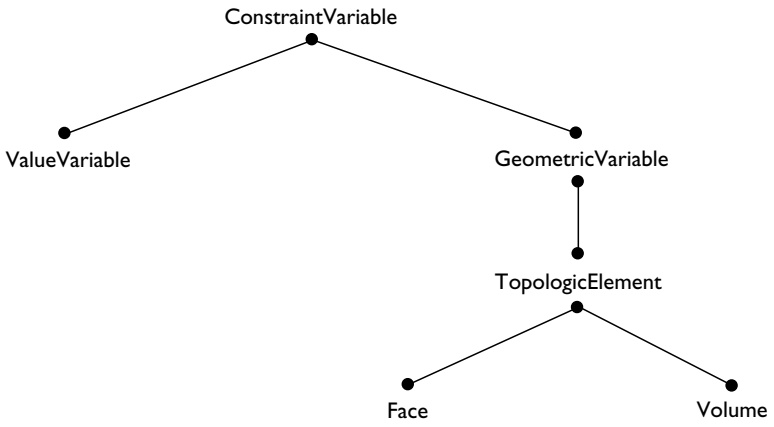


Figure 8.13: C++ class hierarchy for the constraint variables used in geometry-related validation constraints.

8.4 Implementation

Implementation of the geometric solver in SPIFF was done in C++. Figure 8.13 depicts the class hierarchy for the constraint variables that are used in geometry-related validation constraints. The class GeometricVariable stores geometric data such as position and orientation. Its derived class TopologicElement is the base class for the feature element types, of which currently Face and Volume have been implemented. A useful extension of the hierarchy would be a class Datum, derived from GeometricVariable; a Datum could be used as a globally located reference for specifying position and orientation features.

The implementation of the DOF constraint graph is based on the ConstraintGraph class elaborated in Section 6.2. The DOF constraint classes are derived from a base class Constraint; class Geom is derived from the base class ConstraintVariable [Kuiper 1995]. The implementation of the dependency graph is also based on the ConstraintGraph class [Noort 1997].

The map of the geometry-related constraints to the DOF constraints has been implemented by means of object ids. Each geometry-related constraint stores a list of ids of DOF constraint objects, and a list of ids of marker objects that were created in the mapping of the constraint. When geometry-related constraints are mapped to or removed from the map, these lists are updated. Figure 8.14 depicts the different objects involved and their pointer relations.

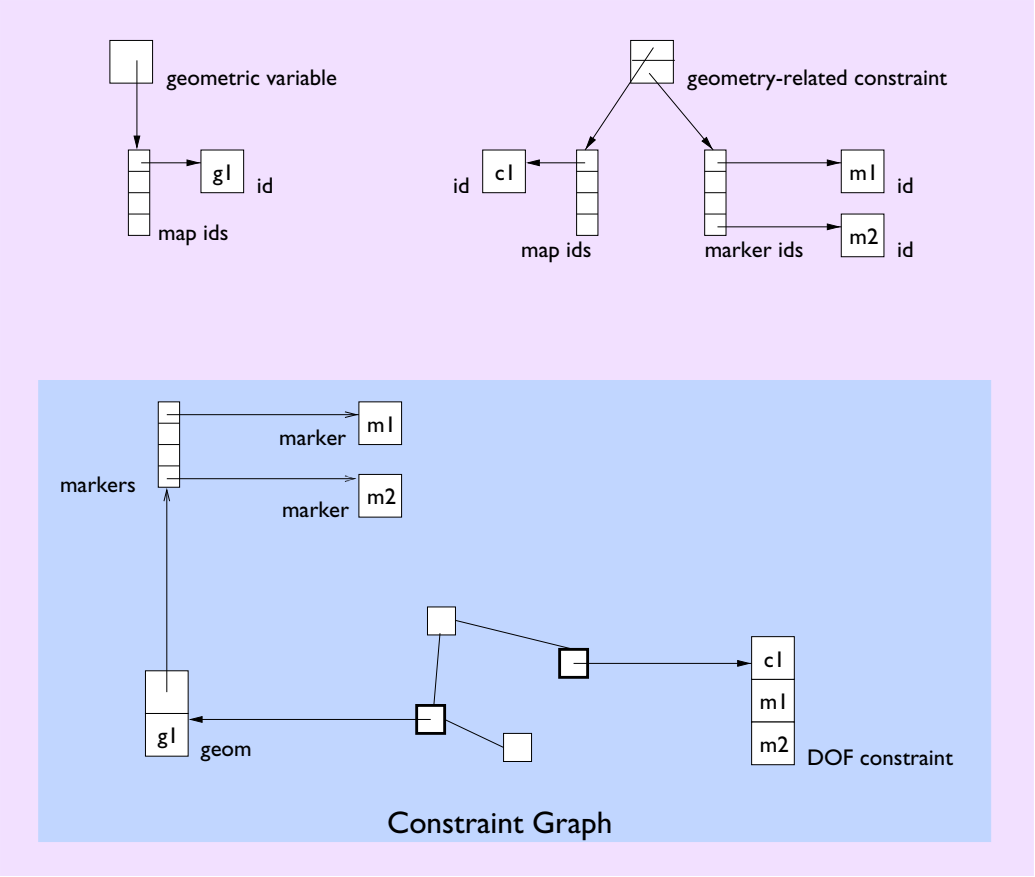


Figure 8.14: Map of geometric variables and geometry-related validation constraints.

8.5 Discussion

The constraint management approach presented in Chapter 6 included a dedicated solver for the primitive geometric constraints that result from mapping the geometry-related constraints. The dedicated geometric solver uses degrees of freedom analysis. Degrees of freedom analysis is a structured, graph-based approach for solving 3D geometric constraints.

It has two main advantages. Firstly, it employs geometric knowledge in the solving process, which ensures geometrically meaningful solutions, and which supports the analysis of over- and underconstrained models. Secondly, its in-

cremental nature allows efficient solving of the primitive geometric constraint graph.

On the other hand, one of the disadvantages of degrees of freedom analysis is that it only supports planar surfaces. This is partly overcome in the mapping process of cylinder shape constraints: cylindrical surfaces are represented by one characteristic point and a radius, which allows geometry-related constraints on a predefined point of a cylinder side face. Other surfaces, e.g. cones and B-spline surfaces, could also be incorporated by defining constraints on characteristic points; this requires further investigation. Another disadvantage is that the implementation is very elaborate; there are many plan fragments required, and many degenerate cases have to be considered.

Comparing degrees of freedom analysis as implemented in SPIFF, with the approach described in [Kramer 1992*b*], we can point out several differences. These are mainly consequences of the more general 3D geometric domain of SPIFF.

When a joint is considered in the solving process, it is in general not known a priori whether it is rigid. It would be possible to enumerate the combinations of constraint types that together could form a rigid joint. However, it is required that elaborate tests are performed for degenerate cases; for example, three coincident constraints form a rigid joint, unless the three markers of the constraints on one geom are colinear. Because action analysis has to perform these tests in the plan fragments anyhow to know the resulting geom state, there is no real advantage in determining a priori whether a joint is rigid or not. Therefore, instead of analyzing a joint before applying action analysis, as done in [Kramer 1992*b*], we have adopted a trial-and-error approach: action analysis is done for each joint, and joints found to be nonrigid are marked. Joints are only re-evaluated if they received constraints due to a rewrite action in locus analysis; this ensures only a small performance loss.

In degrees of freedom analysis, loops are handled by rewriting constraints of one joint to the other two joints in the loop. In this locus analysis, loci of reachable marker locations are intersected; plane loci are often volumetric, and intersection will not yield a discrete solution set. In degrees of freedom analysis as described by Kramer, joints between basis geoms are kinematic joints. In general, these joints restrict more DOFs than the joints in the more general geometric domain; for example, all kinematic joints restrict translational DOFs. It is shown in [Kramer 1992*b*] that to solve a loop in the domain of mechanical linkages, intersections with plane loci do not have to be considered, because the TDOFs that would be removed by the in-plane constraint are removed through other joints; hence, locus analysis for in-plane constraints is not required. In the 3D

geometric modeling domain, however, locus analysis for in-plane constraints is required.

The problem of not finding discrete intersections can be overcome in many situations. A locus can often be simplified by ignoring DOFs of the geoms that are not influenced by the involved constraint; this reduces the dimensionality of the locus. This approach is not limited to in-plane constraints; we have applied it to in-line constraints also. In practice, it has shown to be very useful.

In degrees of freedom analysis as described by Kramer, the rigidity of a loop was determined a priori; a rigid loop was then solved in isolation, by applying action and locus analysis. Because the general geometric domain is not limited to kinematic joints only, and many more degeneracies can occur, we have adopted an approach in which loops are not solved, but restructured with locus analysis. When a loop is restructured, constraints are rewritten; several loops are restructured, until a rigid joint has been created. An important advantage of this restructuring approach is that constraint information of a nonrigid loop is transferred to adjacent loops, i.e. loops that share one or more joints with it. Constraints in one of the loops are rewritten to the joints shared with other loops, and then the constraints in these shared joints are rewritten, thus possibly making joints rigid. If loops were solved in isolation, this would not be possible.

Locus analysis as described is only applicable to loops of three geoms. Kramer proposes to handle larger loops with *extended locus analysis*. Extended loci represent reachable points for a marker on a geom that is connected to another geom which is itself not fully constrained. An extended locus is computed by sweeping a regular locus; hence, an extended locus is likely to be volumetric, and thus not really useful in locus analysis. Instead of computing extended loci, we handle loops with more than three geoms by recursively splitting them into loops of three geoms, and restructuring these with locus analysis to rewrite constraint information.

In the implemented degrees of freedom analysis, geom states are represented in terms of their available TDOFs and RDOFs. These are represented as points, lines and planes that are embedded in the geom; they are said to be *invariant* with respect to the geom [Kramer 1992*a*]. The DOFs may be coupled; consider a 2D example, where a geom embeds a marker that is constrained at a distance from another marker, and embeds another, not coincident, marker that has a relative orientation. In this case, the position of the geom is described by an ellipse [Noort 1997]; therefore, a geom may also embed an ellipse. A representation of the state of a geom in terms of invariant geometric primitives is, however, not suitable to handle all combinations of 3D geometric constraints, because

some states cannot be described in analytical form. Kramer suggests to employ numerical equation solvers for such situations; this requires further investigation.

Depending on the geom state, a constraint may be satisfied in multiple ways. For example, an in-plane constraint for a geom with zero TDOFs and one RDOF can be satisfied with a rotation; however, two rotation directions are possible. Arbitrarily choosing one of the two may not be consistent with other constraints encountered later; in that case, backtracking is required. Since backtracking requires significant overhead in terms of computation and data structure management, we have adopted a simple strategy, where in each joint first constraints are satisfied that remove as many RDOFs as possible, and then the constraints that remove TDOFs are satisfied. In this way, the number of situations with multiple satisfaction possibilities is considerably reduced. In practice, this approach gives satisfactory results.

In degrees of freedom analysis as described by Kramer, the sequence of plan fragments executed is stored, and is re-evaluated when model parameters are changed but the constraint graph topology is not changed. This approach is not correct in general, however, since during re-evaluation, degenerate situations may be encountered, which will lead to a different geom state at the end of the involved plan fragment; in this case, the remainder of the previous sequence is not valid, and should be re-determined by solving the constraint graph. Therefore, storing this sequence is not really useful. The approaches of others who first determine a dependency graph, and then evaluate it for various model parameter values, suffer from the same degeneracy problems.

In our approach, the Constraint Manager selects a part of the model for re-evaluation; the geometry-related constraints in this part are then solved by the geometric solver, while a dependency graph is constructed for possible over- and underconstrained situations. In this way, model re-evaluation is done efficiently, while model validity is ensured.

The implementation of the dependency graphs differs slightly from how it was presented. For implementation reasons, the overconstrained DOFs are not reported by the plan fragment in which the conflict was found; instead the initial DOFs of the constraint are always used. This does not lead to a larger set of involved constraints, because it is accounted for when determining the loops of involved constraints. Consider Figure 8.11, where the actual DOF of constraint inpl4 was (0,1,0). Its initial DOF was (0,1,1); if this direction is used as the overconstrained DOF, loops will be searched between geom b and geom c that contain constraints removing a DOF that is dependent of direction (0,1,1). Constraint inpl2 removes DOF (0,0,1), which is dependent of (0,1,1), but this

constraint is not part of a loop of constraints that remove this DOF. Therefore, constraint inpl2 will not be included in the set of involved constraints.

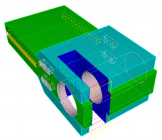
In the current implementation of dependency graphs, the initial DOFs of coincident, in-line and in-plane constraints are assumed to be TDOFs; for these constraints, rotational dependencies are not considered when an overconstrained situation has been detected. The implementation should be extended in this respect in order to find all involved constraints.

The application of dependency graphs to overconstrained models is used to determine the set of conflicting constraints. Which constraint is selected for removal, now depends on the user. It would be desirable to automate this selection. Priorities of constraint types could be used for this; for example, if there is an overconstrained situation because a feature parameter has been set by the user, but is also determined by a geometric model validation constraint, it seems natural to let the model validation constraint overrule the first setting.

For underconstrained models, it is difficult to automatically determine useful default values for feature parameters. Heuristics may be applied, e.g. taking into account other parameter values of the feature, or of the model. It is required that values are chosen in such a way that all validation constraints are satisfied. This requires further investigation.

Dependency graphs are related to *dimension graphs*, which represent information about model dimensions. A dependency graph may be analyzed to obtain a consistent dimensioning scheme; work on this has been reported by Shah and Yan [1996]. Further research is required in order to automatically derive a model dimensioning scheme from a dependency graph.

Part IV Results and conclusions



9

Results

This chapter presents results of the research, demonstrating not only the usability of the multiple-view feature modeling and validation concepts, but also their successful implementation. Firstly, it gives an overview of the functionality of our multiple-view feature modeling system SPIFF. Then, it presents a demonstration multiple-view modeling session with SPIFF.

9.1 SPIFF

When SPIFF is launched, the main window appears, see Figure 9.1(a). From the main window, different actions can be undertaken. The user can open a product to work on. Alternatively, he can open a feature or constraint library for editing. He can also run a demo session which shows how to use SPIFF, or he can start the on-line help tool for explanation of the SPIFF functions. Finally, he can use the system testing tools. When a product has been selected for editing (b), the user must select a view on this product for editing (c).

After a view has been selected, the user is provided with a control panel window for the view, see Figure 9.2; from this panel he can perform modeling actions such as adding or removing feature instances, or changing feature parameters. Furthermore, he can open a feature graph window depicting the structure of the feature model, including feature attachment relations and model validation constraints. Also, he can open one or more independent camera windows in which various visualization methods can be applied, for example, to visualize feature closure faces.

The example model shown in this figure is based on a description of an airplane part called Gehäuse Rohteil from MBB Deutsche Aerospace.

From the view selection window, other views on the product can be opened; for each of them a control panel is created. The view that has the edit status is indicated in all panels; the user in the edit view may release the edit status, after which it can be obtained in another view.



Figure 9.1: SPIFF main window (a), model selection window (b), and view selection window (c).

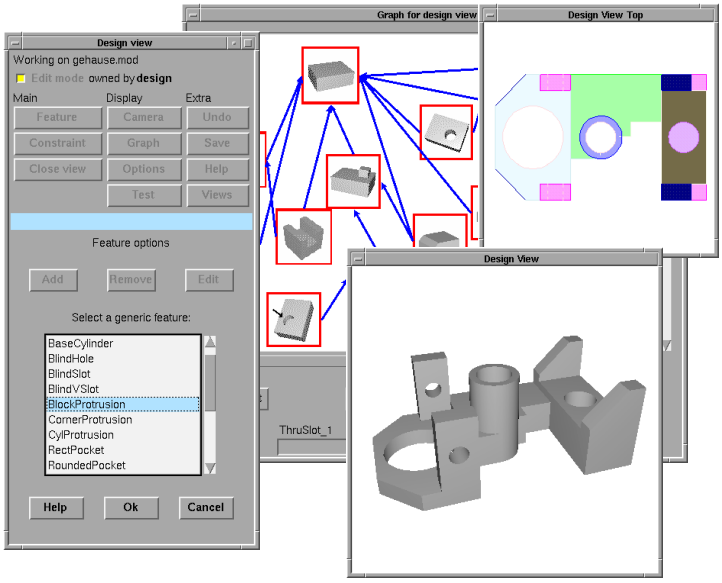


Figure 9.2: Panel, graph and camera windows for a view.

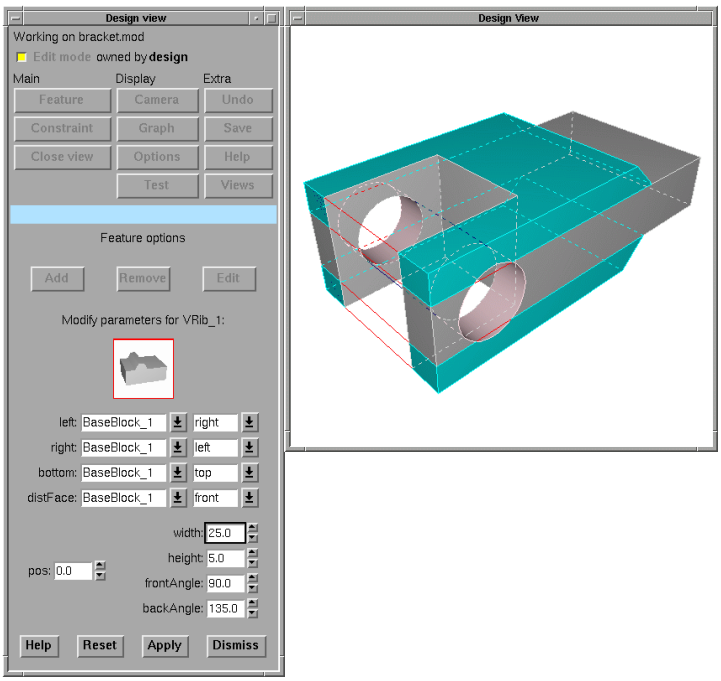


Figure 9.3: The designer changes the widths of the ribs.

9.2 Multiple-view modeling session

A demonstration modeling session with SPIFF shows the usability of the concepts presented in this thesis. In the session, a part is simultaneously engineered from different disciplines. The demonstration shows how feature modeling actions are performed, and how the validity of the model is maintained across multiple views. Throughout the demonstration, different feature visualization techniques are applied, including visualization of not on boundary faces and projected line drawings.

The session is done with the bracket part encountered earlier in Chapter 4. In Figure 9.3 the design view is shown; also, a manufacturing view has been opened for manufacturability analysis. The designer changes the widths of the two ribs. In the manufacturing view, the widths of the steps are adjusted accordingly via the link constraints between the views, see Figure 9.4. A manufacturability requirement for the bracket is that the two bracket extensions should not be too narrow, to prevent vibration damage when milling the slot. Therefore, the manufacturing engineer adds minimum dimension constraints between the through-slot and the sides of the stock.

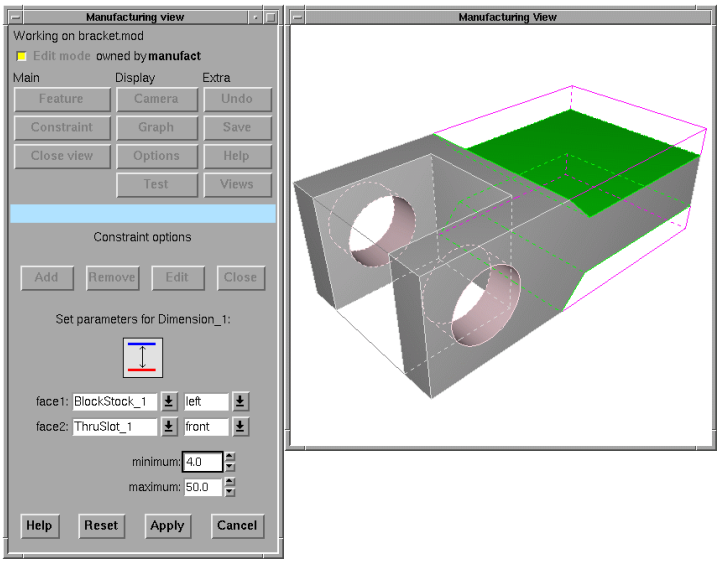


Figure 9.4: The manufacturing engineer adds minimum dimension constraints.

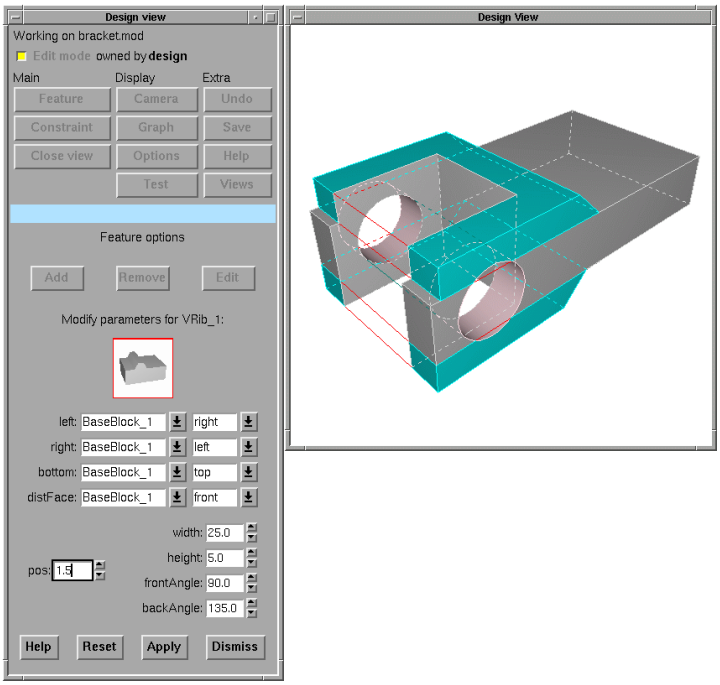


Figure 9.5: The designer changes the position of the ribs.

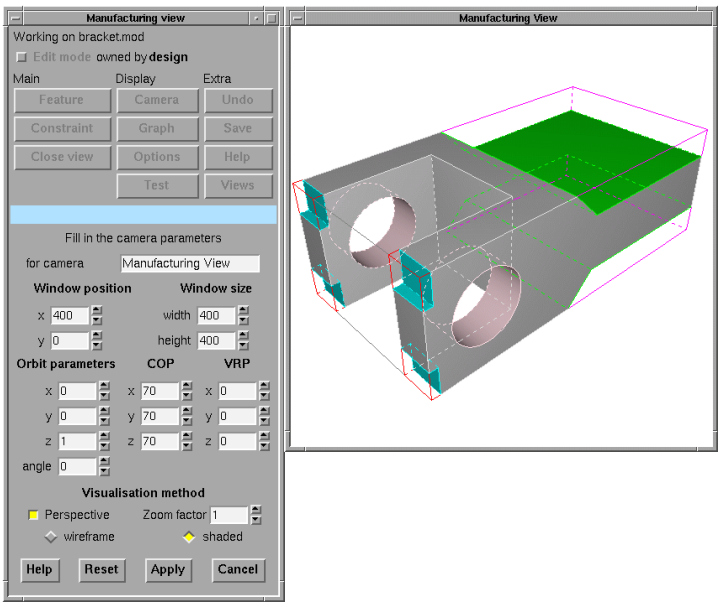


Figure 9.6: In the manufacturing view, new steps have been identified.

To enhance the functionality of the part, the designer changes the positions of the two ribs, see Figure 9.5. This causes a change of product topology, which is detected as a link constraint conflict by the modeling system; the manufacturing view is reopened. In the reopened manufacturing view, see Figure 9.6, new steps have been identified.

The bracket has to be used in an assembly with other parts; therefore, an assembly view is opened. After opening, the assembly engineer instantiates two blind slots on the sides of the bracket, to be used as insertion slots, see Figure 9.7.

Next, he instantiates two rounded pockets on one side of the bracket, to be used for fixing the bracket with two clamps, see Figure 9.8. The second rounded pocket is positioned incorrectly, in such a way that it intersects the nearby blind slot. This is a violation of the on boundary semantic constraint of the blind slot end face, which is signaled. The assembly engineer adjusts the position of the second rounded pocket, after which all constraints are satisfied.

To align the rounded pockets with the bracket base block, the assembly engineer adds two coplanarity constraints between the planar sides of the rounded pockets and the top face of the base block, see Figure 9.9. These coplanarity constraints are depicted in the feature graph window by constraint icons. This addition causes the model to become geometrically overconstrained, since the position and orientation of the rounded pockets were already fully specified. The

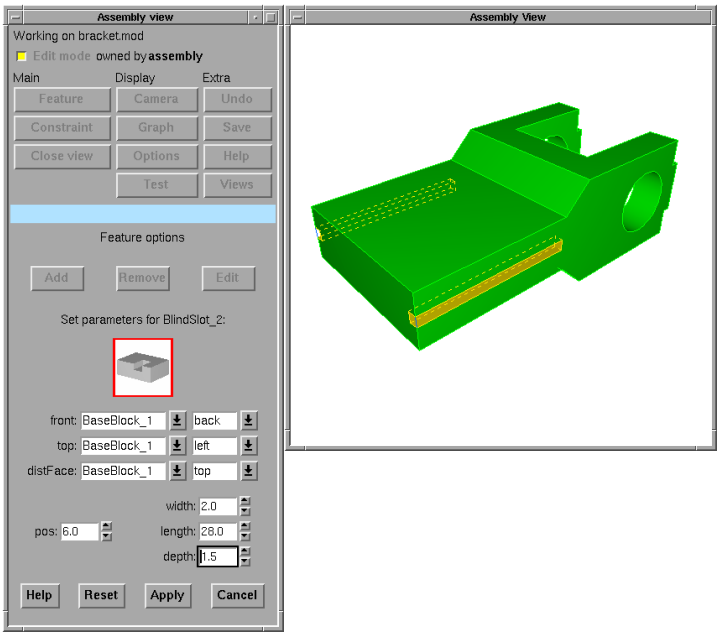


Figure 9.7: The assembly engineer adds blind slots.

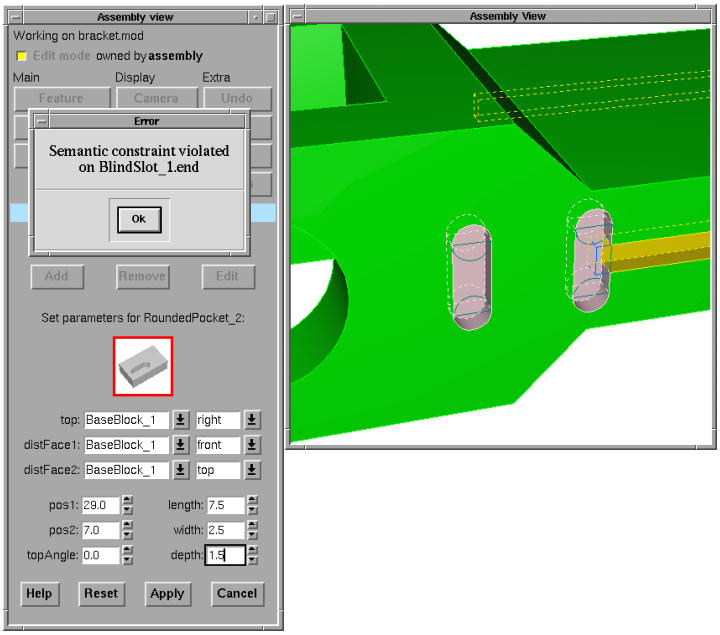


Figure 9.8: The assembly engineer adds rounded pockets.

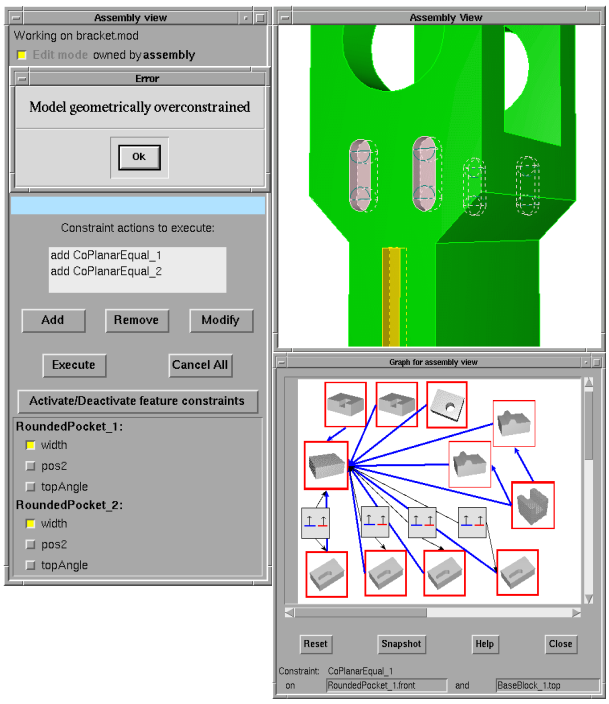


Figure 9.9: The assembly engineer aligns the rounded pockets with the base block top face.

overconstrained situation is signaled, and the involved constraints, including the dimensions, are listed. The engineer de-activates for each rounded pocket the involved position and orientation constraints. Finally, he instantiates two more rounded pockets on the other side of the bracket.

To increase the stiffness of the bracket, the designer wishes to replace the slot by two narrower slots. He first reduces the width of the slot, and positions it closer to the side, see Figure 9.10. This causes a violation of one of the previously added minimum dimension constraint in the manufacturing view, which is signaled; after discussion, the designer and manufacturing engineer decide to cancel the modification. The designer then only changes the width of the slot, after which all constraints are satisfied.

Finally, the designer adds a second slot, leading to the product depicted in Figure 9.11.

This concludes the demonstration session. It has shown how a product can be changed from different views, how view-specific product requirements can be specified by constraints of different domains, and how these are maintained under product modifications.

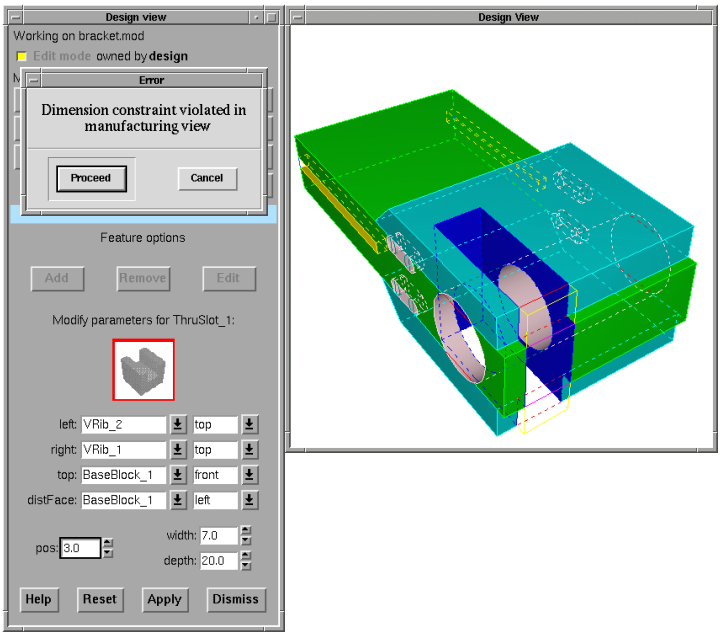


Figure 9.10: The designer changes the position and the width of the slot.

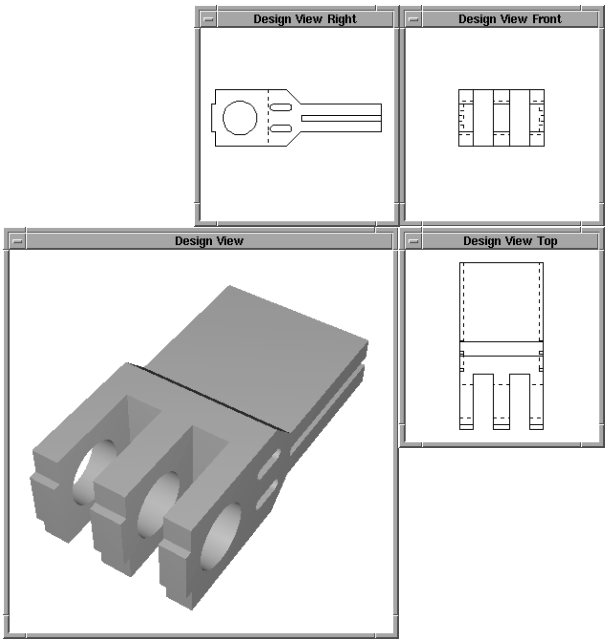


Figure 9.11: The final product, after the designer has added a second slot.

10

Conclusions and future work

This chapter gives concluding remarks about the research presented, and ends with some directions for future research.

10.1 Conclusions

A scheme has been presented for feature validation in a multiple-feature view environment. The validation scheme is declarative, separating validity specification and maintenance. This separation enables flexible and expressive feature validity specifications, combined with powerful feature validity maintenance. The validation scheme ensures that only meaningful products can be specified, and that their validity is maintained under product modifications from different views.

For feature validity specification, various constraint types have been identified. They are used in feature class specifications at the generic level, and as model validation constraints at the instance level. The constraint types are of three different domains: geometric, algebraic and topologic. New constraint types can be created by combining existing types. The constraints of these types provide a good means to express the meaning of the features.

Feature and constraint classes are specified in an interpreted object-oriented programming language, possibly at run time. This allows generic and flexible feature validity specification.

A persistent feature validity maintenance scheme has been developed which is based on constraint management, integrated with management of a cellular geometry representation. The constraint management is separated into two levels: constraint storage and constraint satisfaction. Firstly, the Constraint Manager stores in a constraint graph the set of constraints that result from modeling actions; the constraint graph forms a symbolic representation of the feature models. Secondly, the Constraint Manager ensures satisfaction of all constraints by

deploying several constraint solvers that are each specialized for a particular domain.

The Constraint Manager maintains a map that provides a coupling between the two levels, and it communicates solving results between the different constraint solvers. The advantages of the two-level separation are that the interdependence of the constraints is handled by the Constraint Manager, while the constraints are efficiently satisfied by the solvers. The separation further enables optimization of the maintenance process; for this, the Constraint Manager analyzes the constraint graph to determine subgraphs that need to be solved after a feature model modification. Furthermore, the separation gives a modularity that enables integration of solvers specialized for other domains.

For constraints from the algebraic domain, the SkyBlue local propagation solver is used. Its main advantage is that it provides a constraint priority scheme; the Constraint Manager uses the priority scheme to propagate feature parameter changes in such a way that the model is minimally disturbed. A second advantage of SkyBlue is that it can be easily extended with new types of constraints.

For constraints from the geometric domain, a solver has been developed based on degrees of freedom analysis. It employs 3D geometric reasoning, which has the advantages of ensuring geometrically meaningful solutions, and of supporting the analysis of over- and underconstrained models. Degrees of freedom analysis had been devised earlier to describe the kinematic behavior of mechanical linkages; to be useful in the feature modeling domain, the approach has been extended successfully to deal with considerably more complex combinations of constraints. Newly developed extensions include DOF ignorance for locus intersections, and restructuring of constraint loops. With these extensions, powerful 3D geometric constraint solving is achieved.

For the analysis of over- and underconstrained geometric models, the concept of dependency graph has been developed, which represents the dependencies of geometric variables as determined by the solver. A dependency graph is used to identify the set of involved constraints for over- and underconstrained models; after modification of this set, the dependency graph is used for incremental re-solving of the constraints.

For constraints from the topologic domain, the cellular geometry representation is queried to check that the constraints are satisfied [De Kraker 1997]; the cellular geometry representation is very well-suited for this. Detection of violations of topologic feature properties caused by feature interaction is an important aspect of feature validation; therefore, its successful incorporation in the presented feature validation scheme has considerably enhanced this scheme.

The constraint management scheme allows fairly complex models to be solved in reasonable time. This is achieved in two ways. Firstly, the constraint graph is analyzed to determine which regions of the model need to be re-solved

after a modification. Secondly, the employed SkyBlue and degrees of freedom analysis solvers are incremental by nature, allowing efficient solving in their respective domains.

When multiple feature views are open, modifications of the feature model in a view are propagated to the other views. To propagate feature parameter changes, the feature models of the views are connected by link constraints that couple degrees of freedom of feature elements. This has the advantage that the views are coupled in a generic way, i.e. without explicitly coupling feature parameters, thus requiring no assumptions about which type of parameters can be coupled. A feature parameter change may cause a change of product topology, which is detected as a violation of link constraints; to propagate such a change, the other views are reopened [De Kraker 1997].

When changes are propagated, all feature models are validated. In this process, conflicts between constraints from different views may occur. A view priority scheme is used to determine whether to overrule a violated constraint, or to cancel the edit action. This scheme provides a simple but effective way of dealing with multiple-view conflicts.

The multiple-view validation scheme has been verified by implementing it in the multiple-view feature modeling system SPIFF. The system has a layered architecture, providing a clean functional separation. Each layer is implemented in a programming language that suits the layer's requirements. The layered architecture makes the system flexible and extensible, but involves much communication between the layers.

10.2 Future work

The implemented degrees of freedom analysis solver currently only supports planar surfaces. This is overcome in the constraint mapping process; nonplanar surfaces are represented by characteristic points, which allows geometric constraints on predefined points of nonplanar surfaces. This has been implemented for cylindrical surfaces; other surfaces, e.g. cones or B-spline surfaces, require further investigation.

In the current implementation, only coplanar face attachments are supported; future work should concern oblique and nonplanar attachments, and include edge attachments.

In the constraint management scheme, each solver handles the constraints belonging to its specific domain; the interdependence of the constraints is handled by propagating the values of constraint variables between the constraints

of different domains via the map. This scheme is not yet fully exploited in the current implementation; the algebraic and the geometric solvers solve their primitive constraint graphs completely, before returning the results to the Constraint Manager. This gives a loose coupling of the solver domains. A stronger coupling between the algebraic and the geometric solver is required, alternating between solving some of the algebraic constraints and solving some of the geometric constraints.

By using information about dimensional dependencies in the geometric constraints, it may be possible to automatically derive algebraic relationships between dimensions that can be used directly by the algebraic solver. In such a way, not only value information, but also structural information about dimension dependencies is communicated between different constraint domains; this would reduce the number of alternate calls to the solvers, and, more generally, would enhance the constraint management scheme.

The SkyBlue solver is a local propagation constraint solver, which cannot handle circular dependencies. An iterative solver should be incorporated, used to solve the circular dependencies.

Topologic constraints are currently only checked for violations, which cannot be done until the product geometry has been determined. This passive checking may be replaced by active topologic constraint solving, particularly in an interactive graphical environment with direct manipulation, where a repeated re-evaluation of the product geometry for each manipulation step is not desirable. Such a solver could specialize, for example, in topologic relationships between 2D polygons, such as containment and overlap. The constraint management should be extended with a topologic solver to enable active topologic constraint solving and early detection of violations.

On-going research is performed to extend the validation scheme with feature interaction management, in which, after detection of a feature interaction, an explanation of its causes is given, and reaction mechanisms are provided to the user to overcome the invalidity situation [Bidarra, Dohmen and Bronsvoort 1997]. Feature interaction management would further enhance the presented feature validation scheme.

The view priority scheme could be extended with a constraint priority scheme, to overrule violated constraints of lower priority only. This would allow a more fine-tuned way of handling constraint conflicts between views.

The implemented feature views are based on features that represent geometry; future research should investigate other, enhanced features, e.g. for conceptual design. This could require the incorporation of other constraint domains in the feature validation scheme.

Bibliography

Arbab, F. and Wang, B. [1991], A geometric constraint management system in OAR, *in* P. J. W. ten Hagen and P. J. Veerkamp, eds, 'Intelligent CAD Systems III, Practical experience and valuation', Springer, pp. 205–231.

Bidarra, R. and Bronsvoort, W. F. [1996], Towards classification and automatic detection of feature interactions, *in* D. Roller, ed., 'Proceedings of the 29th International Symposium on Automotive Technology and Automation', Automotive Automation, pp. 99–108.

Bidarra, R., De Kraker, K. J. and Bronsvoort, W. F. [1997], 'Representation and management of feature information in a cellular model', *Computer-Aided Design* . To be published.

Bidarra, R., Dohmen, M. and Bronsvoort, W. F. [1997], Automatic detection of interactions in feature models, *in* 'CD-ROM Proceedings of the 1997 ASME Design Engineering Technical Conferences and Computers in Engineering Conference', ASME.

Bidarra, R. and Teixeira, J. C. [1994], A semantic framework for flexible feature validity specification and assessment, *in* 'Proceedings of the 1994 ASME International Computers in Engineering Conference', Vol. 1, ASME, pp. 151–158.

Borning, A. [1981], 'The programming language aspects of ThingLab, a constraint-oriented simulation laboratory', *ACM Transactions on Programming Languages and Systems* **3**(4), 353–387.

Borning, A. [1986], Defining constraints graphically, *in* 'CHI'86 Proceedings', ACM Press, pp. 137–143.

Borning, A., Anderson, R. and Freeman-Benson, B. [1996], Indigo: a local propagation algorithm for inequality constraints, *in* 'Proceedings of the Ninth Annual Symposium on User Interface and Software Technology', ACM Press, pp. 129–136.

Borning, A., Duisberg, R., Freeman-Benson, B., Kramer, A. and Woolf, M. [1987], Constraint hierarchies, *in* 'OOPSLA '87 Proceedings', ACM Press, pp. 48–60.

Bouma, W., Fudos, I., Hoffmann, C., Cai, J. and Paige, R. [1995], 'A geometric constraint solver', *Computer-Aided Design* **27**(6), 487–501.

Bowen, J. and Bahler, D. [1992], 'Frames, quantification, perspectives, and negotiation in constraint networks for life-cycle engineering', *Artificial Intelligence in Engineering* **7**, 199–226.

Bronsvoort, W. F. and Jansen, F. W. [1993], 'Feature modelling and conversion – Key concepts to concurrent engineering', *Computers in Industry* **21**(1), 61–86.

Brüderlin, B. [1986], Constructing three-dimensional geometric objects defined by constraints, *in* F. Crow and S. M. Pizer, eds, 'Proceedings 1986 Workshop on Interactive 3D Graphics', ACM Press, pp. 111–128.

Brunetti, G., De Martino, T., Elter, H. and Falcidieno, B. [1996], Modeling shape and semantics through an intermediate model, *in* D. Roller, ed., 'Proceedings of the 29th International Symposium on Automotive Technology and Automation', Automotive Automation, pp. 71–81.

Brunetti, G., De Martino, T., Falcidieno, B. and Haßinger, S. [1995], A relational model for interactive manipulation of form features based on algebraic geometry, *in* C. Hoffmann and J. Rossignac, eds, 'Solid Modeling '95, Third Symposium on Solid Modeling and Applications', ACM Press, pp. 95–103.

Brunetti, G., Ovtcharova, J. and Vieira, A. [1996], A proposal for a feature description language, *in* D. Roller, ed., 'Proceedings of the 29th International Symposium on Automotive Technology and Automation', Automotive Automation, pp. 117–124.

Chen, X. and Hoffmann, C. M. [1995*a*], 'On editability of feature-based design', *Computer-Aided Design* **27**(12), 905–914.

Chen, X. and Hoffmann, C. M. [1995*b*], 'Towards feature attachment', *Computer-Aided Design* **27**(9), 695–702.

Chung, J. C. H. and Schussel, M. D. [1989], Comparison of variational and parametric design, *in* 'Proceedings Autofact '89', Society of Manufacturing Engineers, pp. 5–27–5–44.

Clément, A. and Rivière, A. [1993], Tolerancing versus nominal modelling in next generation CAD/CAM system, *in* 'CIRP Seminar on Computer Aided Tolerancing', pp. 97–113.

Davis, E. [1987], 'Constraint propagation with interval labels', *Artificial Intelligence* **32**, 281–331.

Dohmen, M. [1995], 'A survey of constraint satisfaction techniques for geometric modeling', *Computers & Graphics* **19**(6), 831–845.

Dohmen, M., De Kraker, K. J. and Bronsvoort, W. F. [1995], Constraint techniques for feature validation, *in* T. Tomiyama, M. Mäntylä and S. Finger, eds, 'Preprints of the First IFIP WG 5.2 Workshop on Knowledge Intensive CAD', IFIP, pp. 179–192.

Dohmen, M., De Kraker, K. J. and Bronsvoort, W. F. [1996], Feature validation in a multiple-view modeling system, *in* J. McCarthy, ed., 'CD-ROM Proceedings of the 1996 ASME Design Engineering Technical Conferences and Computers in Engineering Conference', ASME.

Van Emmerik, M. J. G. M. [1989], 'Creation and modification of parametrized solid models by graphical interaction', *Computers & Graphics* **13**(1), 71–76.

Van Emmerik, M. J. G. M. [1991], 'Interactive design of 3D models with geometric constraints', *The Visual Computer* **7**(5/6), 309–325.

Fa, M., Fernando, T. and Dew, P. M. [1993], Interactive constraint-based solid modelling using allowable motion, *in* J. R. Rossignac, J. Turner and G. Allen, eds, 'Proceedings Second Symposium on Solid Modeling and CAD/CAM Applications', ACM Press, pp. 243–252.

Freuder, E. C. [1982], 'A sufficient condition for backtrack-free search', *Journal of the ACM* **29**(1), 24–32.

Fudos, I. and Hoffmann, C. M. [1997], 'A graph-constructive approach to solving systems of geometric constraints', *ACM Transactions on Graphics* **16**(2), 179–216.

Gleicher, M. [1992], Integrating constraints and direct manipulation, *in* 'Proceedings 1992 Symposium on Interactive 3D Graphics', ACM Press, pp. 171–174.

Gleicher, M. and Witkin, A. [1991], Differential manipulation, *in* 'Graphics Interface '91', Canadian Information Processing Society, pp. 61–67.

Gorlen, K. E., Orlow, S. M. and Plexico, P. S. [1991], *Data abstraction and object-oriented programming in C++*, John Wiley & Sons, Inc.

Henderson, D. L. [1996], Constraint management for collaborative design, PhD thesis, Purdue University, USA.

Hoffmann, C. M. and Juan-Arinyo, R. [1997], 'On user-defined features'. Submitted for publication.

Van Holland, W. and Bronsvoort, W. F. [1997], Assembly features and sequence planning, *in* M. Pratt, R. D. Sriram and M. J. Wozny, eds, 'Product Modeling for Computer Integrated Design and Manufacture', Chapman & Hall, pp. 275–284.

Hsu, C. and Brüderlin, B. D. [1997], A graph based degrees of freedom analysis algorithm to solve geometric constraint problems, *in* W. Strasser, R. Klein and R. Rau, eds, 'Geometric Modeling: Theory and Practice', Springer.

Hyvönen, E. [1992], 'Constraint reasoning based on interval arithmetic: the tolerance propagation approach', *Artificial Intelligence* **58**(1-3), 71–112.

Juster, N. P. [1992], 'Modelling and representation of dimensions and tolerances: a survey', *Computer-Aided Design* **24**(1), 3–17.

Keirouz, W., Pabon, J. and Young, R. [1992], 'Integrating parametric geometry, features, and variational modeling for conceptual design', *International Journal of Systems Automation: Research and Applications (SARA)* **2**, 17–36.

Keirouz, W. T., Kramer, G. A. and Pabon, J. [1993], Exploiting constraint dependency information for debugging and explanation, *in* 'First Workshop on Principles and Practice of Constraint Programming'.

De Kleer, J. [1986], 'An assumption-based TMS', *Artificial Intelligence* **28**, 127–162.

De Kraker, K. J. [1997], Feature conversion for concurrent engineering, PhD thesis, Delft University of Technology, The Netherlands.

De Kraker, K. J., Dohmen, M. and Bronsvoort, W. F. [1994], High-level constraints in interactive CSG modelling, *in* A. Middleditch and A. A. G. Requicha, eds, 'Proceedings CSG 94, Set-theoretic Solid Modelling: Techniques and Applications', Information Geometers, pp. 275–289.

De Kraker, K. J., Dohmen, M. and Bronsvoort, W. F. [1995], Multiple-way feature conversion to support concurrent engineering, *in* C. Hoffmann and J. Rossignac, eds, 'Solid Modeling '95, Third Symposium on Solid Modeling and Applications', ACM Press, pp. 105 – 114.

De Kraker, K. J., Dohmen, M. and Bronsvoort, W. F. [1997*a*], Feature validation and conversion, *in* D. Roller and P. Brunet, eds, 'CAD Systems Development', Springer, pp. 121–142.

De Kraker, K. J., Dohmen, M. and Bronsvoort, W. F. [1997*b*], Maintaining multiple views in feature modeling, *in* C. Hoffmann and W. Bronsvoort, eds, 'Solid Modeling '97, Fourth Symposium on Solid Modeling and Applications', ACM Press, pp. 123–130.

Kramer, G. A. [1992*a*], 'A geometric constraint engine', *Artificial Intelligence* **58**(1-3), 327–360.

Kramer, G. A. [1992*b*], *Solving geometric constraint systems: a case study in kinematics*, The MIT Press.

Kripac, J. [1995], A mechanism for persistently naming topological entities in history-based parametric solid models - Topological ID System, *in* C. Hoffmann and J. Rossignac, eds, 'Solid Modeling '95, Third Symposium on Solid Modeling and Applications', ACM Press, pp. 21–30.

Kuiper, A. J. [1995], Solving geometric constraints with degrees of freedom analysis, Master's thesis, Delft University of Technology, The Netherlands.

Laakko, T. and Mäntylä, M. [1993], 'Feature modelling by incremental feature recognition', *Computer-Aided Design* **25**(8), 479–492.

Lagus, K., Laakko, T. and Mäntylä, M. [1994], ICONF - an incremental constraint facility in a feature modelling system, *in* 'Proceedings IFIP WG 5.3 International Conference on Feature Modelling and Recognition in Advanced CAD/CAM Systems', Vol. 1, IFIP, pp. 161–177.

Lamounier, E. [1996], An Incremental Constraint-based Approach to Support Engineering Design, PhD thesis, University of Leeds, United Kingdom.

Leler, W. [1988], *Constraint Programming Languages, Their Specification and Generation*, Addison-Wesley Publishing Company.

Light, R. and Gossard, D. [1982], 'Modification of geometric models through variational geometry', *Computer-Aided Design* **14**(4), 209–214.

Lin, V. C., Gossard, D. C. and Light, R. A. [1981], 'Variational geometry in computer-aided design', *Computer Graphics* **15**(3), 171–177.

Mandorli, F., Cugini, U., Otto, H. E. and Kimura, F. [1997], Modeling with self validation features, *in* C. Hoffmann and W. Bronsvoort, eds, 'Solid Modeling '97, Fourth Symposium on Solid Modeling and Applications', ACM Press, pp. 88–96.

Mehlhorn, K. and Näher, S. [1995], 'LEDA, a platform for combinatorial and geometric computing', *Communications of the ACM* **38**(1), 96–102.

Molina, A., Al-Ashaab, A. H., Ellis, T. I. A. and Young, R. I. M. [1995], 'A review of computer-aided simultaneous engineering systems', *Research in Engineering Design* **7**, 38–63.

Niemeyer, P. and Peck, J. [1996], *Exploring Java*, O'Reilly & Associates.

Noort, A. [1997], Solving over-constrained geometric models, Master's thesis, Delft University of Technology, The Netherlands.

Noort, A., Dohmen, M. and Bronsvoort, W. F. [1997], Solving over- and underconstrained geometric models, *in* 'Proceedings of the Workshop on Geometric Constraint Solving & Applications'. To be published.

Van Overveld, C. W. A. M. [1993], '30 years after Sketchpad: relaxation of geometric constraints revisited', *CWI Quarterly* (4).

Owen, J. C. [1991], Algebraic solution for geometry from dimensional constraints, *in* J. Rossignac and J. Turner, eds, 'Proceedings Symposium on Solid Modeling Foundations and CAD/CAM Applications', ACM Press, pp. 397–407.

Parametric [1993], *Pro/ENGINEER Modeling User's Guide, Version 12*, Parametric Technology Corporation, Waltham, Maryland, USA.

Peeters, E. A. J. [1993], Design and implementation of an object-oriented, interactive animation system, *in* C. Mingins, W. Haebich, J. Potter and B. Meyer, eds, 'Technology of Object-Oriented Languages and Systems, TOOLS 12 & 9', Prentice Hall, pp. 255–267.

Peeters, E. A. J. [1995], Design of an Object-Oriented, Interactive Animation System, PhD thesis, Eindhoven University of Technology, The Netherlands.

Rocheleau, D. N. and Lee, K. [1987], 'System for interactive assembly modelling', *Computer-Aided Design* **19**(2), 65–72.

Rossignac, J. R. [1986], Constraints in constructive solid geometry, *in* F. Crow and S. M. Pizer, eds, 'Proceedings 1986 Workshop on Interactive 3D Graphics', ACM Press, pp. 93–110.

Salomons, O. W. [1995], Computer support in the design of mechanical products, constraint specification and satisfaction in feature based design for manufacturing, PhD thesis, University of Twente, The Netherlands.

Sannella, M. [1992], The SkyBlue constraint solver, Technical Report 92-07-02, Dept. of Computer Science and Engineering, University of Washington, USA.

Sannella, M. [1993], The SkyBlue constraint solver and its applications, *in* 'First Workshop on Principles and Practice of Constraint Programming'.

Serrano, D. and Gossard, D. [1988], Constraint management in MCAE, *in* J. S. Gero, ed., 'Artificial Intelligence in Engineering: Design', Elsevier, pp. 217–240.

Shah, J. J. [1991], 'Conceptual development of form features and feature modelers', *Research in Engineering Design* **2**, 93–108.

Shah, J. J., Ali, A. and Rogers, M. T. [1994], Investigation of declarative feature modeling, *in* 'Proceedings of the 1994 ASME International Computers in Engineering Conference', Vol. 1, ASME, pp. 1–11.

Shah, J. J., Balakrishnan, G., Rogers, M. and Urban, S. D. [1994], Comparative study of procedural and declarative feature based geometric modeling, *in* 'Proceedings IFIP WG 5.3 International Conference on Feature Modelling and Recognition in Advanced CAD/CAM Systems', Vol. 2, IFIP, pp. 647–671.

Shah, J. J. and Mäntylä, M. [1995], *Parametric and feature-based CAD/CAM*, John Wiley & Sons, Inc.

Shah, J. J., Rogers, M. T., Sreevalsan, P. C., Hsiao, D. W., Mathew, A., Bhatnagar, A., Liou, B. B. and Miller, D. W. [1990], The A.S.U. features testbed: an overview, *in* G. L. Kinzel and S. M. Rohde, eds, 'Proceedings of the 1990 ASME International Computers in Engineering Conference and Exposition', Vol. 1, ASME, pp. 233–241.

Shah, J. J. and Yan, Y. [1996], Representation and mapping of geometric dimensions from design to manufacturing, *in* J. McCarthy, ed., 'CD-ROM Proceedings of the 1996 ASME Design Engineering Technical Conferences and Computers in Engineering Conference', ASME.

Sohrt, W. and Brüderlin, B. D. [1991], Interaction with constraints in 3D modeling, *in* J. Rossignac and J. Turner, eds, 'Proceedings Symposium on Solid Modeling Foundations and CAD/CAM Applications', ACM Press, pp. 387–396.

Spatial [1996], *Geometric Modeler Programmers Reference, Version 2.1*, Spatial Technology, Inc, Boulder, Colorado, USA.

Sreevalsan, P., Rogers, M. and Shah, J. [1990], Object oriented tools for feature database design, *in* '1990 ASME Winter Annual Meeting', ASME.

Sutherland, I. E. [1963], Sketchpad – a man-machine graphical communication system, *in* 'Proceedings Spring Joint Computer Conference', pp. 329–346.

Suzuki, H., Ando, H. and Kimura, F. [1990], 'Geometric constraints and reasoning for geometrical CAD systems', *Computers & Graphics* **14**(2), 211–224.

Thornton, A. C. and Johnson, A. [1993], Constraint specification and satisfaction in embodiment design, *in* 'Proceedings International Conference on Engineering Design ICED 93', Heurista, pp. 1319–1326.

Tsang, E. [1993], *Foundations of constraint satisfaction*, Academic Press.

Tyugu, E. and Uustalu, T. [1993], Higher-order functional constraint networks, *in* B. Mayoh, ed., 'Proceedings NATO ASI on Constraint Programming', LNCS ASI number 131, Springer, pp. 123–147.

Veltkamp, R. C. and Arbab, F. [1992], Geometric constraint propagation with quantum labels, *in* B. Falcidieno, I. Herman and C. Pienovi, eds, 'Computer Graphics and Mathematics, Proceedings of the Eurographics Workshop on Computer Graphics and Mathematics', Springer, pp. 211–228.

Vergouwe, B. [1996], A graphical user interface for SPIFF, Master's thesis, Delft University of Technology, The Netherlands. In Dutch.

Versluis, J. W., De Kraker, K. J., Bronsvoort, W. F. and Seebregts, K. [1997], Feature visualization, *in* 'CD-ROM Proceedings of the 1997 ASME Design Engineering Technical Conferences and Computers in Engineering Conference', ASME.

Vieira, A. S. [1995], Consistency management in feature-based parametric design, *in* R. Gadh, ed., 'Proceedings of the 1995 Design Engineering Technical Conferences', Vol. 2, ASME, pp. 977–987.

Watterson, B. [1995], *The Calvin and Hobbes Tenth Anniversary Book*, Andrews and McMeel.

Index

A

ACIS, 55
action analysis, 106
algebraic constraint, 19, 63, 98
AlgebraicConstraint, 100
Align, 59
AngleParameter, 100
application, 1
Assumption-based Truth Maintenance, 17
ASU Features Testbed, 39, 42
ATMS, 17
attach constraint, 59
attribute, 53

B

basis geom, 106
Briar, 23

C

canonical feature element, 53
canonical shape, 53
canonical shape model, 54
cellular model, 54
coincident, 105
compound constraint, 67
compound feature, 53
concurrent engineering, 2
constraint, 11
    algebraic, 19, 63, 98
    attach, 59
    dimension, 63
    engineering, 34
    fix global location, 79
    fix value, 79
    geometric, 25, 62, 112
    semantic, 43, 61
    shape, 78
    topologic, 61
constraint activation, 85
constraint arity, 11
constraint conflict, 70
constraint cycle, 14
constraint direction, 11
constraint graph, 12, 54, 79
    implementation, 81
constraint graph pruning, 87
constraint inference, 14
constraint instance, 77
constraint interaction, 91
constraint library, 53
constraint loop, 14
constraint maintenance, 82
constraint management, 77
Constraint Manager, 55, 77
constraint parameter, 11
constraint priority, 97
constraint propagation, 13
constraint satisfaction, 11
constraint satisfaction optimization problem, 13
constraint satisfaction problem, 11
constraint solving, 11
constraint strength, 97
constraint type, 59
constraint variable, 11
constraint-based feature validation, 3
Contact, 59
CSOP, 13
CSP, 11

D

declarative feature validation, 41

default rules, 29
degrees of freedom analysis, 103
dependency graph, 117
design by features, 2
design for X, 2
dimension constraint, 63
dimension graph, 127
dimension limit, 37
dimensioning scheme, 64
direct parameterization, 35
Division, 100
DOF, 103
DOF constraint, 112
DOF ignorance, 110

E

edit a view, 50
engineering constraint, 34
Equal, 100
exhaustive search, 17
explicit shape parameter, 78
EXTDesign, 41
extended locus analysis, 125

F

Face, 122
feature, 1
feature attachment, 37
feature class, 2
feature constituent, 53
feature conversion, 4
feature element, 2
Feature Geometry Manager, 55
feature instance, 2
feature interaction, 37
feature intersection, 37
feature library, 53
Feature Manager, 55
feature model, 2
feature nature
    additive, 53
    subtractive, 53
feature parameter, 2
feature property, 37
feature recognition, 2
feature shape, 53
feature shape validation, 37
feature specification language, 38, 51
feature topology entity, 2
feature validation, 3, 37
feature validation constraint, 63
feature validity conditions, 3, 37, 59
feature validity maintenance, 3, 67
feature validity specification, 3, 64
finite domain CSP, 11, 14
fix global location constraint, 79
fix value constraint, 79
FROOM, 42

G

Gehäuse Rohteil, 131
generic feature, 2
geom, 104
geometric constraint, 25, 62, 112
geometric constraint satisfaction
    graph-based, 29
    local, 25
    operational, 26
    rule-based, 28
geometric domain CSP, 25
geometric reasoning, 28
GeometricVariable, 122
geometry-related constraint, 86
GeoNode, 27, 39
graph matching, 23
graph-based geometric constraint satisfaction, 29
graph-based techniques, 23
grounded geom, 106

H

higher-level constraint, 67
hill-climbing, 17
history-based approach, 52
hyper-edges, 12
hyper-graph, 12

I

implicit shape parameter, 78
in-line, 105
in-plane, 105
incremental constraint satisfaction, 27
indirect parameterization, 35
infinite domain CSP, 11, 18
interval CSP, 21
interval inference, 14

J

JAVA, 53
joint, 103, 105

K

k-consistency, 15

L

label, 11
label inference, 14
layer, 56
LEDA, 55, 81
link constraint, 51, 67
local geometric constraint satisfaction, 25
location limit, 37
locus, 109
locus analysis, 108
locus intersection, 109
LOOKS, 53, 64
loop processing, 111

M

macrogeom, 105
map, 77
map-coloring problem, 12
marker, 105
MBB Deutsche Aerospace, 131
mechanical linkage, 103
Minus, 100
model dimension, 34
model validation constraint, 63
model variable, 34

N

Newton-Raphson iteration, 22
NIH, 55
NotOnBoundary, 62
NoVolumetricIntersection, 61
numeric techniques, 22

O

OnBoundary, 62
open a view, 50
operational geometric constraint satisfaction, 26
overconstrained, 11, 119

P

parallel, 105
Parameter, 100
parametric modeling, 34
parent feature, 39
partial constraint satisfaction problem, 13
PCSP, 13
persistent topologic naming, 52
plan fragment, 106
Plus, 100
Plus3, 66
primitive constraint graph, 77
problem reduction techniques, 16
procedural feature validation, 39
product development, 1
product life cycle phase, 1
product model, 51
Pro/ENGINEER, 3
propagation of degrees of freedom, 20
propagation of known states, 13, 20

Q

quantum approach, 26

R

relation
    (a)symmetrical, 11
    (non)hierarchical, 11
relaxation, 22

reopen a view, 51, 71
rigid joint, 105
RoundedPocket, 64
rule-based geometric constraint satisfaction, 28

S

satisfiability, 15
search techniques, 16
semantic constraint, 43, 61
  checking, 84
shape constraint, 78
shape graph, 81
Simplex technique, 22
simulated annealing, 17
simultaneous view maintenance, 3, 49, 92
SINFONIA, 42
Sketchpad, 20
SkyBlue, 20, 95
SkyBlue constraint types, 95
Smalltalk, 20
solution synthesis, 17
SPIFF
  architecture, 55
  GUI, 55, 131
Stay, 96
stochastic search techniques, 17
subgeom, 105
subgraph, 54
symbolic solutions, 28

T

TCL/TK, 55
Thinglab, 13, 20
Times, 100
tolerance CSP, 21
topologic constraint, 61
TopologicElement, 122

U

underconstrained, 11, 121
update map, 82
UserDefinedParameter, 100

V

value inference, 13
ValueVariable, 100
variational modeling, 35
view, 3, 49
view conflict, 70
view consistency, 50
view coupling, 49
View Manager, 55
view priority, 50, 70
view specification, 53
Volume, 122

Summary

Geometric modeling tools are now commonly used in various phases of product development, for example to generate product images or NC-code. Many applications, however, require functional information that is not contained in geometric models.

A feature in a product model combines geometric information with functional information, such as information about its function for the user in a design application, or its manufacturing process in a manufacturing application. An important issue in feature modeling is feature validation, which concerns the meaning of a feature, given by its information content. A feature modeling system should ensure that product modifications by a user are in accordance with the meaning of the features.

In an ideal product development environment, multiple views of a product coexist, each one representing the product in terms of features corresponding to a specific application or engineering discipline. In concurrent product development, modifications made to the product in any view should be reflected in the other views, while the feature models in all views remain valid.

A constraint-based feature validation scheme has been developed that provides the means to specify and maintain the validity of features. Constraints are used to specify such feature validity conditions; constraint satisfaction techniques are applied to maintain feature validity under product modifications from multiple views.

For feature validity specification, various constraint types of different domains have been identified: geometric, algebraic and topologic. Constraints of these domains are combined in feature class specifications.

A scheme for persistent feature validity maintenance has been developed, which is based on constraint management, integrated with management of a cellular geometry representation. The constraint management is separated into two levels: constraint storage and constraint satisfaction. Firstly, the constraint management module maintains in a constraint graph the set of constraints that result from modeling actions. Secondly, it ensures satisfaction of all constraints by deploying several constraint solvers that are each specialized for a particular domain.

The constraint management module maintains a coupling between the two levels, and it propagates solving results between the constraint solvers. The advantages of the two-level separation are that the interdependence of the constraints is handled by the constraint management module, while the constraints are satisfied efficiently by the solvers.

A geometric constraint solver has been developed based on extended 3D degrees of freedom analysis. Newly developed extensions include degrees of freedom ignorance for locus intersections, and restructuring of constraint loops. With these extensions, powerful 3D geometric constraint solving is achieved. For the analysis of over- and underconstrained geometric models, the concept of dependency graph has been developed, to identify the set of involved constraints.

Algebraic constraint solving is done with the SkyBlue solver; its constraint priority scheme is used to propagate feature parameter changes in such a way that the model is minimally disturbed.

Topologic constraints are checked by querying the cellular geometry representation; it allows detection of violations of topologic feature properties caused by feature interactions.

Product modifications are propagated between the views by link constraints that provide a generic coupling of the feature models of the views. A view priority scheme is used to handle conflicts between constraints of different views.

The multiple-view feature validation scheme has been implemented in the prototype modeling system SPIFF. Feature and constraint classes are specified declaratively in an interpreted object-oriented programming language. SPIFF has a generic graphical user interface and provides new facilities for feature visualization.

Constraint-based feature validation enables flexible and expressive feature validity specifications, combined with powerful feature validity maintenance. The validation scheme ensures that only meaningful products can be specified, and that their validity is maintained under product modifications from different views.

Samenvatting

Hulpmiddelen voor geometrisch modelleren worden tegenwoordig op grote schaal gebruikt in verschillende ontwikkelingsfasen van een produkt, bijvoorbeeld om afbeeldingen te maken of freesbanen te bepalen. In veel van dergelijke toepassingen is echter functionele informatie nodig die niet is opgenomen in geometrische modellen.

Een feature in een produktmodel koppelt geometrische aan functionele informatie, zoals informatie over de functie ervan voor de gebruiker in een ontwerptoepassing, of over het fabricageproces in een fabricagetoepassing. Een belangrijk onderwerp in feature modelleren is feature validatie; dit betreft de betekenis van een feature, zoals gegeven door zijn informatie-inhoud. Een feature modelleringssysteem moet ervoor zorgdragen dat veranderingen aan het produkt door een gebruiker, overeenstemmen met de betekenis van de features.

In een ideale produktontwikkelomgeving bestaan meerdere visies, of 'views', op een produkt; ieder ervan representeert het produkt in termen van features die horen bij een specifieke toepassing of ingenieursdiscipline. Bij simultane produktontwikkeling, of 'concurrent engineering', dienen produktwijzigingen die in een van de views gemaakt worden, in de andere views door te werken, terwijl de feature modellen in alle views geldig moeten blijven.

Er is een schema voor feature validatie ontwikkeld, gebaseerd op constraints; het verschaft het middel om de geldigheid van features te specificeren en te handhaven. Constraints worden gebruikt voor het specificeren van zulke geldigheidscondities voor features; oplostechnieken voor constraints worden toegepast om de geldigheid van features te handhaven bij produktwijzigingen vanuit meerdere views.

Voor het specificeren van geldigheidscondities voor features, zijn verscheidene typen van constraints uit verschillende domeinen geïdentificeerd: geometrische, algebraïsche en topologische. Constraints van deze domeinen worden samen gebruikt in generieke feature specificaties.

Voor de blijvende handhaving van de geldigheid van features is een schema ontwikkeld, dat gebaseerd is op constraint beheer, dat geïntegreerd is met het beheer van een cellulaire geometrie-representatie. Het constraint beheer is gescheiden in twee nivo's: het opslaan van constraints, en het oplossen ervan. Ten eerste slaat de constraint beheermodule in een constraint graaf de verzameling

van constraints op die het resultaat zijn van modelleeracties. Ten tweede zorgt deze ervoor dat aan alle constraints is voldaan, door verscheidene constraint oplosmodules in te zetten, die elk gespecialiseerd zijn voor een bepaald domein.

De constraint beheermodule onderhoudt de koppeling tussen de twee nivo's, en hij propageert oplosresultaten tussen de constraint oplosmodules. Het voordeel van de scheiding in twee nivo's is, dat de constraint beheermodule zorgt voor de onderlinge afhankelijkheid tussen de constraints, terwijl de constraints op efficiënte wijze worden opgelost door de oplosmodules.

Een oplosmodule voor geometrische constraints is ontwikkeld, gebaseerd op uitgebreide 3D analyse van vrijheidsgraden. Nieuw ontwikkelde uitbreidingen omvatten het negeren van vrijheidsgraden bij het doorsnijden van loci, en het herstructureren van constraint lussen. Deze uitbreidingen geven een krachtige oplosmethode voor 3D geometrische constraints. Voor de analyse van over- en ondergespecificeerde geometrische modellen, is het concept van de afhankelijkheidsgraaf ontwikkeld, om de verzameling van betrokken constraints te bepalen.

Het oplossen van algebraïsche constraints wordt gedaan met de SkyBlue oplosmodule; het constraint prioriteitenschema wordt gebruikt om wijzigingen van feature parameters te propageren, zodanig dat het model zo weinig mogelijk wordt verstoord.

De geldigheid van topologische constraints wordt gecontroleerd door de cellulaire geometrie-representatie te raadplegen; het maakt de detectie mogelijk van schendingen van topologische feature eigenschappen, veroorzaakt door interacties van features.

Produktwijzigingen worden tussen de views gepropageerd via koppelconstraints, die op generieke wijze de feature modellen van de views koppelen. Een schema met view prioriteiten wordt gebruikt om conflicten tussen constraints van verschillende views af te handelen.

Het feature validatieschema voor meerdere views is geïmplementeerd in het experimentele modelleringssysteem SPIFF. Generieke beschrijvingen van features en constraints worden gemaakt op declaratieve wijze, in een geïnterpreteerde object-georiënteerde programmertaal. SPIFF heeft een generieke grafische gebruikersinterface, en biedt nieuwe mogelijkheden voor het visualiseren van features.

Constraint-gebaseerde feature validatie maakt flexibele en expressieve specificaties voor de geldigheid van features mogelijk, gecombineerd met een krachtige handhaving ervan. Het validatieschema zorgt ervoor dat alleen betekenisvolle produkten kunnen worden gespecificeerd, en dat hun geldigheid gehandhaafd blijft onder produktwijzigingen vanuit verschillende views.

Curriculum Vitae

Maurice 'Moos' Dohmen was born on August 25, 1967 in Geleen, The Netherlands. His nickname was thought up by Marjo de Kraker, with whom he lives together.

He obtained his Gymnasium-β certificate in 1985 from Scholengemeenschap St. Michiel in Geleen. In 1991 he received a master's degree in Computer Science from the Department of Mathematics and Computing Science of Eindhoven University of Technology. His master's thesis describes a technique for management of geometric constraints in a geographical information system; the research was partly carried out at the Geodetics Department of the Municipality of Nijmegen.

Following, he stayed with the Department of Mathematics and Computing Science as a researcher, working on the specification and implementation of an interpreter for the LOOKS programming language.

In 1992, he joined the Faculty of Industrial Design Engineering of Delft University of Technology as a PhD student; research subject was the application of automatic gesture recognition for modification of geometric models.

In 1993, he switched to the Faculty of Technical Mathematics and Informatics of the same university; as a member of the Computer Graphics and CAD/CAM group, he carried out the PhD research on constraint-based feature validation which resulted in this thesis.

In October 1997, he joined Applicare Medical Imaging B.V. in Zeist, where he can be reached at maurice@applicare.nl.

