Semantics of Families of Objects

Proefschrift

ter verkrijging van de graad van doctor
aan de Technische Universiteit Delft,
op gezag van de Rector Magnificus Prof. dr. ir. J.T. Fokkema,
voorzitter van het College voor Promoties,
in het openbaar te verdedigen op 26 november 2008 om 10.00 uur

door

Hilderick Anne VAN DER MEIDEN,
informatica ingenieur,
geboren te Zevenaar.

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

co-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, co-promotor
Prof. dr. C. Witteveen, Technische Universiteit Delft
Dr. J.S.M. Vergeest, Technische Universiteit Delft
Prof. dr. M.H. Overmars, Universiteit Utrecht
Prof. dr. R. Joan-Arinyo, Universitat Politecnica de Catalunya
Prof. dr. C.M. Hoffmann, Purdue University

This research was sponsored by the Netherlands Organisation for Scientific Research (NWO)

Preface

This thesis is the result of my research at the Computer Graphics and CAD/CAM group of the Faculty of Electrical Engineering, Mathematics and Computer Science of Delft University of Technology.

The title, Semantics of Families of Objects, suggests a very broad, almost philosophical subject. However, this research was motivated by practical problems in Computer-Aided Design (CAD); the objects in the title are objects modelled by CAD systems, e.g. parts of industrial products. The thesis discusses the problem of modelling families of such objects, and focusses in particular on the semantics of such families, i.e. properties of such families that are important to designers and users.

Even though the motivation is a practical one, several fundamental problems needed to be solved. The main contributions of this thesis are a new declarative model for families of objects, a new method for solving geometric constraints, a first-ever method for solving topological constraints, and methods for computing parameter ranges in such models. Most of the results presented in this thesis have already been published in various journals and conference proceedings. These papers can be found in the bibliography [van der Meiden and Bronsvoort, 2005a, 2005b, 2006a, 2006b, 2007a, 2007b and 2008].

I have been working in the Computer Graphics and CAD/CAM group for more than six years, as a Master's student, as a PhD student and currently as a Postdoc, which is perhaps a long time by today's standards, but it has been such a pleasure to work here, it is simply difficult to leave.

I would like to thank, first of all, Dr. Wim Bronsvoort, for calling me up a couple of months after my graduation, just before I was about to accept a boring job, offering me the opportunity to do a PhD, which I readily took, and for which I am very grateful still. He has been a wonderful supervisor, open to new ideas, patient, understanding, and I have learned a lot from him about academic life.

Next, I thank Prof. Erik Jansen, for supporting my work, and, in particular, for giving me the opportunity to finish my thesis when it was already supposed to be done a year ago.

I thank the members of the defense committee for their constructive

comments on the draft thesis, which have resulted in significant improvements.

I thank my current and former colleagues who have worked on feature modelling, in particular Dr. Rafael Bidarra for creating the semantic feature modelling approach, on which my work is based. I thank all my other colleagues, for interesting discussions and for creating a great working environment.

I thank my family, Tieme, Aly, Niels en Dolf, who have always been convinced that I will be a professor one day. Well, not yet, but who knows?

I thank Nicole, my love, for taking my mind of work occasionally to have some fun and a social life, and for generally keeping things running smoothly around me whenever I'm lost in thought.

Finally, I kindly thank all friends, relatives and strangers who kept asking me what my thesis would be about, over and over, so I had to explain it, again and again, and by doing so, I actually found interesting new perspectives on my work, and that motivated me to continue.

Rick van der Meiden
Oktober 13th, 2008
Delft

Contents

1 Introduction **1**
- 1.1 Families of objects . . . 2
- 1.2 Research questions . . . 4
- 1.3 Outline of the thesis . . . 5

2 Literature review **7**
- 2.1 History-based models . . . 7
- 2.2 Dual-representation models . . . 12
- 2.3 Procedural, rule-based and declarative models . . . 15
- 2.4 The Semantic Feature Model . . . 18
- 2.5 Creating and using families of objects . . . 22

3 The Declarative Family of Objects Model **27**
- 3.1 Overview of the model . . . 27
- 3.2 Geometry and topology . . . 29
- 3.3 Representation of features and families . . . 32
- 3.4 Realisations . . . 36
- 3.5 Family membership and subfamilies . . . 39
- 3.6 Implementation . . . 40

4 Geometric constraint solving **43**
- 4.1 Introduction . . . 44
- 4.2 Clusters . . . 47
- 4.3 Solving approach . . . 51
- 4.4 Incremental algorithm . . . 56
- 4.5 Solution selection . . . 60
- 4.6 Constraints on 3D primitives . . . 65

5 Topological constraint solving **71**
- 5.1 Mapping topological constraints . . . 71
- 5.2 Boolean constraint solving . . . 77

6 Computing geometric parameter ranges **85**
6.1 Basic approach 85
6.2 Degenerate subproblems 89
6.3 Parameter range computation algorithm 92
6.4 Example constraint problem 96

7 Tracking topological changes **99**
7.1 Relating parameters and topology 99
7.2 Computing critical values 102
7.3 Degenerate entities 107
7.4 Parameter range computation 110

8 Conclusions and future research **113**
8.1 Feasibility and advantages of the approach 113
8.2 Implementation and possible applications 115
8.3 Limitations and future research 116

Appendices:

A Rewrite rules for clusters **119**
A.1 2D rewrite rules 120
A.2 2D/3D rewrite rules 121
A.3 3D rewrite rules 122

B Selection criteria for the intended solution **125**
B.1 Definition of the intended solution 125
B.2 Subproblem analysis 127
B.3 Construction analysis 133

Bibiography **139**

Summary **147**

Samenvatting (Summary in Dutch) **151**

Curriculum Vitae **154**

Chapter 1

Introduction

Computer-Aided Design (CAD) is now used in all engineering and design disciplines, e.g. mechanical engineering, electrical engineering, aerospace engineering, industrial design and architectural design. Within these different disciplines, CAD systems with different capabilities are being used. In this thesis we focus on solid modelling systems, which are used to create solid, or volumetric, models, i.e. models of objects that have volume. These systems are mostly used for part design in mechanical engineering and industrial design.

The current generation of solid modelling systems are feature modelling systems. In such systems, the user builds a model from features. Features are aspects of the shape of the model that can be individually identified and manipulated. In most systems, features are parametrised shapes that add a volume to or remove a volume from the model, e.g. protrusions, holes, cuts and blends. With such features, a design can be modelled in fewer steps than with low-level geometric operations, and by changing parameter values, complex modifications can be made easily. Also, features can contain functional information for use in different phases of the design process, e.g. requirements for the conceptual design phase, shape constraints for the detailed design phase, material properties for the analysis phase, and tolerances for the manufacturing planning phase.

The main reason for using CAD is that it allows designs to be easily modified and analysed without building physical prototypes, resulting in increased productivity and in cost reduction. Another reason that is sometimes cited, is the potential for reuse of CAD models, because they can be used to model families of objects. However, in current CAD systems, as we shall see, support for families of objects is very limited.

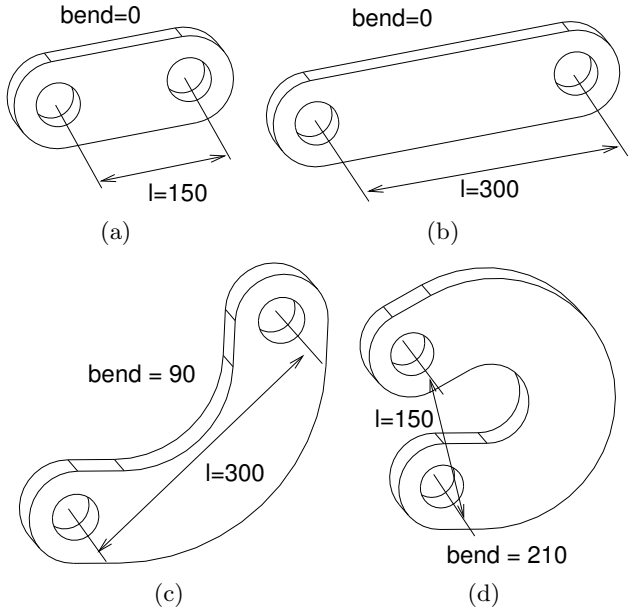


Figure 1.1: Variants of a CAD model with two parameters, l and *bend*.

1.1 Families of objects

Usually a single CAD model is interpreted as representing a single object. However, a feature model, and in general, any parametric model, can also be thought of as a representation of a family of objects. By varying the parameters of the features in the model, variants of the model are obtained, each representing a different but similar object (see Figure 1.1). The (infinite) set of all possible objects that can thus be obtained, is the family of objects represented by the model. The objects in this set are the members of the family.

A CAD model of a family of objects is useful in several situations. Firstly, the model can be used for manufacturing series of similar products, e.g. tools of different sizes, and even customised products. Secondly, the model can be reused as a part of a larger CAD model, with appropriate parameter values to fit it. Thus, modelling families of objects can yield increased design productivity and considerable cost reduction.

However, there is no widely accepted formal definition for families of objects in CAD [Shapiro and Vossler, 1995; Hoffmann and Joan-Arinyo, 2002]. In general, we can say that a family of objects is a set of similar objects. But similar in what respect? Depending on the exact definition, the objects in Figure 1.2 may or may not belong to the same family as those in Figure 1.1. On the one hand, it can be argued that these objects are members of the same family because they are constructed in the same way, i.e. from the

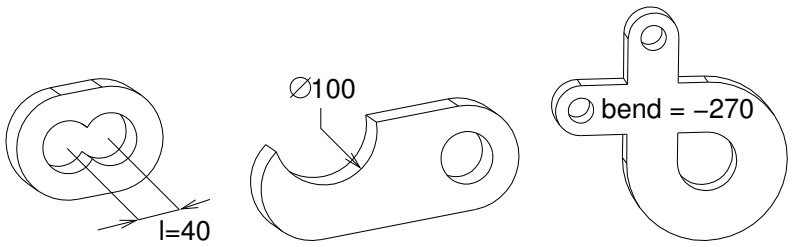


Figure 1.2: Objects that may or may not belong to the family in Figure 1.1.

same geometric primitives. Such procedural definitions are mostly used in practice [Hoffmann and Joan-Arinyo, 2002], but, as we shall see, are often not satisfactory. On the other hand, it can also be argued that these objects do not belong to the same family, because their topology is different from the topology of the objects in Figure 1.1, i.e. the objects cannot be mapped onto each other by a continuous transformation. Such mathematical definitions are not widely accepted, because the practical implications are not well understood [Shapiro and Vossler, 1995]. More importantly, these two definitions do not take into account that for different families of objects, different properties may be important, and it is thus desirable that when modelling families of objects, the user is able to precisely specify which objects are family members, and which are not, i.e. the semantics of a family.

For mechanical and industrial design, shape and function of objects are the most important aspects of family semantics. For example, the semantics of the family of objects in Figure 1.1 may be that members must be able to function as a connecting rod between two round shafts, and thus each member must contain two round holes. These aspects are related to geometric and topological properties of features and feature models. For modelling families of objects, it should thus be possible to specify geometric and topological properties, of features and families, in a generic way.

Current CAD systems, although they can be used to model families of objects, have not been designed for that purpose from the start. This can be seen, for example, from the fact that the model must at all times during the modelling process represent a single object; parameters must always have a value and no degrees of freedom are allowed in sketches. It is therefore not surprising that current CAD systems are often inadequate for modelling families of objects. There are two major problems.

Firstly, in current CAD systems, feature semantics cannot be precisely specified and is not adequately maintained [Bidarra and Bronsvoort, 2000b]. Features are supposed to represent design intent, but the actual semantics of features in current systems is not always consistent with the user's in-

tention. Thus, when parameter values of a model are changed, unexpected interactions between features may result in undesirable models, i.e. models with undesirable geometric or topological properties. This means that some families are difficult to create or impossible to represent in current systems.

Specifying precise semantics for features or families of objects, and adequately maintaining these semantics for all feature instances or family members, is not possible with current CAD systems, because these systems work with history-based models and boundary representations. History-based models are essentially procedural models, which are not suitable for specifying invariant properties of a family, i.e. properties that hold for all members of the family. Also, boundary representations do not contain sufficient topological information to verify feature semantics.

Secondly, current CAD systems do not provide adequate tools for modelling and exploring families of objects. A family consisting of a potentially infinite number of objects, is an abstract concept that is difficult to visualise and interact with in a natural way. In particular, when instantiating members of a family, choosing parameter values that correspond to members can be difficult. Also, it is difficult for a user to predict how parameter values affect the geometric and topological properties of the corresponding members of the family.

Thus, new representations and tools are needed for modelling families of objects.

1.2 Research questions

This thesis addresses the question of how families of objects can be modelled properly. There are two different aspects to this, corresponding to the two problems mentioned above. Firstly, how to model families of objects in general, i.e. how to represent such families, such that semantics can be specified and maintained. Secondly, how to model a specific family of objects, i.e. what kind of tools are needed to create and use a model of a family of objects. This leads to the following research questions:

Question 1 *How to model families of objects, such that semantics of features and families in general can be specified and maintained?*

More specific questions are:

- How to represent families of objects?
- How to specify and maintain semantics for all members of a family?
- How to instantiate members of the family?
- How to classify objects as members or non-members of a family?

Question 2 *What kind of tools are needed for creating and using families of objects?*

More specifically:

- How to interact, through a modelling system, with models of families of objects?
- How to explore the set of objects in a family model?
- How to analyse the relations between parameter values and the geometric and topological properties of the corresponding family members?

1.3 Outline of the thesis

We propose the following answer to Question 1:

> Families of objects should be modelled declaratively, using features with geometric and topological constraints.

To defend this proposition, firstly, in Chapter 2 we will argue that current CAD models cannot properly represent families of objects, because these models are not suitable for specifying the semantics of features and families in general. Various theoretical frameworks and alternative models are discussed, and we conclude that (1) a declarative model is best suited for modelling families of objects, and (2) semantics should be represented with geometric and topological constraints.

Next, in Chapter 3 we will present a model called the Declarative Family of Objects Model (DFOM), which can represent families of objects using features with geometric and topological constraints. Members of a family are instantiated by solving the system of geometric and topological constraints. Family membership is defined in terms of DFOMs, such that we can classify objects as members or non-members of a given family, by comparing DFOMs.

To instantiate family members and to classify objects with respect to a given family, we need constraint solving algorithms for geometric and topological constraints. A new method for geometric constraint solving is presented in Chapter 4, and a method for topological constraint solving, the first ever published, is presented in Chapter 5.

To answer Question 2, we propose the following:

> For creating and using models of families of objects, tools are needed that compute parameter ranges and critical parameter values.

When creating models of families of objects, specifying geometry and topology is most important. Visualisation of the model and direct manipulation of its elements is needed. Direct visualisation of incompletely specified geometry found in models of families of objects is probably not feasible, nor desirable. Instead, members of the family should be visualised and interacted with, and operations on those objects should be mapped to operations on the family model.

For a given parameter, the parameter range is the set of all values for which the model satisfies its geometric and topological constraints. Such parameter ranges help users to instantiate members of a family, and can be used as a tool for exploring a family of objects. In Chapter 6 and Chapter 7, algorithms are presented for computing the range of any parameter of a declarative model.

The algorithms developed for this determine so-called critical values. These are the parameter values for which geometric subproblems degenerate or for which topological entities degenerate. If for any value in between two subsequent critical values all constraints in the model are satisfied, then for all values between these critical values the constraints are satisfied. Thus, from the critical values we can determine the intervals which constitute the parameter range.

The critical values are also useful for analysing the behaviour of the model. While varying a parameter, changes in the topology occur only at critical values. Thus, critical values can be used to find objects with certain topological variations that may or may not be desirable members of the family. We therefore also present these critical values to the user.

In Chapter 8, we present conclusions drawn from this research and ideas for future research.

Throughout this thesis, the advantages of a declarative approach to modelling families of objects will become clear. It allows us to properly specify the semantics of features and families of objects in general, and we can analyse such declaratively specified families of objects, e.g. by computing parameter ranges and critical parameter values. These results could not have been obtained with the traditional, procedural modelling approach.

Chapter 2

Literature review

Families of objects can be represented by various types of models, some of which are used in commercial CAD systems and some of which have been suggested by academics. In this chapter, we discuss the advantages and limitations of various types of models for representing families of objects.

First we discuss history-based models, which are created by most current commercial CAD systems, in Section 2.1. Next we discuss models for families of objects more in general, from different perspectives. The first perspective considers the different representations used in such models. In general, such models are dual-representation models, which are discussed in Section 2.2. The second perspective considers the way semantics is specified in models of families of objects. From this perspective, models can be classified as procedural, rule-based or declarative, as discussed in Section 2.3. One particular declarative model, the Semantic Feature Model, which is used as the basis for the work described in this thesis, is described in Section 2.4. Finally, the different ways in which users can interact with a system for modelling families of objects, i.e. the operations and queries available to create and use such models, are discussed in Section 2.5.

2.1 History-based models

Current commercial CAD systems, e.g. Pro/Engineer[1], CATIA[2], SolidWorks[3], NX[4], SolidEdge[5], and Inventor [6], are parametric feature modelling systems. Almost all such systems create models that essentially describe a history of modelling operations. These operations are typically operations on a boundary representation (B-rep), in particular operations that add features to a

[1]Pro/Engineer, Parametric Technology Corp., http://www.ptc.com
[2]CATIA, Dassualt Systèmes, http://www.3ds.com/products/catia
[3]SolidWorks, Dassualt Systèmes, http://www.3ds.com/products/solidworks
[4]NX, Siemens, http://www.plm.automation.siemens.com
[5]SolidEdge, Siemens, http://www.plm.automation.siemens.com
[6]Inventor, Autodesk, http://www.autodesk.com

B-rep. To remove features, the corresponding operations are removed from the history. We refer to these systems as history-based modelling systems, and to models produced by these systems as history-based models.

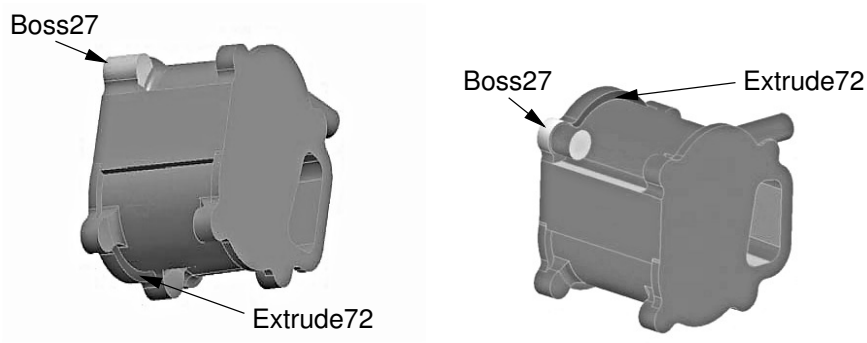
The procedure for modelling a family of objects with these systems, is as follows. Initially a single family member is modelled, which is referred to as the prototype or generic object. Other members of the family are instances of this prototype with different parameter values. To instantiate a model for a member of the family, the history of modelling operations, stored in the prototype, is re-evaluated with a new set of parameter values. Thus, each modelling operation is executed with the new parameter values, to create a new model, which is the requested member of the family.

History-based models, even though they are the de-facto standard for commercial modelling systems, are not really suitable for representing families of objects. In [Bidarra and Bronsvoort, 2000b], several major problems with the history-based approach are identified, of which the most relevant, in the context of families of objects, are the persistent naming problem, the feature ordering problem, and the inability to maintain feature semantics.

The persistent naming problem, essentially, is the problem of identifying corresponding entities in different members of a family. This identification is necessary because operations in the modelling history of a CAD model can contain references to geometric entities that were created by previous operations in the modelling history, in particular, references to B-rep entities. Such a reference can be used, for example, for positioning a feature. However, when evaluating the history for different parameter values, a new geometric representation is built, and references must now point to the corresponding entities in this new representation. Therefore, a so-called persistent naming scheme is needed that can identify corresponding entities in geometric representations created for different parameter values. One of the issues that a persistent naming scheme must deal with, is that entities may disappear or may be split when parameter values are changed.

Developing a persistent naming scheme is very difficult, and it is clear that current CAD systems use a flawed approach that can result in errors. For an example, see Figure 2.1. Here, a feature called Boss27 is edited, and as an unexpected side-effect, a feature named Extrude72 jumps to a different location, i.e. before the operation, Extrude72 was not adjacent to Boss27, but afterwards, it is. Obviously, such unpredictable behaviour is not desirable.

Several schemes have been brought forward to alleviate the persistent naming problem, so that history re-evaluation can at least be consistently executed [Kripac, 1995; Chen and Hoffmann, 1995; Capoyleas et al., 1996; Lequette, 1997; Wu et al., 2001]. See [Marcheix and Pierra, 2002] for a recent survey, containing several more references. All these schemes use auxiliary data structures to keep track of how faces and edges evolve, which,



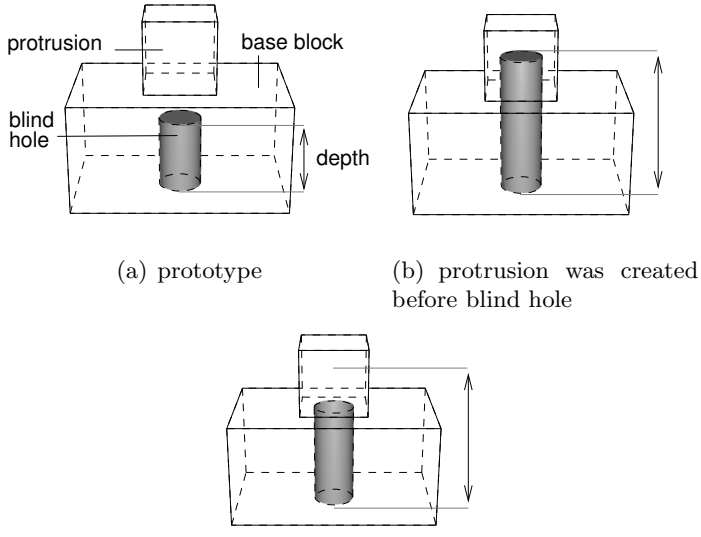
(a) before (b) after editing Boss27

Figure 2.1: Error in a commercial modelling system, related to persistent naming. Feature Extrude72 jumps to a different location when Boss27 is edited. Source: [Raghothama, 2006]

however, does not really solve the problem. The reason for this is that the schemes try to keep track of B-rep entities that are not persistent, and this is impossible in a truly generic way.

A different solution to the persistent naming problem is presented in [Bidarra et al., 2005b]. Model entities, i.e. entities in the geometric model, can here be selected and used for attaching and positioning features. However, the system converts the attach constraints and positioning constraints to constraints on feature entities, which are persistent. If necessary, the system automatically generates datum planes, on which additional constraints are imposed. Thus, no references to non-persistent model entities are stored in the model. However, in general, a unique description for model entities in terms of feature entities and datum planes cannot always be found, thus some feature placements may still have to be done manually, i.e. by selecting feature entities and creating additional datums.

In the context of families of objects, the persistent naming problem implies that a family may contain objects that are not desirable, and/or may not contain all objects that are desirable. Previous research on families of objects has focussed mainly on the persistent naming problem, by proposing new geometric representations, such that entities in different models, created from the same features, can be identified, e.g. the Generic Geometric Complex (GGC) [Rappoport, 1997]. The GGC represents families of Selective Geometric Complexes (SGCs). A SGC [Rossignac and O'Connor, 1988] represents an object by carriers, which are n-dimensional algebraic or parametric geometries, and by entities, which are disjoint (non-overlapping) point sets obtained from intersections of carriers. The GGC represents a family of SGCs with the same carriers, and a set of entities identified by a combination of carriers and additional selection criteria, to differentiate



(a) prototype

(b) protrusion was created before blind hole

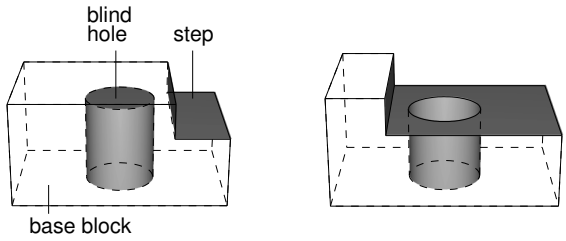
(c) protrusion was created after blind hole

Figure 2.2: Example of the feature ordering problem. When the depth of the blind hole feature in the prototype (a) is increased, either model (b) or model (c) may emerge, depending on the modelling history. Source: [Bidarra and Bronsvoort, 2000b]

multiple intersections. However, this approach does not always work, in particular when the feature geometry consists of curves and curved surfaces. Thus feature models with references to entities in the geometric representation can still be ambiguous.

The second problem with history-based models, the feature ordering problem, is caused by the fact that features add or remove material from the model in a fixed order. The order of modelling operations that seemed appropriate for a particular family member, namely the prototype object, may not yield the expected result when re-evaluating the history to create other family members. Consider, for example, Figure 2.2. The prototype object consists of a base block, a protrusion feature and a blind hole feature, as shown in Figure 2.2(a). When instantiating a variant object where the depth of the blind hole is increased beyond the height of the base block, two results are possible, depending on the order in which the features were created in the prototype. If the protrusion was created before the blind hole, the model shown in Figure 2.2(b) emerges. If, however, those features were created in the reverse order, the model shown in Figure 2.2(c) emerges.

The implication of the feature ordering problem for modelling families of objects, is that the order of operations must be taken into consideration,



(a) prototype (b) edit step feature

Figure 2.3: The semantics of a blind hole requires that the hole has a bottom face, as in the prototype (a). When the step feature is changed as in model (b), the blind hole is changed into a through hole. Source: [Bidarra and Bronsvoort, 2000b]

even though the effect may not be visible in the prototype. Although the order of features in the modelling history can usually be edited, this complicates design and editing of family models, in particular models with many interacting features.

The third problem with history-based models is maintaining feature semantics. Users of a CAD system expect features to have certain semantics, i.e. certain properties that are meaningful for the function or manufacturing of the product being modelled. In particular, topological properties are relevant for feature semantics. Due to interaction with other features, however, the topological properties of a feature may change. For example, Figure 2.3(a) shows a prototype object consisting of a base block, a blind hole feature and a step feature. The semantics of a blind hole requires that the hole has a bottom, i.e. that the hole does not cut entirely through the object. When the step feature is changed as in Figure 2.3(b), the blind hole feature does cut through the object, thus the semantics of the feature has changed, from the semantics of a blind hole to the semantics of a through hole.

By using a limited set of feature types and strict adherence to proven modelling practice, undesirable situations as described above can sometimes be avoided. However, this practice has in fact obscured the problems with history-based models, which will still occur, in unpredictable ways.

To maintain feature semantics, topological properties must be verified, and if necessary, action must be taken to restore feature semantics, i.e. the user should be informed. Current CAD systems, however, can only check the topological properties of a feature during instantiation of the feature into the model. If, due to interaction with other features, the topological properties of a feature change at later stages in the evaluation of the modelling history, this cannot be detected. The reason is that the result of feature operations

is stored in a B-rep, and the topology of the features cannot be stored in this representation. As a result, topological properties of the features cannot be adequately verified.

Specifying and maintaining feature semantics is essential for families of objects, because feature semantics should determine which objects are members of a family and which are not. In current modelling systems, feature semantics is in fact hard-coded in feature operations. For modelling families of objects, the available feature types and their semantics may not be sufficient. Feature semantics may be too restrictive or not strict enough for modelling certain families, i.e. certain objects are incorrectly included or excluded from the family. In current modelling systems, only the ordering of and the relations between features can be changed, but for modelling families of objects, it may be necessary to change the semantics of features too. Thus, it is desirable that the precise semantics, in particular topological properties, can be specified for features, and also for families as a whole. This gives more modelling freedom and results in more reliable models.

2.2 Dual-representation models

To understand the problems with modelling families of objects in a more general, theoretical context, the concept of dual-representation models has been introduced [Shapiro and Vossler, 1995]. Such models consist of a parametric representation, e.g. a CSG representation, and a geometric representation, e.g. a B-rep. History-based models are dual-representation models too, where the parametric representation is the modelling history, and the geometric representation is the B-rep.

Geometric representations for static, solid objects have been studied extensively in the past. For an overview of such representations, see [Rossignac, 2007]. Most of these representations, e.g. the B-rep, are cell-complex representations. Such representations describe both geometric and topological aspects of objects. By topology we refer to the connectivity of points sets in Euclidean space. The topology of an object is represented by relations between point sets of different dimensions, called entities. For example, in a B-rep, connectivity relations between faces, edges and vertices are stored. The geometry of the object is represented by attributes associated with the entities, e.g. the coefficients of curves and surfaces, and the coordinates of vertices.

However, for modelling a family of objects, geometry and topology must be represented in a generic way for all the members of the family. A parametric representation characterises a family with a set of parameters. For a given set of parameter values, the parametric representation can be evaluated, resulting in a geometric representation of a family member. In typical parametric representations, e.g. in the CSG representation and the history-

based representation discussed in Section 2.1, the parameters determine the geometry of the features, which are then combined using Boolean operations, thus determining the topology of the whole object.

This view has led to considering two types of families: the parameter-space family and the representation-space family. The parameter-space family is the set of all objects that can be obtained by varying the parameters. The representation-space family corresponds to the set of all objects that can be obtained by certain operations on the geometric representation. These two families are related by the procedures used to construct the geometric representation from the parametric representation, but this relation is not yet well understood. It is, however, generally agreed upon that the concept of a family is related to a certain notion of continuity in the geometric representation, and that for maintaining semantics of families of objects, it is necessary to establish a relation between parametric operations and continuous geometric transformations.

A specific representation-space family is described by the concept of boundary representation deformation [Raghothama and Shapiro, 1998]. Basically, a family of objects is here defined by a prototype B-rep, and contains all objects that can be created by a continuous deformation of the prototype. The authors acknowledge that this definition of a family is too restrictive for practical modelling of families of objects, because the boundary representation deformation cannot account for splitting and merging of topological entities.

A more general framework for families of objects has been proposed in [Raghothama and Shapiro, 2002]. Here, the concept of part families is described using category theory, a branch of mathematics that deals with broad classes of mathematical objects, such as the category of sets and the category of topological spaces. A part family is defined as a sub-category of the category of cell-complexes, such that there is a mapping between the cell-complexes in the part family, with certain continuity preserving properties. While this formulation of a family of objects does allow for some topological variations, e.g. splitting and merging of topological entities, it does not help to explicitly relate variations in the parameter space to variations in the topological space.

In [Raghothama, 2006], representations are classified as constructive or non-constructive. Parametric representations, e.g. CSG and history-based representations, are constructive representations, which allow models to be constructed incrementally using parameterised operations. However, these models do not explicitly represent topology, and are not suitable for enforcing continuity in Euclidean space. Non-constructive representations, such as cell-complexes, do explicitly represent topology, but are difficult to parameterise. For modelling families of objects, the author proposes a constructive topological representation (CTR), which can be used as parametric model

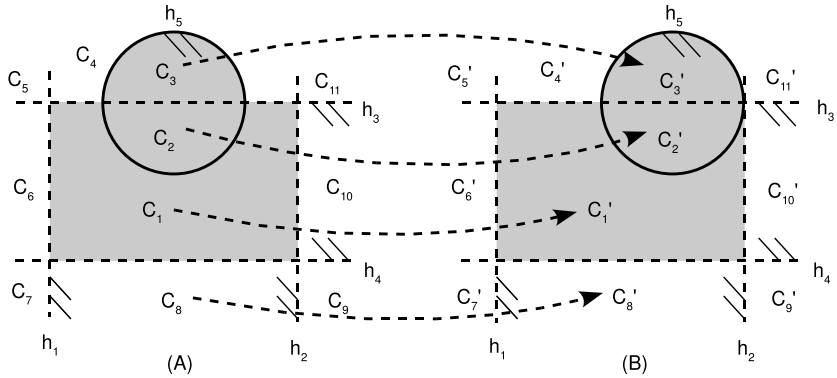


Figure 2.4: Continuous CTR map from CTR (A) to CTR (B). Carriers are h_1 through h_5 and h'_1 through h'_5, atoms are C_1 through C_{11} and C'_1 through C'_{11}. Source: [Raghothama, 2006]

and also to enforce spatial continuity.

Such a constructive topological representation is derived from a non-topological constructive representation, e.g. a CSG representation, which can be described by a boolean combination of carriers, e.g. geometric primitives. A CTR is uniquely described by a set of atoms, which represent all disjoint subsets of Euclidean space induced by the carriers in the model. It is topologised by defining a neighbourhood for each atom. The neighbourhood of an atom is defined using the *Star* operator, which is the set of all atoms in the CTR that are overlapping with it or adjacent to it, including itself. Consider model (A) in Figure 2.4. The model consists of carriers $h_1 \dots h_5$ and atoms $C_1 \dots C_{11}$. Here $Star(C_2) = (C_1, C_2, C_3, C_4)$.

If there is a so-called continuous CTR map between two CTRs, then the two objects are in the same representation space family. A mapping g from one CTR to another is a continuous CTR map if for each atom, the mapping applied to the Star of the atom is a subset of the Star of the mapping applied to the atom, i.e. $g(Star(C_i)) \subset Star(g(C_i))$. A continuous CTR map is illustrated in Figure 2.4. Every atom C_i in model (A) is mapped by a function g to a carrier C'_i in model (B). For example, for atom C_2, we find that $g(Star(C_2)) = g(C_1, C_2, C_3, C_4) = (C'_1, C'_2, C'_3, C'_4)$, and that $Star(g(C_2)) = Star(C'_2) = (C'_1, C'_2, C'_3, C'_4, C'_{10}, C'_{11})$. In this case, and for all other atoms, we find that the mapping of the Star of the atom is a subset of the Star of the mapping of the atom, and thus the mapping is a continuous CTR map, i.e. the models are in the same family.

With the CTR map, it is possible to determine whether models with different parameter values are in the same representation space family, i.e. have similar topology. Note that for a correct family definition, the existence of a CTR map must be an equivalence relation, which is not shown in

[Raghothama, 2006], but seems plausible. However, a CTR does not explicitly represent the relation between parameters and topology, e.g. it is not possible to explicitly determine the parameter values for which the topology changes.

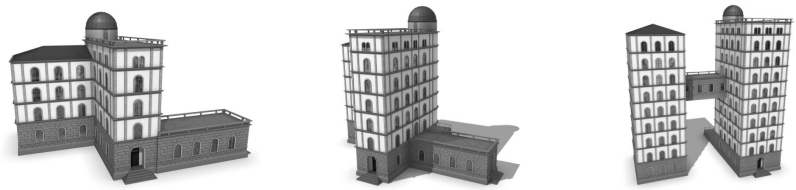
The models for families of objects discussed above are mostly concerned with preserving continuity in the geometric representation, and therefore, families are defined in terms of continuous transformations or mappings. However, this definition of a family of objects is rather limited; in practice, it may be desirable for a family of objects to have members with different topology. For example, a family of objects with two hole features, may have some members in which these two holes intersect, whereas in other members of that family, these two holes do not intersect. It should be possible to specify whether this is desirable, i.e. the semantics of the family (See Section 1.1), but in the models discussed above, semantics is fixed. Also, other shortcomings of the history-based approach with respect to modelling families of objects, in particular the feature ordering problem and the problem of maintaining feature semantics, are not addressed.

2.3 Procedural, rule-based and declarative models

The previous section discussed different representations used in models for families of objects. In this section, we consider the way in which semantics can be specified in such models. In particular, we distinguish three classes of models: procedural models, rule-based models, and declarative models.

For modelling families of objects, the ability to specify invariant properties, i.e. properties that must hold for all objects in the family, is essential. In particular, because we are mostly concerned with the shape of objects, we should be able to specify invariant geometric and topological properties, e.g. the diameter of a hole feature and that the hole must be a blind hole. We discuss for each of the three classes of models if and how they can be used to specify families of objects with such invariant properties.

Procedural models (or imperative models), specify how to construct objects in the form of a procedure that generates objects for given parameter values. The history-based model is essentially a procedural model; it specifies a history of modelling operations, which is basically a procedure for generating geometric representations from parameter values. Each modelling operation itself is a procedure that determines a new geometric representation from a previous geometric representation. The geometry and topology of the new representation depends on the order and the parameters of all the modelling operations in the history. Creating a history of modelling operations such that a geometric or topological property holds for all members of the corresponding family, is therefore very hard. In particular, topological properties cannot be properly specified and maintained in history-based



(a) (b) (c)

Figure 2.5: Architectural designs created using a shape grammar. Source: [Müller et al., 2006]

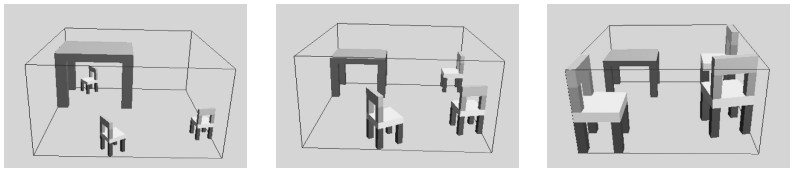
modelling systems, as we have seen in Section 2.1, and in general, creating procedural models with specific invariant properties is very difficult.

Families of objects may also be created using knowledge-based engineering systems and automated design synthesis systems. These systems determine possible design solutions for a given set of requirements, implicitly specified by a set of rules. The rules can be executed by the system in any order, to construct a set of objects, representing possible design solutions. We refer to the models in such systems as rule-based models.

An example of a rule-based model based on a shape grammar is presented in [Stiny and Gips, 1972]. A shape grammar specifies how shapes can be formed using elementary replacements in geometric representations. Typically, a computer system generates alternative shapes, starting from an initial shape, by applying one or more grammar rules. Some implementations support only 2D shapes, e.g. [Tapia, 1999], but progress has also been made with 3D shape grammars, e.g. [Chau et al., 2004]. Such shape grammars are mostly used in architectural design, e.g. [Müller et al., 2006] (see also Figure 2.5).

Rule-based models can also be used for topology optimisation. For example, in [Shea et al., 1997] a technique called shape annealing is used to find truss structures, e.g. dome structures, with optimal strength. A large number of models, each with a different topological structure, is generated by a shape grammar, and a performance measure is calculated for each model. A search algorithm similar to simulated annealing is used to find models with near-optimal performance.

With the rule-based approach, as was the case with the procedural approach, it is difficult to specify invariant properties, because rules are basically procedures, and it is difficult to combine different procedures such that invariant properties are always satisfied. Rule-based systems can be used to search for particular objects, by testing whether the objects generated by the rules satisfy certain invariant properties. However, in general, this is not an efficient way to find all the objects that satisfy the invariant properties, in particular when the invariant properties are continuous properties, e.g.



(a) (b) (c)

Figure 2.6: Variations upon the "A table and 3 chairs" theme, from Multiformes. Source [Ruchaud and Plemenos, 2002]

geometric properties. In general, these systems are therefore not used for modelling families of objects, but rather for support of the model creation process or topology optimisation.

Declarative models explicitly state invariant properties of objects, but not how to construct those objects. Such a model specifies variables, i.e. elements that exist in all objects in the family, but whose properties can vary, and constraints, which state invariant properties by imposing relations between variables. The model does not specify how to satisfy those constraints, but rather, a generic constraint solver is used to determine values for the variables such that all constraints are satisfied, i.e. the solutions or realisations of the model. In general, there can be many solutions to a system of constraints, and thus a declarative model can naturally describe a family of objects.

Constraints have been used in several declarative scene modelling systems, e.g. WordEye [Coyne and Sproat, 2001], DEM^{2}ONS [Kwaiter et al., 1997] and Multiformes [Ruchaud and Plemenos, 2002]. Typically, the model in such a system consist of several objects that are placed in 3D space to satisfy topographical constraints, specifying, for example, that object A must be to the left of object B. The modelling system determines various configurations of the objects in the scene, and presents these scenes to the user (see Figure 2.6). In this way, these systems support the creative process.

However, our focus is not on modelling scenes and supporting the creative process, but on modelling families of objects. To specify a family of objects in a completely declarative way, we must be able to specify all sorts of invariant geometric and topological properties. The topographical constraints used in declarative scene modelling systems cannot be used for this. Instead, geometric and topological constraints are needed.

Geometric constraints have long been used in CAD systems for specifying geometric relationships in sketches, e.g. Sutherland's 1963 Sketchpad system [Sutherland, 2003]. Current history-based feature modelling systems use essentially the same sketching approach for what are generally called sketched features. Geometric constraints are imposed on 2D geometric primitives such as lines and circles, to constrain the dimensions of

those objects (lengths, radii), their relative position and orientation (angles, distances, parallelism), and other relations (adjacency, tangency).

Topological constraints state invariant topological properties that must be satisfied by all members of a family. Useful topological constraints state requirements on the connectivity of specific point sets in a model that are meaningful to the user, i.e. features or faces of features. For example, a topological constraint may state that the bottom face of a blind hole feature must be on the boundary of the model, so that the hole is actually blind. In this way, topological constraints determine the possible topological variations of a model, independently of the geometric constraints. Thus, whereas geometric constraints are used to parameterise the shape of the model, topological constraints are used to limit the range of topological variations of the shape.

Topological constraints cannot, in general, be specified in current CAD systems, because these systems create history-based models and use a B-rep for representing the geometry. The topology of the B-rep is determined by evaluating the modelling history, independently of any topological constraints. And although some topological aspects may be implicitly checked by such systems, in general, topological constraints cannot be verified, because the B-rep does not contain all topological information needed for this.

One particular declarative modelling approach, the Semantic Feature Model, discussed in the next section, supports both geometric and topological constraints, which can be used to specify the semantics of features and of families of objects.

2.4 The Semantic Feature Model

The Semantic Feature Model (SFM) [Bidarra, 1999; Bidarra and Bronsvoort, 2000b], is a declarative model that allows feature semantics to be adequately specified and maintained. Also, the order in which features are added to the model does not determine the resulting object (at least, in most cases, see below).

A SFM consists of a set of features and additional constraints between features. The shape and position of all features is determined by solving the constraints specified in the features (feature constraints), and the additional constraints between features (model constraints).

Each feature is instantiated from a feature class. A feature class consists of shape elements, positioning elements, topological elements, and a user interface, as shown in Figure 2.7. The user interface of a feature class contains parameters that control the shape and position of the feature. In particular, *shape parameters* determine the shape of the feature, via *shape constraints*, which are imposed on *shape entities*. *Positioning parameters* are parameters of *positioning constraints*, which are imposed on shape entities

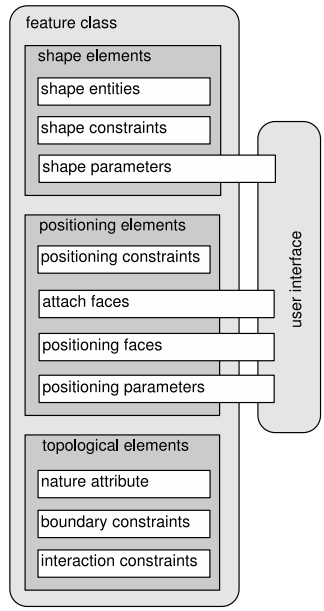


Figure 2.7: Elements of a feature class definition.

and on *attach faces* and *positioning faces*, which are faces of other features in the model or reference geometry, set by the user.

The topological elements of a feature class are its nature attribute, boundary constraints and interaction constraints.

The *nature* attribute can be *additive* or *subtractive*, indicating whether the feature adds material to the model or removes material from the model.

A *boundary constraint* is associated with a feature face, and specifies that the face must be (partially or completely) on the boundary of the model, or may not be (partially or completely) on the boundary of the model. A boundary constraint can be used to specify, for example, that the bottom face of a blind hole feature must be on the boundary of the model, so that the hole is always blind.

Interaction constraints are associated with a feature as a whole. Interactions with other features may create specific topological patterns, which can be disallowed by these constraints. An interaction constraint can be used to specify, for example, that a feature may not be split into disjoint parts by other features in the model. Table 2.1 lists interactions commonly found in feature models that can be constrained in the SFM.

The geometric representation of the SFM is the *cellular model* (CM), a cell-complex representation that can be used to store semantic feature information [Bidarra et al., 1998]. The cellular model consists of topological entities, i.e. vertices, edges, faces and cells, and all topological relations be-

tween these entities. Note that in literature on cell-complex representations, usually all topological entities are called cells, whereas here we use the word cell only for those entities representing volumes. All cells are quasi-disjoint, meaning that cells may touch (share a face, edge or vertex), but they cannot intersect. Each cell represents either a volume filled with material, i.e it is part of the modelled object, or it represents an empty volume, i.e it is not part of the object.

The CM is constructed by combining all the feature shapes in the model, and can be updated efficiently when the feature model is changed [Bidarra et al., 2005a]. If features intersect, they are split into non-intersecting new entities, which are then added to the CM. The CM thus contains the geometry of all the features, including the geometry that is not on the boundary of the model. In contrast, the B-rep of a history-based model loses feature geometry with each set operation. For each cell, the CM stores a list of features that overlap with the cell, referred to as the *owner list* of the cell.

For each cell, it is determined whether it contains material, by *dependency analysis*. The positioning faces and attach faces specified in the user interface determine dependency relations among features. If feature F_1 refers to one or more faces of a feature F_2, then F_1 is said to be *directly dependent* on F_2. These relations are represented by a *dependency graph*, which is a directed graph, where every direct dependency of a feature F_1 on a feature F_2 is represented by an edge (F_1, F_2). In general, a feature F_x is said to be dependent on a feature F_y if there is a path from F_x to F_y in the dependency graph. Feature precedence is a partial ordering derived from the feature dependency graph, as follows: if a feature F_x depends on a feature F_y, then F_x precedes F_y. For each cell in the CM, a precedence order is determined for the features in the owner list of the cell. The nature of the feature with the highest precedence determines whether the cell contains material. If the nature of that feature is additive, the cell contains mate-

Interaction type	Description
Splitting	Causes the boundary of a feature to be split into two or more disconnected subsets
Disconnection	Causes the volume of an additive feature (or part of it) to become disconnected from the model
Obstruction	Causes (partial) obstruction of the volume of a subtractive feature
Closure	Causes a subtractive feature volume to become (part of) a closed void inside the model
Absorption	Causes a feature to cease completely its contribution to the model boundary

Table 2.1: A list of interactions in feature models. Adapted from [Bidarra and Bronsvoort, 2000b].

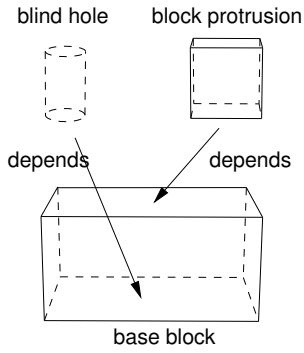


Figure 2.8: Feature dependencies for the model in Figure 2.2. The blind hole and the protrusion both depend on the base block, but are independent of each other.

rial. If the nature of that feature is subtractive, the cell does not contain material.

When for each cell in the CM it has been determined whether it contains material, the validity of all features is checked by verifying that all boundary constraints and interaction constraints are satisfied. If any constraint is not satisfied, the model is invalid, and the user is guided through a recovery process, until validity has been restored.

Problems occur when there are features in the owner list of a cell that are independent and have conflicting natures. For example, Figure 2.8 shows the feature dependencies of the model in Figure 2.2. The protrusion and blind hole features are both dependent on the base block, because they refer to it for positioning, but there are no dependencies between these two features. Thus no feature precedence can be determined for these two features, and again either Figure 2.2b or Figure 2.2c emerges, dependent on the order of feature creation, just like in history-based systems. Interestingly, the SFM approach can detect that the semantics of the blind hole feature in Figure 2.2c is incorrect, because the bottom of the blind hole in the cellular model does not correspond to the bottom of the blind hole in the feature definition, i.e. the boundary constraint on this feature face is not satisfied. However, this information is not used to determine the correct feature precedence order.

A model for families of objects, based on the SFM, is the Semantic Model Family [Bidarra and Bronsvoort, 2000a]. A family consists here of all models with the same features and constraints, but different parameter values. Boundary and interaction constraints guarantee that every member of the family has valid feature semantics.

The main shortcoming of the SFM as a basis for defining families of objects, is that feature dependency analysis cannot always unambiguously

decide which cells should contain material, in particular when there are features in the owner list of a cell that are independent and have conflicting natures, as discussed above. In general, feature dependency analysis does not respect the semantics of features as specified by topological constraints. Topological constraints are only checked after a model has been created, instead of being used to create a valid model. As a result, a family of objects defined by a SFM is not complete, i.e. sometimes no object is found that satisfies the topological constraints, even though such an object exists.

2.5 Creating and using families of objects

In [Rappoport, 1997], a model is defined as a representation with specific queries and operations. The available queries and operations determine how we interact with a system to create and use models of families of objects.

In current CAD systems, a family of objects is designed by first modelling a prototype object. The prototype is a fully determined object, and can thus be visualised and be edited by manipulating a graphical representation of the geometry and features in the model.

While modelling a prototype object, the designer is forced to make choices, to satisfy certain requirements. However, as requirements are often refined or changed during the design process, it may be necessary to undo previous choices, e.g. by changing parameter values or by removing features, to satisfy the new requirements. One problem here is that a feature often cannot be removed from the model without also removing the features that were added to the model later. Also, it is possible to miss alternative (perhaps better) solutions because of choices made early in the modelling process.

For declarative models, the design process can be thought of as a gradual narrowing down of a broad family of objects to a smaller family of objects, by adding or changing requirements, specified in the model using constraints. No arbitrary design choices have to be made to create a single object, until the last moment, e.g. just before analysis or manufacturing. Thus, no potential solutions are discarded during the design process. Also, in these systems, features and constraints can be removed without limitations due to a fixed modelling history, and thus design requirements can be changed and incorporated at any time.

For declarative scene modelling systems, interaction can generally be described by a three-step model [Gaildrat, 2007], consisting of a description step, a generation step and a lookup step. For the description step, most declarative systems use a modelling language to specify model entities and constraints, e.g. [Bonnefoi and Plemenos, 2000]. Alternatively, natural language is used, and natural statements are converted to constraints, e.g. [Coyne and Sproat, 2001]. In the generation step, all realisations of the

model are generated, if any exist at all. For models with an infinite number of realisations, this is not possible, so a representative set should be generated. In the lookup step, the realisations, generated in the previous step, are presented to the user. If the set of objects represented by the model is large, some means of exploring the set is necessary.

Two modes for interacting with a declarative modelling system are defined in [Bonnefoi et al., 2004]. In *exploration mode*, all realisations are generated, and the user can interactively explore the solution space. In *solution search mode*, only one realisation is generated. The solver will require the user to specify more details, interactively, during the solving process.

Such declarative modelling systems provide better tools for exploration of families of objects than current CAD systems. However, the input techniques used in these systems are not particularly suitable for designing part families for engineering and design. In particular, these systems provide no graphical interaction with the model.

Preferably, it should be possible to design families of objects through direct manipulation of a graphical representation. In models of families of objects, just like in models of single objects, shape aspects, in particular geometry and topology, are most important. Direct manipulation is much more intuitive for manipulating shapes than a language-based approach. Also, this is what designers using current CAD systems are used to.

However, in models of families of objects, geometry and topology may not be fully determined, and there is no obvious and meaningful way that these aspects can be visualised and interacted with. Thus, in practice, only members of the family can be visualised and interacted with, as is done in current modelling systems. Ideally, a system for modelling families of objects should allow a family model to be modified by interacting with any member model. Operations on a member model should be mapped by the modelling system to operations on the family model.

The most important operations in CAD systems are those for specifying and modifying shape aspects. In feature-based systems, fine control over shape is provided by feature parameters. Adding and removing features provides more global control over shape and semantics.

In industrial design, the type of features and the parameters necessary for a particular application are referred to as the modelling context. Modelling context may change over time, and feature conversions may be needed while a product is being modelled. Dynamic Shape Typing [Vergeest et al., 2002] is a framework for dynamically allocating a computational type, i.e. a geometric representation, to match the modelling context. In mechanical design, invariant properties of a model are more important, and features are more used to capture design intent; thus modelling context is usually considered static. For models of families of objects, we need to be able to control the shape and semantics for all members simultaneously. To

be able to do this, we assume a static modelling context, although feature conversions are possible in principle, in particular in multiple-view feature modelling [Bronsvoort and Noort, 2004].

The basic operations that should be supported by a model for families of objects are addition and removal of features and model constraints, and setting and unsetting of parameters. Note that setting and unsetting of parameters is essentially the same as adding and removing value constraints on variables that represent parameters. Free variables represent shape variations within a family, and variables with a user-specified value represent parameters that have the same value for all members of the family.

A system for modelling families of objects should also provide tools that allow the user to inspect the set of objects in a family. Tools from declarative scene modelling systems for exploring families of objects are useful for this, e.g. the possibility to visualise several members at the same time. However, to explore a family in this way, members of the family have to be instantiated. This can be problematic, because it is not always clear which parameter values will result in valid family members. This is, in particular, the case for complex models, and models that have been created by a third party.

A useful tool that can help when instantiating members of a family, is one that computes the range of allowable values for a parameter. Computing a range for several parameters simultaneously might also be useful, but only for perhaps two or three parameters, because a higher dimensional parameter range is difficult to present to the user [Hoffmann and Kim, 2001]. Methods for computing parameter ranges are discussed in Chapter 6 and Chapter 7.

Also, because the set of family members is often infinite, not all members can be inspected, and it will be difficult to get an overview of the modelled family. In particular, members with undesirable topological properties may exist that are hard to find by manually instantiating members. Therefore, it can be useful to know the parameter values for which topological changes occur in the model, i.e. the critical parameter values. By identifying topological changes, and the corresponding critical values, the designer can explore the topological variations of a family model. Computing critical values is discussed in Chapter 7.

To summarise, for modelling families of objects, we need a model in which the semantics of features and families can be specified and maintained, independent of the modelling history. In particular, it must be possible to specify invariant geometric and topological properties. Declarative models with geometric and topological constraints are most suitable for this. To unambiguously instantiate family members from such a model, we need to be able to solve systems of geometric and topological constraints. To ex-

plore the set of objects in the family, it should be possible to determine parameter ranges for the family model, and critical parameter values to inspect the topological variations between those objects. A new declarative model for families of objects is presented in the next chapter, and methods for solving geometric and topological constraints, and for determining parameter ranges and critical values, are presented in subsequent chapters.

Chapter 3

The Declarative Family of Objects Model

In the previous chapter, we have argued that models created by current commercial CAD systems do not adequately represent families of objects. Various other types of models have been discussed in that chapter too. Of these, declarative models seem the most appropriate for modelling families of objects. We have also seen that such models should include geometric and topological constraints to specify the semantics of features.

In this chapter, we present a new declarative model for families of objects, called the Declarative Family of Objects Model (DFOM). It is a generalisation of the Semantic Feature Model (SFM), presented in Section 2.4. Like the SFM, the DFOM is a declarative model with features and geometric and topological constraints. However, unlike the SFM, the geometry and topology of a DFOM does not have to be fully specified. Thus, a DFOM may have any number of realisations, and represents a family of objects. Also, topological constraints are solved to determine realisations, and thus the ambiguity of the feature dependency analysis used in the SFM is avoided. This allows us to properly specify families of objects.

In Section 3.1, an overview of the DFOM and the rest of the chapter is given.

Parts of this chapter have already been published in [van der Meiden and Bronsvoort, 2006b; van der Meiden and Bronsvoort, 2007a].

3.1 Overview of the model

The notion of a family of objects is often defined differently for various applications. Thus, a model for families of objects must be as general as possible and allow families with specific semantics to be defined. In mechanical and industrial design applications, shape and function are the most important aspects of objects to be modelled. These aspects are related to geometric

and topological properties of models.

The basic elements of any declarative model, and thus of the DFOM, are variables and constraints. All aspects of a family that can vary between members are represented by variables, e.g. the geometric and topological aspects of the members of a family are represented by geometric and topological variables. Constraints are imposed on one or more variables, of the same type or of different types, to specify invariant properties of features and the family as a whole, e.g. geometric and topological properties are specified by geometric and topological constraints.

More in detail, a family of objects is represented by a DFOM with geometric variables, called *carriers*, and topological variables, called *constructs*. Carriers define surfaces that partition space, e.g. a planar carrier defines a planar surface and two sides of the surface. Constructs basically represents point sets, i.e. volumes, surfaces, curves and individual points, constructed by intersections of subspaces defined by carriers. Carriers and constructs are related via so-called *subspace* constraints. Geometric properties of a family can be specified by geometric constraints on carriers, and topological properties by topological constraints on constructs. This representation allows us to declaratively specify almost any family of objects. The declarative representation of geometry and topology is further discussed in Section 3.2.

Variables and constraints that occur together frequently can be combined into features, which thus provide semantics at a higher level abstraction, closer to the function of a family of objects. We simply define a feature as a subset of the variables and constraints in a DFOM. Because we are interested in modelling volumetric objects, every feature must include a variable representing its volume. This definition is conceptually very simple, but allows for many different types of features to be specified. The representation of features and families is further discussed in Section 3.3.

Implicitly, a DFOM defines a set of realisations, i.e. all possible models of objects that satisfy the constraints in the DFOM, thus representing all possible family members. The geometric representation of realisations is the cellular model (CM), the same representation that is used in the SFM (see Section 2.4). This representation allows us to relate the declaratively specified geometry and topology of families, including parts of features that are not on the boundary of the model, to the geometry and topology of realisations.

To determine realisations, first the geometric constraints are solved, to determine the shape of the cells of a realisation's CM, and then the topological constraints are solved, to determine which cells contain material. Topological constraint solving replaces the feature dependency analysis of the SFM, and because it determines all possible realisations, the interpretation of the DFOM is unambiguous. This interpretation process is discussed in Section 3.4.

A DFOM can have zero, one, a finite or a (countable or uncountable) infinite number of realisations, represented by CMs. These realisations correspond to the members of the family. However, family membership can better be defined in terms of DFOMs. A member DFOM is defined as a DFOM with exactly one realisation, created by adding constraints to a DFOM representing a larger family, e.g. constraints to assign values to variables. Similarly, we can define a subfamily of a given family as a DFOM with a superset of the original DFOM's constraints. Thus, family membership is a well-defined relationship, which can be tested by comparing DFOMs. This is much easier than comparing geometric representations of realisations, which is difficult due to the persistent naming problem. Family membership is further discussed in Section 3.5.

Finally, the DFOM has been implemented in a prototype feature modelling system called SPIFF, developed at Delft University of Technology. This implementation is discussed in Section 3.6.

3.2 Geometry and topology

To be able to represent the variant shape of a family of objects, and specify its invariant geometric and topological properties, we use geometric variables, called *carriers*, and topological variables, called *constructs*, related by *subspace* constraints. This representation is elaborated here.

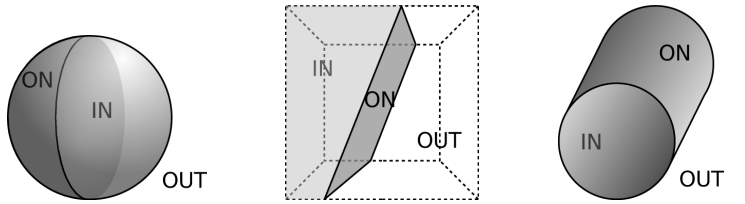
Carriers are used in various representations for families of objects, e.g. in [Rappoport, 1997; Raghothama, 2006]. Although carriers can be defined in any dimension, here we consider a carrier to be a function that partitions 3D Euclidean space into three subspaces, labelled IN, OUT and ON. The subspaces that correspond to IN and OUT are each connected, 3D point sets. The subspace labelled ON is a surface, separating the IN and OUT subspaces. The three subspaces must partition space, i.e. every point in space is either IN, ON or OUT. Thus, a carrier can be described by a three-valued function $C : \mathbb{R}^3 \rightarrow \{IN, ON, OUT\}$.

Algebraic surfaces, described by an equation $g(p) = 0$, can be used to define carriers, by defining $C(p)$ as follows:

$$C(p) = IN \iff g(p) < 0$$
$$C(p) = ON \iff g(p) = 0$$
$$C(p) = OUT \iff g(p) > 0$$

Examples of carriers based on simple algebraic surfaces are shown in Figure 3.1.

A carrier in a DFOM is a geometric variable, i.e. the geometry of a carrier by itself is not determined. For carriers based on algebraic surfaces, this means that the coefficients of the algebraic function that defines the surface, may also be variables. These variables can be assigned a value



(a) spherical carrier (b) planar carrier (c) cylindrical carrier

Figure 3.1: Carriers corresponding to simple algebraic surfaces

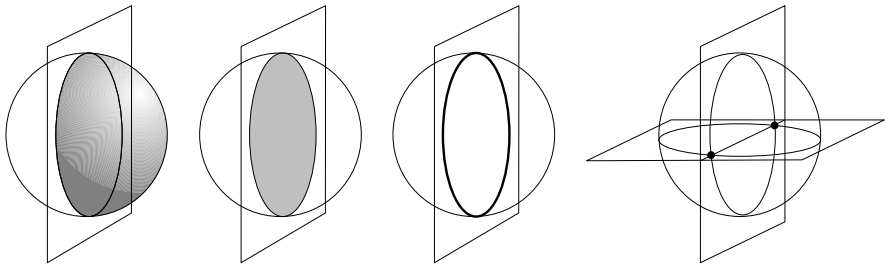
by a value constraint, or related by algebraic constraints. Primitives such as planes and spheres, used to define carriers, can be related by geometric constraints. For example, a planar carrier P can be constrained tangent with a spherical carrier S, by a constraint $tangent(P, S)$.

Note that in many modelling applications, parametric surfaces such as NURBS are used, which cannot easily be used to define carriers. Also, geometric constraints on such carriers cannot easily be solved. In this thesis, we limit carriers to planes, spheres and cylinders, with variable radii, positions and orientations. For these primitives, we can efficiently solve geometric constraints (see Chapter 4).

The topological variables of a DFOM are called constructs. A construct is a variable that represents a point set, corresponding to a subspace of $\mathbb{R}^3$. The actual point set represented by a construct, i.e. its value in a specific realisation, is determined by the subspace constraints and the topological constraints imposed on it.

Subspace constraints specify that a construct is a subset of the IN, ON or OUT subspace of a carrier. For example, the subspace constraint $IN(X, C_i)$, where X is a construct and C_i is a carrier, specifies that $X \subseteq S_i$ where subspace $S_i = \{p \in \mathbb{R}^3 | C_i(p) = IN\}$. The subspace constraints $ON(X, C_i)$ and $OUT(X, C_i)$ are defined similarly. Subspace constraints determine the maximal point set that can be represented by a construct. Thus, a construct X that is constrained by n subspace constraints to n subspaces $S_1, \ldots, S_n$ (of n carriers $C_1, \ldots, C_n$), represents a subset of the intersection of the subspaces, i.e. $X \subseteq S_1 \cap \ldots \cap S_n$. The exact point set represented by a construct is determined by solving the topological constraints in the model.

A construct that is not constrained to any carrier represents the space $\mathbb{R}^3$, and is generally not used in a DFOM. A construct that is constrained ON a single carrier generally represents a surface. A construct that is constrained ON two carriers generally represents one or more curves. A construct that is constrained ON three carriers generally represents a finite set of points. Finally, a construct may also be constrained IN or OUT with respect to several carriers, resulting in a point set that is bounded by the surfaces of those carriers. For example, a construct constrained IN a planar



(a) half-sphere (b) disk (c) circle (d) 2 points

Figure 3.2: Constructs built from a spherical carrier and one or two planar carriers.

Figure 3.2	construct	sphere	plane 1	plane 2
(a)	half-sphere	IN	IN	
(b)	disk	IN	ON	
(c)	circle	ON	ON	
(d)	2 points	ON	ON	ON

Table 3.1: Constructs from Figure 3.2 and the constraints relating them to carriers.

carrier and IN a spherical carrier generally represents a half-sphere volume. A construct constrained ON a planar carrier and IN a spherical carrier generally represents a disk. Some examples of systems of constructs and carriers are shown in Figure 3.2 and Table 3.1.

Note that the intersection of carrier subspaces that defines the maximal point set of a construct, is not the regularised intersection commonly used in CSG representations. The maximal point set may thus be open or closed, and bounded or unbounded. However, realisations of a DFOM are represented by a CM, which contains all maximal point sets of the constructs in the DFOM, as well as the closure of any of those point sets that are open (see Section 3.4). The represented object is the regularised union of those volume cells in the CM that contain material, as determined by solving the topological constraints in the model. Thus, even though subspace constraints may define constructs with an open topology, realisations always have a closed topology.

Constructs are determined by carriers of which the geometry can vary, and thus the point set represented by a construct can degenerate. For example, a construct defined ON a sphere and ON a plane generally represents a circle. However, for some values of the carriers, it may represent a point (if the plane is tangent to the sphere) or an empty set (if the plane does not intersect the sphere). Degenerate constructs are allowed in the model, but by imposing topological constraints, such degenerate cases can be disallowed

in realisations.

Topological constraints specify topological properties of constructs in relation to each other or in relation to the whole model. Thus, topological constraints determine whether a construct is completely, partially, or not at all part of a realisation. For example, a CompletelyOnBoundary constraint specifies that a surface construct must be completely on the boundary of the model. Thus, a point set will be assigned to the construct, if possible, such that this point set is completely on the boundary of the model. Solving topological constraints is further discussed in Chapter 5.

3.3 Representation of features and families

A DFOM can be schematically represented by a constraint graph. This is a bi-partite graph, of which nodes are either variables or constraints, and edges only run between variable nodes and constraint nodes. An edge between a particular constraint and variable indicates that the constraint is imposed on that variable.

The constraint graph of an example DFOM, representing a family of blocks, is shown in Figure 3.3. In this figure, rectangles represent variables and rhombuses represent constraints. Note that the roles (or order) of variables in constraints are not indicated. In most cases, constraints are either symmetrical, or the role of constraint variables can be inferred from their type. The example model defines several planar carriers, e.g. `bottom` and `top`, related by geometric constraints, e.g. the distance between `bottom` and `top` is equal to a real number variable `height`. A volume construct called `block` is related with `IN` constraints to the six carriers.

For modelling more complex families of objects, it should be possible to create a DFOM using features, which capture semantics at a higher level of abstraction. A more or less accepted definition of a feature is that it is a representation of shape aspects of a product that are mappable to a generic shape and functionally significant for some product life-cycle phase [Shah and Mantyla, 1995]. In many feature models, for example [Hoffmann and Joan-Arinyo, 1998] and [van den Berg et al., 2003], features are implemented as feature classes, from which feature instances can be generated, which are then added to the model.

In a pure declarative model, there is essentially no difference between feature classes, i.e. definitions of features, and feature instances, i.e. features in a model. Because a declarative model may have several realisations, a feature instance in such a model may also have several realisations. In fact, a feature in a model may even have exactly the same (infinite) set of realisations as the feature class from which it was instantiated. However, a feature in a model typically has fewer realisations, or just one realisation, because of other constraints in the model.

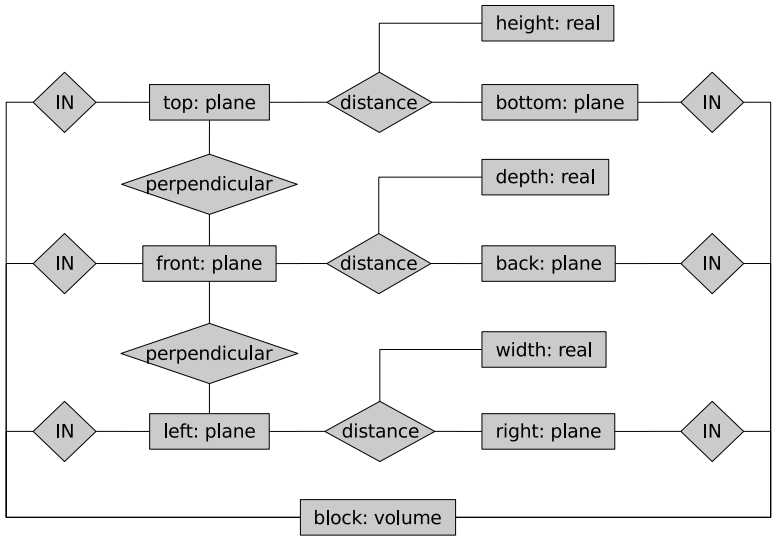


Figure 3.3: Constraint graph of a DFOM that defines a family of blocks. Rectangles represent variables and rhombuses represent constraints.

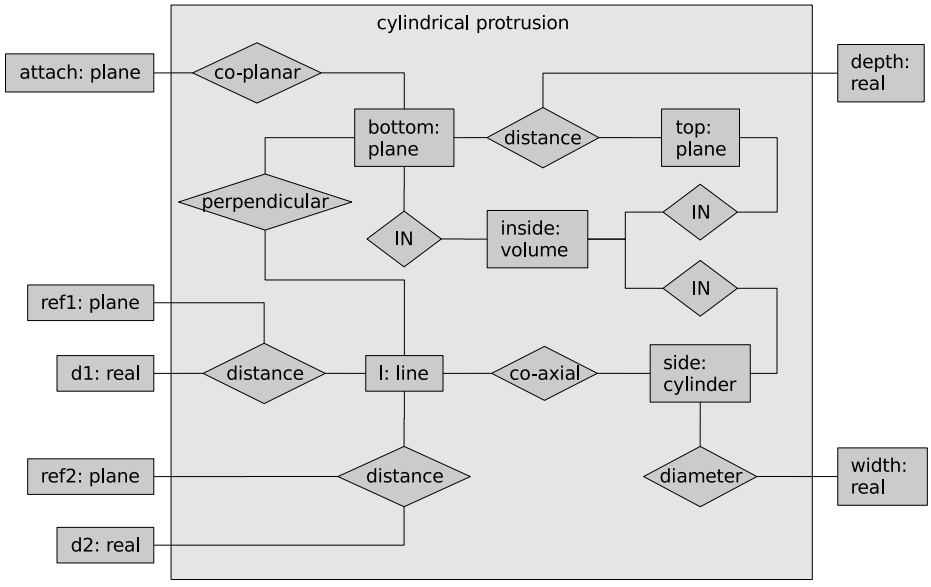


Figure 3.4: Constraint graph of a DFOM that defines a cylindrical protrusion feature.

A feature can thus also be considered a family of objects, although by itself it may not represent a family of solid objects. For example, a solid block is often used as a base feature, and is also a family of solid objects by itself. On the other hand, a cylindrical blind hole, for example, is not a solid object. It is, however, normally attached, via constraints, to another feature, e.g. a solid block, from which it removes material, and together they form a family of solid objects.

We simply define a feature as a specific subset of the variables and constraints in a DFOM. The definition of a cylindrical protrusion feature is shown in Figure 3.4. Here, variables and constraints inside the `cylindrical protrusion` rectangle are part of the feature. To instantiate such a feature in another DFOM, we copy the feature definition, i.e. the specified subset of the variables and constraints, into that DFOM. The variables and constraints in the feature definition are thus added to the target model when the feature is added to it, and removed from it when the feature is removed.

A feature should represent a volumetric shape, and should therefore contain at least one variable representing a volume construct. For some feature shapes, it may be necessary to define several volume constructs. Additional, non-volumetric constructs may also be needed, e.g. constructs representing the boundary of the feature. Carriers and constructs are used to represent the canonical shape of features, i.e. the shape of a feature without considering its interaction with other features. The actual geometry and topology of a feature in realisations depends on the geometric and topological constraints imposed on the carriers and constructs of the feature.

Typically, a feature definition contains constraints of which one or more variables are not included in the definition. This set of variables and constraints is called the *interface* of the feature. When instantiating a feature, the interface variables are not copied to the target model (since they are not in the set that defines the feature), but instead variables from the target model, specified by the user, are used in their place, such that the interface constraints of the feature are imposed on those variables.

The feature interface can be used to specify how a feature will be attached to other features in the model. For example, consider the DFOM shown in Figure 3.4, representing a cylindrical protrusion feature. This feature requires that its bottom face is coincident with an existing face in the model. Thus, the feature contains a co-planar constraint, between the bottom face and a face outside the feature, i.e. the attach face. When the feature is instantiated in a model, the user must specify which face in the model is used as the attach face. The feature interface in the example also specifies two positioning faces, which are used to constrain the position of the axis of the cylindrical protrusion feature.

A DFOM is basically a system of variables and constraints, plus a collection of disjoint subsets of those variables and constraints, representing

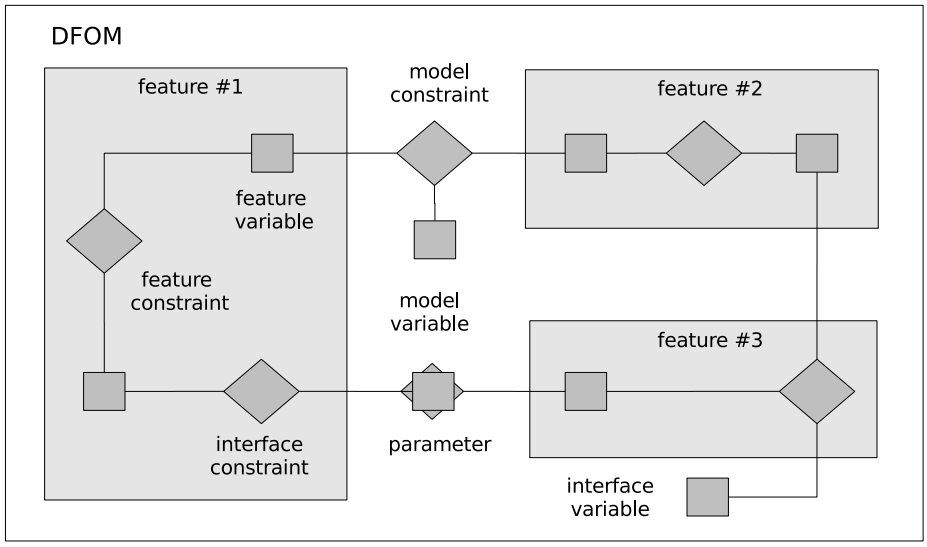


Figure 3.5: Schematic representation of DFOM, showing several features, feature constraints and variables, interface constraints and variables, model constraints and variables, and a parameter.

features. A schematic representation of a DFOM with several features is shown in Figure 3.5. Constraints and variables can be classified as model constraints and variables, feature constraints and variables, and interface constraints and variables. Model constraints and variables are those which are not in any feature. Feature constraints and variables are part of a feature. Feature constraints that have variables outside the feature are interface constraints, and the latter variables are interface variables.

A parameter is an aggregation of a variable and a value constraint, represented in Figure 3.5 by a combination of a rectangle and a rhombus. The user can introduce new parameters, e.g. to replace interface variables when instantiating a feature, or create a parameter by adding a value constraint to a variable. The value of a parameter can only be changed by the user, not by the modelling system.

The subsets that define features in a model must be disjoint, i.e. no variable or constraint may be in more than one feature. This ensures that features can be safely removed from a model without affecting the semantics of other features. A model may, however, also contain variables or constraints that are not part of any feature, i.e. model variables and constraints. This allows the user to add construction geometry, e.g. datum planes, and additional constraints between features.

New feature types can be defined by simply creating a new model, from carriers, constructs, and geometric and topological constraints, and then specifying which variables and constraints are inside the new feature. Also,

it is possible to edit features in a model, or to re-use parts of models as features.

The representation of a DFOM can be summarised by the following grammar:

model = set of variables + set of constraints + set of features
feature = set of variables + set of constraints
variable = carrier | construct | real (etc.)
constraint = geometric | subspace | topological (etc.)
carrier = plane | sphere | cylinder (etc.)
construct = volume | surface | curve | point
geometric = distance | angle | coincident (etc.)
subspace = IN | OUT | ON
topological = nature | boundary | interaction (etc.)

This grammar does not list all possible types of constraints and variables that can be used in a DFOM. Other types of geometric variables and constraints can be supported by our geometric constraint solver, discussed in Chapter 4. Topological constraints in the DFOM include boundary constraints, interaction constraints or nature constraints. These constraints are imposed on the constructs of a model, corresponding to volumes (nature and interaction constraints) or surfaces (boundary constraints). These constraints, and other types of constraints supported by our topological constraint solver, are discussed in Chapter 5.

Other types of variables and constraints, e.g numerical variables and algebraic constraints, can also be included in the model. One approach to solving such systems of mixed types of constraints, is to use different specialised solvers alternately. In each iteration, from the solutions of one solver, parameter values may be inferred that can be used by another. This alternating iteration is continued until a complete solution has been found, until no more parameter values can be inferred, or until some iteration maximum has been reached. In general, however, systems of mixed constraints cannot be solved efficiently, and are not further considered in this thesis.

3.4 Realisations

A DFOM does not explicitly represent the geometry and topology of its members. Instead, a number of *realisations* may be derived from a DFOM by a process called *interpretation*. A realisation is represented by a cellular model (CM), with a value assigned to each volume cell, specifying whether the cell contains material, i.e. whether it represents a solid part of the object. The represented object is the regularised union of all the cells that contain material. From the CM we can also determine all faces, edges and vertices that are part of the represented object, which may be used for visualisation

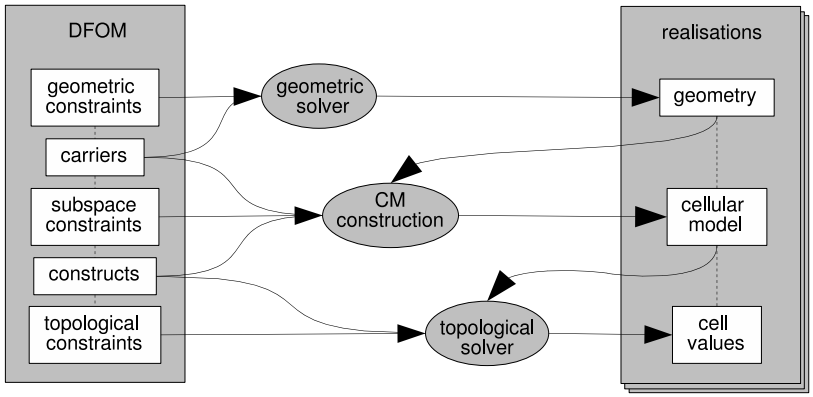


Figure 3.6: The interpretation of a DFOM consists of a geometric solving step, a CM construction step, and a topological solving step.

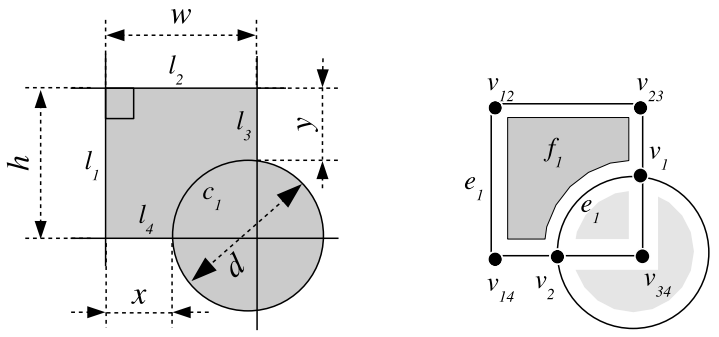
and analysis. The interpretation process guarantees that all constraints specified in the DFOM are satisfied in the realisations it finds, thus these realisations represent valid family members.

The interpretation of a DFOM starts with solving the geometric constraints on the carriers in the DFOM. This yields a number of geometric solutions. In each geometric solution, the geometry of the carriers has been be determined. With this carrier geometry, and the subspace constraints relating carriers and constructs, a CM is constructed. Then, the system of topological constraints is solved, which yields a number of topological solutions. Each topological solution specifies for each cell in the CM whether it contains material. This is illustrated in Figure 3.6.

Typically, if a DFOM represents a family of objects, the system of geometric constraints will be underconstrained, i.e. it has one or more degrees of freedom. This occurs when, for example, some dimensions of some features have not been specified. A system that is underconstrained has an infinite number of solutions, and these cannot be represented explicitly. Implicitly, the model represents all the realisations that would be obtained if we could generate all the geometric solutions and interpret them further.

Only if the geometric system has a finite number of solutions, then the geometric solutions can be generated explicitly and interpreted further. However, the number of solutions can be very large, exponential to the number of geometric constraints in the system. The number of geometric constraints for a typical model with just a few features is already so large that it is not desirable to generate all geometric solutions. Therefore, features should be defined in such a way that if the system is well-constrained, the number of geometric solutions is low, preferably just one. This can be done using various solution selection mechanisms, described in Section 4.5.

When the geometric constraint system has been solved, the geometry of



(a) DFOM (b) CM

Figure 3.7: The 2D model in (a) has two features: a box, defined by four linear carriers $l_1 ... l_4$ and a disk defined by circular carrier c_1. The CM in (b) shows several topological entities: vertices are labelled v_*, edges e_* and faces f_*

.

all carriers has been determined. The geometry of all constructs can also be determined, by evaluating the subspace constraints. Thus the geometry of a construct can be determined by intersecting carrier geometry. A cellular model is constructed that contains a cell for every volume construct or intersection of volume constructs. The boundaries of cells are represented by faces, edges and vertices. A 2D example DFOM and the corresponding CM are shown in Figure 3.7.

Unlike constructs, the entities in a cellular model should be connected point sets. If a construct represents a connected point set, it can be represented by a single entity, otherwise it should be represented by several entities. Also, for every entity in the CM that represents an open point set, the CM must also contain entities that represent the closure of that point set. Thus, a volume construct, which describes an open point set, is represented by a cell in the CM, and its boundary is also represented in the CM, by a number of faces. Finally, the entities in the CM must be disjoint, i.e. they may not intersect each other.

The CM can be generated efficiently by mapping constructs to entities and then adding these entities to the CM one at a time [Bidarra et al., 2005b]. If an added entity intersects with an entity already in the CM, the entities are split into non-intersecting entities, and new entities are added that represents the intersection of the entities. For each entity in the CM, a list of references is kept to the original constructs from which it was derived. This allows us to relate the topology of realisations to the constraints in the DFOM. This information is used for solving topological constraints (Chapter 5) and tracking topological changes (Chapter 7).

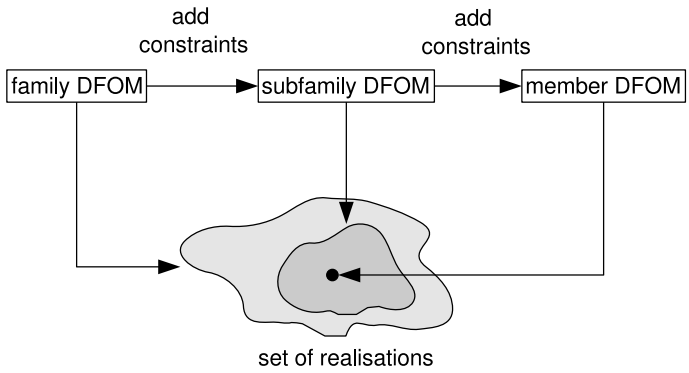


Figure 3.8: Specialisation and instantiation. By adding constraints to a DFOM, subfamilies and members are obtained. The realisations of a subfamily, and the realisation of a member, are in the set of realisations of the original family.

3.5 Family membership and subfamilies

The interpretation of the DFOM, as described above, guarantees that realisations satisfy all constraints, and therefore have the semantics specified by the model. However, because the set of realisations can be infinite, we cannot, in general, verify family membership by generating and comparing realisations. Therefore, family membership is defined in terms of DFOMs, as follows.

A DFOM M represents a member of the family represented by a DFOM F, if and only if

- M has the same set of variables as F,
- M has a superset of the constraints of F, and
- M has exactly one realisation.

In other words, members of a family are instantiated by adding more constraints, until the number of realisations is just one. Subfamilies are defined in a similar way: they are also represented by models with extra constraints, but can have more than one realisation. Thus, adding constraints to a DFOM is equivalent to specialisation of a family. Instantiation is essentially the same as specialisation; members are subfamilies with just one realisation. Models with no realisation at all are invalid.

The relations between DFOMs representing families, subfamilies and members, and the relations between the realisations of these DFOMs, are illustrated in Figure 3.8. To better understand these relations, consider that a subfamily DFOM is defined by a superset of constraints, and because

more constraints have to be satisfied, the set of realisations of the subfamily DFOM is a subset of the set of realisations of the family DFOM. A member DFOM is a subfamily with one realisation, which is thus guaranteed to be in the set of realisations of the family.

For every possible realisation of a family DFOM, a member DFOM can always be found, simply by adding constraints. It is easy to see that if any variable in a DFOM has degrees of freedom, then a value constraint can be imposed to determine the variable. However, in practice, finding such a value constraint, or some other constraint, such that there are no conflicts with other constraints can be difficult (see Chapters 6 and 7).

To verify family membership, we need to compare systems of variables and constraints, and we need to determine whether a model has zero, one, or more realisations. The geometric constraint solver used here (see Chapter 4) is able to identify overconstrained and underconstrained situations corresponding to zero realisations and an infinite number of realisations, respectively, and can determine any finite number of realisations if needed. The topological constraint solver (see Chapter 5) can also generate the finite set of all topological solutions, and thus we can determine the number of solutions of a DFOM in general.

Sets of features and constraints can easily be compared if features and constraints are uniquely identified by a name. This is only the case if one model is directly derived from another. To verify membership for models from another source, we can only consider the types of variables and constraints, and the associations between them. In that case, the membership test is equivalent to graph matching, for which many algorithms are known, e.g. [Ullmann, 1976].

3.6 Implementation

Our implementation of the DFOM is based on Spiff, a feature modelling system developed at Delft University of Technology. Spiff originally implemented the Semantic Feature Model (SFM, see Section 2.4). Because the DFOM is based on concepts of the SFM, many modules of the modelling system could be re-used, and the same types of features and types of constraints are supported.

The topological constraints defined by the SFM, i.e. boundary constraints and interaction constraints, can also be used in the DFOM. Boundary constraints specify that a surface construct is partially or completely on the boundary of the model, or is not on the boundary, partially or completely. Interaction constraints specify that some volumetric interaction should not occur (see Table 2.1). Such constraints were imposed on features in the SFM, but are imposed on volume constructs in the DFOM. The topological constraints available in our implementation of the DFOM

constraint name	description
NATURE(V,ADDITIVE)	V has additive nature
NATURE(V,SUBTRACTIVE)	V has subtractive nature
COMPLETELYONBOUNDARY(S)	S is completely on the boundary
PARTIALLYONBOUNDARY(S)	S is partially on the boundary
COMPLETELYNOTONBOUNDARY(S)	S is completely not on the boundary
PARTIALLYNOTONBOUNDARY(S)	S is partially not on the boundary
NOSPLITTING(V)	V has no disconnected subsets
NODISCONNECTION(V)	V is not disconnected from the model
NOOBSTRUCTION(V)	V is completely free of material
NOCLOSURE(V)	V is not a closed void
NOABSORBTION(V)	V is not absorbed

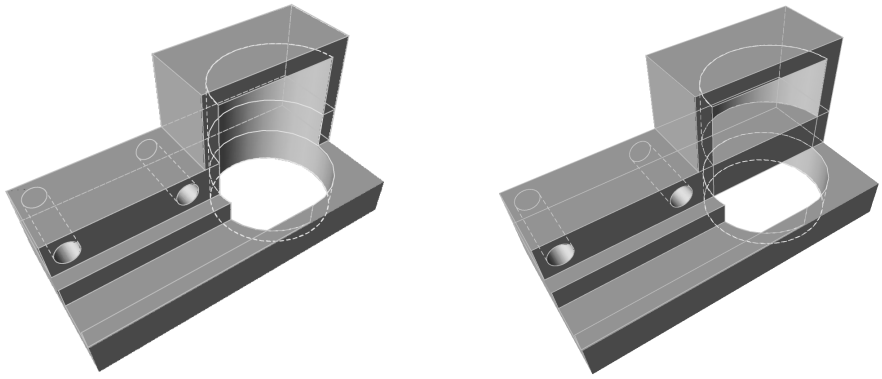
Table 3.2: Topological constraints in the implementation of the DFOM. Here V represents a volume and S represents a surface.

are listed in Table 3.2. These constraints can be used to specify the topological semantics of features, when used as feature constraints, and additional topological semantics of a family when used as model constraints. More details about these constraints can be found in Chapter 5.

The nature of features can still be specified, now using nature constraints. A constraint NATURE (V, ADDITIVE) or NATURE (V, SUBTRACTIVE) specifies that the nature of a feature volume V is additive or subtractive, respectively. However, unlike in the SFM, features do not necessarily have a specific nature, i.e. no nature constraints have to be imposed on a feature volume.

The most important modification to SPIFF concerns the interpretation process, which determines whether cells in the cellular model contain material. Originally, from a partial feature precedence order, determined by feature dependency analysis, a strict feature precedence order was derived, by giving precedence to features that were more recently created or edited. For each cell in the cellular model, whether it contains material was then determined by the nature of the feature with the highest precedence in the owner list of that cell. Topological constraints were only verified by the system after the cellular model had been evaluated.

With the new interpretation, cells values are determined by nature constraints only if feature dependency analysis yields a strict ordering for the features in the owner list of the cell. If this is not the case, then other topological constraints determine the cell's material value. To determine those cell values, the complete system of topological constraints is solved (see Chapter 5). Because of this, a DFOM may have zero, one or more realisations, whereas for a SFM, always a single realisation was found. The modelling system textually displays the total number of realisations of the model, and graphically displays one of the realisations chosen by the user.



(a) (b)

Figure 3.9: Two realisations of a single DFOM in SPIFF.

In Figure 3.9, two realisations of a DFOM created in SPIFF are shown. The model contains a large blind hole that has a PARTIALLYONBOUNDARY constraint on its bottom face. The user can choose one of the realisations, or add constraints to reduce the number of realisations. If a NOOBSTRUCTION or NOSPLITTING constraint is added to the blind hole, only realisation (a) will be found, because in model (b) the boundary of the blind hole is split, and the volume of the hole is obstructed.

If a model is invalid, i.e. it has no realisations, then the user is informed which constraints cannot be satisfied, and the affected features are highlighted. Note that to visualise such a model, some "nearly satisfied" realisation should be generated. Currently, such a realisation is determined using only feature precedence, i.e. the model is interpreted as a SFM. From this realisation, it is determined which constraints are not satisfied and which features are involved. The user is presented with a dialog that gives options to restore model validity by changing or removing the features involved.

Altogether, the DFOM described in this section allows families of objects to be specified with clear semantics. To instantiate members of such a family, we need geometric and topological constraint solving methods. Such methods are presented in the next two chapters.

Chapter 4

Geometric constraint solving

In the DFOM, the geometric properties of a family are specified using geometric constraints. To determine the realisations of a DFOM, and to decide family membership of DFOMs, such systems of geometric constraints must be solved. A geometric constraint solver is needed that satisfies the following requirements:

- it should be able to determine solutions for systems of geometric constraints on carriers; in particular, it must solve distance, angle and coincidence constraints on points, lines, planes, spheres and cylinders
- it should be able to determine whether a system is well-constrained, underconstrained or overconstrained, and consistent or inconsistent
- it should be able to find all solutions of a well-constrained system, or any particular, a-priori specified, solution
- it should be fast enough to be used for interactive modelling, and for parameter range computation (see Chapter 6); in particular, it must be able to efficiently re-compute the solutions after incremental changes to a system.

A new geometric constraint solving approach is presented in this chapter that satisfies these requirements. The approach can also be used to solve geometric constraints in other applications. It is particularly suitable for solving problems that cannot be decomposed into rigid subproblems, and efficient at solving problems many times with incremental changes.

Parts of this chapter have already been published in [van der Meiden and Bronsvoort, 2008] and [van der Meiden and Bronsvoort, 2005b].

4.1 Introduction

Geometric constraints are used in current CAD systems to specify dimensions in 2D sketches, and to position parts in 3D assembly modelling. In some declarative modelling approaches, e.g. semantic feature modelling and the DFOM, presented in the previous chapter, geometric constraints are used to define feature classes and families of objects.

Note that geometric constraints in a DFOM may have variables, e.g. distance and angle variables, on which other types of constraints, e.g. algebraic constraints, are imposed. Such systems of mixed type constraints cannot be solved by a specialised geometric constraint solver, such as presented here. It is assumed that all variables of geometric constraints are geometric variables. Values of parameters of the geometric constraints, i.e. distances and angles, must have been determined before the geometric constraint system is solved.

In principle, geometric constraint problems can be considered as algebraic problems. However, generic algebraic solving methods are either too expensive, or they are incomplete, i.e. they cannot find all solutions for a given problem. Solving large constraint systems using symbolic algebraic methods is too expensive. Known methods, e.g. methods based on characteristic sets, such as Wu's method [Wu, 1986], have exponential running times, in relation to the number of constraints. Numerical methods, such as Newton-Raphson iteration, are not useable either, because these methods, although fast, cannot find all solutions to a given problem. Homotopic continuation techniques have been used to find all solutions for small problems, e.g. the octahedral problem [Durand and Hoffmann, 2000]. However, for larger problems, this technique is also too expensive, because the number of homotopy paths grows exponentially.

The most successful geometric constraint solvers are so-called constructive solvers, which determine a decomposition of a problem into generically rigid subproblems, known as clusters. These clusters are solved independently, and the solutions of the clusters are used to construct a solution for the complete problem. The advantages of using constructive solvers in CAD, and requirements for such solvers, are discussed in [Hoffmann et al., 2001a].

For 2D problems, generic rigidity is characterised by Laman's theorem [Laman, 1970]. This theorem formulates generic rigidity for graphs, where edges in the graph correspond to distance constraints and vertices correspond to point variables, as follows:

> Let a graph G have exactly $2n-3$ edges, where n is the number of vertices in G. Then G is generically rigid in $\mathbb{R}^2$ if and only if $e' \leq 2n'-3$ for every subgraph of G with n' vertices and e' edges.

In other words, Laman's rule states that a 2D constraint problem with n points and e distance constraints is generically rigid, or well-constrained, if and only if $e = 2n - 3$, and for every subproblem with n' points and e' distance constraints, $e' \leq 2n' - 3$. If the number of constraints is smaller than specified by the rule, the system is called underconstrained. If, for any subproblem, the number of constraints is larger, the system is overconstrained. Note that a well-constrained system is not necessarily consistent, i.e. depending on the actual parameter values of the constraints there may or may not be any solutions. Also, overconstrained systems are not necessarily inconsistent; for some values of the constraint parameters, the system may have solutions, and is called consistently overconstrained.

The simplest constructive solving approach is the cluster rewriting approach, also referred to as the bottom-up approach. In this approach, patterns of geometric constraints that are known to be generically rigid, are recognised as clusters. Certain patterns of clusters are also recognised and merged into larger clusters. If the problem is well-constrained, a single cluster will remain at the end of the rewriting process.

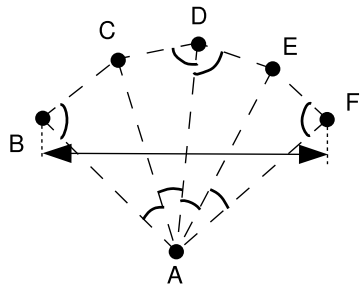
Most 2D solvers use a variant of the cluster rewriting approach with optimised data structures for representing systems of clusters, e.g. the graph-constructive approach used in [Bouma et al., 1995; Hoffmann and Vermeer, 1995]. For some of these algorithms a proof of correctness has been given, e.g. [Fudos and Hoffmann, 1996; Joan-Arinyo and Soto, 1997], showing that if a system is reduced to a single cluster, then this cluster represents a correct solution for the original system.

The cluster rewriting approach is not complete, because no set of rewrite rules is known that will reduce all well-constrained systems, even if they consists only of distance constraints, to a single rigid cluster. In practice this means that some well-constrained systems may not be solved, and may be classified incorrectly as underconstrained.

Using Laman's theorem, a complete top-down decomposition algorithm can be devised for 2D problems, e.g. as suggested in [Hoffmann et al., 2001a; Joan-Arinyo et al., 2004]. However, such a complete algorithm is expensive, and cannot deal with consistently overconstrained situations.

Some 3D solvers based on the cluster rewriting approach have been presented in [Hoffmann and Vermeer, 1995; Durand and Hoffmann, 2000]. However, in 3D, the incompleteness issues of this approach are even more severe than in 2D. Until recently, there was no 3D equivalent to Laman's theorem, but a characterisation of generic rigidity for 3D systems of distance constraints has now been found [Sitharam, 2006]. This result may lead to the development of a complete algorithm for solving systems of 3D geometric constraints. However, at the time of writing, no solving algorithm based on this result is known.

Most 3D solvers, e.g. [Kramer, 1992; Hoffmann et al., 2001b; Sitharam



(a) graphical representation

$Angle(A, B, C)$ $Angle(A, D, C)$
$Angle(A, D, E)$ $Angle(A, F, E)$
$Angle(B, A, C)$ $Angle(C, A, D)$
$Angle(D, A, E)$ $Angle(E, A, F)$
$Distance(B, F)$

(b) textual representation

Figure 4.1: System of constraints on points $[A, \ldots, F]$. An angle constraint is represented graphically by an arc between dashed lines. A distance constraint is represented graphically by a two-sided arrow.

et al., 2006; Gao et al., 2006] are based on a technique called degrees-of-freedom (DOF) analysis. The DOF-based approach creates a top-down decomposition using heuristic rules to determine the generic rigidity of a problem and its subproblems. Solving algorithms attempt to find a minimal set of rigid subproblems based on the results of the DOF analysis. In practice, DOF-based rules correctly determine well-constrainedness for many systems of constraints. However, the DOF-based approach is also not complete, since current DOF-analysis techniques are based on heuristics for well-constrainedness in 3D. In particular, DOF-based algorithms sometimes incorrectly classify overconstrained systems as well-constrained.

To be able to efficiently determine solutions for problems with incremental changes, the cluster rewriting approach is preferable. It is fast and an incremental algorithm can be easily implemented. However, the cluster rewriting approach can only be used to solve a relatively small class of problems, in particular, problems that can be decomposed into fairly small rigid subproblems. The DOF-based approach can solve a larger class of problems, but is generally more expensive, and no incremental algorithm is known. In particular when a problem cannot be decomposed into two or more rigid subproblems, the whole problem must be solved using expensive symbolic algebraic methods, which are also not incremental.

In general, problems that cannot be decomposed into rigid clusters cannot be solved incrementally and efficiently with existing approaches. Consider, for example, the 2D constraint problem in Figure 4.1. Here, we have a number of points (the variables), constrained by several angle constraints, and one distance constraint. The whole constraint system is rigid, but there is no subset of variables and constraints that forms a rigid system. A solver based on the cluster rewriting approach will not be able to solve the system, because it cannot find any rigid clusters, unless this particular system would

be explicitly programmed in the solver. Using a DOF-based approach, the system may be found to be well-constrained, but the system is considered as a single cluster and must be solved using expensive symbolic algebraic methods.

We present a new solving approach that identifies not only *rigid clusters* (corresponding to generically rigid subproblems), but also so-called *scalable clusters* and *radial clusters*, which correspond to subproblems with particular internal DOFs. The different types of clusters are introduced in Section 4.2.

Our solving approach, presented in Section 4.3, is based on cluster rewriting. The basic idea is to exhaustively apply a small set of rewrite rules to a system of rigid and non-rigid clusters. The set of clusters remaining when no more rewrite rules can be applied, represents the generic solution of the system. With this approach, we can also determine whether the system is well-constrained, underconstrained or overconstrained.

It is relatively easy to update the solution(s) of a system when changes are made to it, i.e. when values of constraint parameters are changed, or when constraints are added to or removed from the system. An efficient incremental algorithm is presented in Section 4.4.

In general, a geometric constraint problem, even if it is well-constrained, can have a large number of solutions. Typically, for applications such as the DFOM, only one or a few specific solutions are needed. In Section 4.5, we present two methods for solution selection that can be used in the presented solving algorithm: declarative solution selection and prototype-based solution selection.

The cluster rewriting algorithm can only solve systems of constraints on points, whereas in a DFOM, and many other applications, constraints are imposed on 3D primitives, e.g. planes, spheres and cylinders. Our approach to solving systems of geometric constraints on 3D primitives is discussed in Section 4.6. Basically, constraints on such primitives are mapped to a system of distance and angle constraints on point variables. This system is then mapped to a system of rigid and non-rigid clusters, which is solved. The solutions of the system of clusters are used to construct the solutions for the original problem involving 3D primitives.

4.2 Clusters

A cluster basically represents a collection of distance and angle constraints on a set of points. We define three types of clusters: *rigid clusters*, *scalable clusters* and *radial clusters*. The type of a cluster determines which distances and angles are constrained by it. Also, a set of *configurations* is associated with a cluster, each of which determines an alternative set of values for the distances and angles constrained by the cluster.

The distances $\delta(p, q)$ constrained by a cluster are defined as:

$$\delta(p, q) = \sqrt{(q - p) \cdot (q - p)}$$

and the angles $\measuredangle(p, q, r)$ as:

$$\measuredangle(p, q, r) = cos^{-1}(\frac{p - q}{\delta(p, q)} \cdot \frac{r - q}{\delta(q, r)})$$

where $p, q, r \in \mathbb{R}^2$ or $\mathbb{R}^3$ are points in the cluster.

By this definition, $\measuredangle(p, q, r) \in [0, \pi]$. In 3D, only this unsigned angle can be used. In a 2D variant of the solver, signed angles may be used instead. A signed angle $\measuredangle(p, q, r) \in [-\pi, \pi]$ is the angle of rotation to transform a unit-vector $p - q$ to a unit-vector $r - q$ in $\mathbb{R}^2$.

A *configuration* is a set of assignments of coordinates to point variables. For a set of point variables $A = [p_1, \ldots, p_n]$, each point p_i is assigned a vector v_i, and for this configuration we write: $c_A = \{p_1 = v_1, p_2 = v_2, \ldots, p_n = v_n\}$.

The actual values of the distances and angles constrained by a cluster are determined by the configurations associated with the cluster. When there are no configurations associated with a cluster, the cluster is considered unsatisfiable, i.e. there are no solutions for this cluster. If there are several configurations associated with a cluster, then these determine alternative values for the distances and angles, i.e. the distance and angle values determined by one of the configurations must be satisfied.

For a cluster with a given type and set of configurations, the distance and angle values can be determined as follows. Suppose, the type of the cluster specifies that the distance $\delta(p_1, p_2)$ is constrained, and associated with the cluster are two configurations, $c_1 = \{p_1 = (0, 0, 0), p_2 = (1, 1, 1)\}$ and $c_2 = \{p_1 = (2, 0, 0), p_2 = (1, 0, 0)\}$. Configuration c_1 specifies a constraint $\delta(p_1, p_2) = \sqrt{3}$. Alternatively, configuration c_2 specifies a constraint $\delta(p_1, p_2) = 1$. When solving a system containing this cluster, one of these constraints must be satisfied.

The system of distance and angle constraints represented by a cluster, when considered as independent constraints, is in some cases overconstrained. However, the values of these distance and angle constraints are determined by a configuration, and therefore these constraints are in fact not independent. Because a configuration assigns a point in $\mathbb{R}^3$ to each variable in the cluster, the system of constraints is always consistent (see Section 4.3). Consequently, a cluster with one or more configurations can be considered a constraint, and, at the same time, a solution for a system of constraints.

A *rigid cluster* is a constraint on a set of points $[p_1, \ldots, p_n]$, such that the relative position of all points is constrained, i.e. all distances and angles in the set of points are constrained (see Figure 4.2(a)). This type of cluster

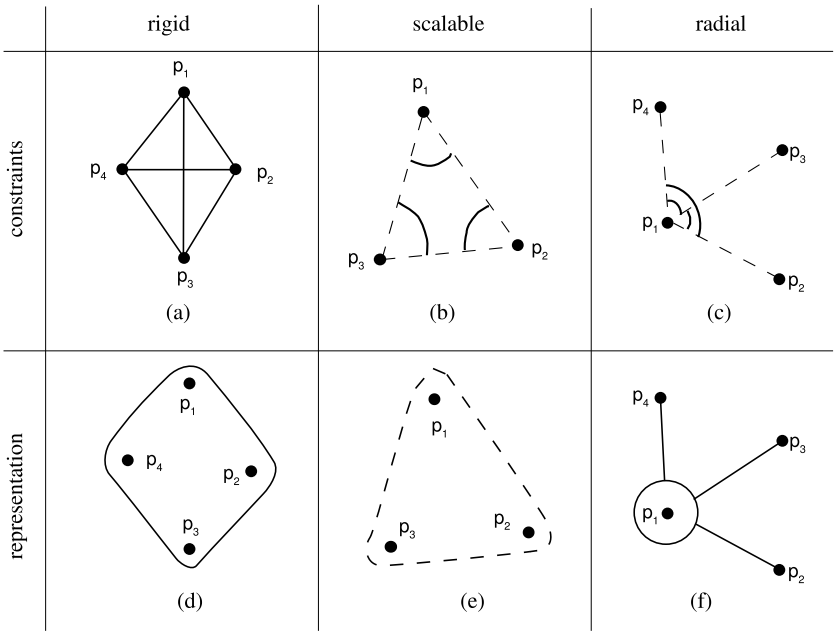


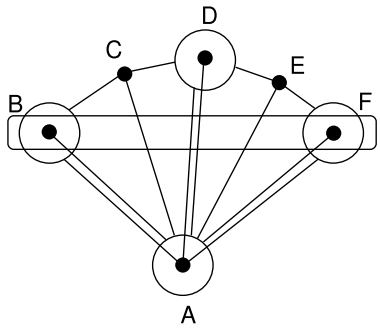
Figure 4.2: Constraints imposed by different cluster types (a,b,c) and their graphical representation (d,e,f).

has no internal DOFs, and is invariant to translation and rotation, i.e. the constraint is satisfied independently of such transformations. The notation for a rigid cluster on a set of points $[p_1, \ldots, p_n]$ is: $Rigid([p_1, \ldots, p_n])$.

A *scalable cluster* is a constraint on a set of points $[p_1, \ldots, p_n]$ such that for all $i, j, k \in [1, n]$, the angles $\measuredangle(p_i, p_j, p_k)$ are constrained (see Figure 4.2(b)) The constraint has one internal DOF, namely it may be scaled uniformly, and is invariant to translation and rotation. The notation for a scalable cluster on this set of points is: $Scalable([p_1, \ldots, p_n])$.

A *radial cluster* is a constraint on a set of points $[p_c, p_1, \ldots, p_n]$ such that for all $i, j \in [1, n]$, the angles $\measuredangle(p_i, p_c, p_j)$ are constrained (see Figure 4.2(c)). Point p_c is called the centre point and points $p_1, \ldots, p_n$ are called radial points. This constraint is invariant to translation and rotation, and has n internal DOFs (each point $p_1, \ldots, p_n$ can move along a line through the centre point). The notation for a radial cluster on these points is: $Radial(p_c, [p_1, ..., p_n])$.

We use a graphical notation for clusters, as shown in Figures 4.2(d)-(f). A point variable is represented by a dot with the name of the corresponding variable next to it. A rigid cluster is represented by a solid curve enclosing the set of points constrained by the cluster. A scalable cluster is represented by a dashed curve enclosing the set of points constrained by the cluster. Finally, a radial cluster is represented by a circle around the centre point



(a) graphical representation

$Radial(A, [B, C, D, E, F])$
$Radial(B, [A, C])$
$Radial(D, [C, A, E])$
$Radial(F, [A, E])$
$Rigid([B, F])$

(b) textual representation

Figure 4.3: The system of clusters corresponding to the problem in Figure 4.1.

and lines connecting the circle to the radial points.

Distance and angle constraints on points are easily mapped to clusters and configurations, as follows.

A distance constraint between two points is equivalent to a rigid cluster of two points with one associated configuration. For example, a distance constraint $\delta(p_1, p_2) = 1$ can be represented by a cluster $Rigid(p_1, p_2)$ and a configuration $\{p_1 = (0, 0), p_2 = (1, 0)\}$. Obviously, the choice of this particular configuration is somewhat arbitrary: infinitely many different configurations can be used to set the distance value.

An angle constraint on three points is equivalent to a radial cluster with one centre point and two radial points, and one associated configuration. For example, the angle constraint $\measuredangle(p_1, p_2, p_3) = \frac{1}{2}\pi$ can be represented by a cluster $Radial(p_2, [p_1, p_3])$ and a configuration $\{p_1 = (1, 0), p_2 = (0, 0), p_3 = (0, 1)\}$.

A system of distance and angle constraints on points can thus be mapped to a system of clusters. Figure 4.3 shows the system of clusters corresponding to the problem in Figure 4.1. Note that some angle constraints (e.g. $\measuredangle ADC$ and $\measuredangle ADE$) that are initially mapped to overlapping radial clusters (i.e. $Radial(D, [A, C])$ respectively $Radial(D, [A, E])$ have been merged (i.e. $Radial(D, [C, A, E])$). In the visual representation, such overlapping radial clusters cannot be easily distinguished, and because new angles can be inferred in such cases, the clusters are merged. This results in a simpler representation with more constraint information (see next section and Rule 1 in Appendix A).

4.3 Solving approach

To solve a system of clusters, we basically try to rewrite the system to a single rigid cluster, by exhaustively trying to apply a set of rewrite rules.

A *rewrite rule* specifies a *pattern*, describing its *input clusters*, and its *output cluster* in a generic way, and it specifies a *procedure* to determine the configurations of the output cluster from the configurations of the input clusters. A rewrite rule can be applied if a set of clusters is found in the system that matches the input clusters in the pattern. The corresponding output cluster is added to the system, and the configurations of the output cluster are determined by the procedure.

A pattern specifies a number of input clusters of a given type and a number of pattern variables. These pattern variables are matched by the solving algorithm to the point variables of the clusters in the system, such that the number of variables and the type of the cluster match. A pattern may also specify that an input cluster can match any cluster with a superset of the given variables. If a variable name occurs several times in the pattern, it must be matched with a single point variable that is constrained by several clusters in the constraint system.

To determine the configurations of the output cluster of a rewrite rule, the procedural part of the rule is applied for every combination of input cluster configurations. Suppose, for example, that two clusters are used as the input of a rewrite rule, and that each cluster has two configurations associated with it, then four different configurations for the output cluster are computed.

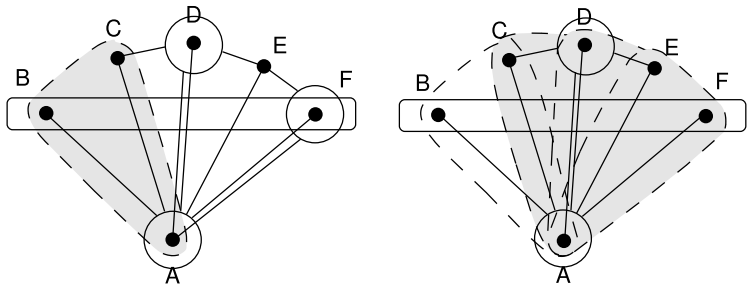
A set of rewrite rules for 2D and 3D problems is given in Appendix A. Of the 14 rules in total, 3 rules are specific for 2D problems, 6 rules are specific for 3D problems, and 5 more can be used in both 2D and 3D.

In the remainder of this section, we show how these rewrite rules are used to solve the problem illustrated in Figure 4.3. Consider Rule 7 from Appendix A.

Rule 7 Derive a scalable cluster from two radial clusters

Pattern: $Radial(p_1, [p_3, p_2, \ldots]) \cup Radial(p_2, [p_1, p_3, \ldots])$
$\rightarrow Scalable([p_1, p_2, p_3])$
Procedure: $c_1 \times c_2 \rightarrow c_R$
$c_R(p_1) = (0, 0, 0)$
$c_R(p_2) = (1, 0, 0)$
$c_R(p_3) =$ intersection
ray from $c_R(p_1)$ direction $\measuredangle(c_1(p_3), c_1(p_1), c_1(p_2))$
ray from $c_R(p_2)$ direction $\measuredangle(c_2(p_1), c_2(p_2), c_2(p_3))$

This rule can be applied (in 2D or 3D) when two radial clusters (A and



(a) single application of Rule 7

(b) repeated application of Rule 7

(c) after applying Rule 3

(d) after removing redundant clusters

Figure 4.4: Intermediate results for solving the system in Figure 4.3.

B) share three points, including the centre point of each cluster. When a match is found, a new scalable cluster (R) is added to the system, and a configuration (c_R) is computed by intersecting two rays (directed half-lines).

This rule can be applied to the problem in Figure 4.3, as follows. We find the following matches:

$$Radial(A, [B, C, D, E, F]) \cup Radial(B, [A, C]) \rightarrow Scalable([A, B, C])$$
$$Radial(A, [B, C, D, E, F]) \cup Radial(D, [C, A, E]) \rightarrow Scalable([A, C, D])$$
$$Radial(A, [B, C, D, E, F]) \cup Radial(D, [C, A, E]) \rightarrow Scalable([A, D, E])$$
$$Radial(A, [B, C, D, E, F]) \cup Radial(F, [A, E]) \rightarrow Scalable([A, E, F])$$

By applying the rewrite rule to the first match, the system shown in Figure 4.4(a) is obtained. Repeated application of the rule for all the matches listed above, results in the system shown in Figure 4.4(b).

When a rewrite rule is applied, the input clusters may become *redundant* and should be removed from the system. A cluster is redundant if all distances and angles constrained by the cluster are also constrained by newer clusters. Thus, an input cluster is removed from the system if all the distances and angles in the cluster are also in the output cluster.

In the system in Figure 4.4(a), the cluster $Radial(B, [A, C])$ is redundant and removed, because the angle $\measuredangle ABC$ constrained by this cluster, is also constrained by the newer cluster $Scalable([A, B, C])$. The cluster $Radial(A, [B, C, D, E, F])$, however, is not removed, even after repeated application of the rewrite rule (result shown in Figure 4.4(b)), because it constrains angles that are not in any of the scalable clusters (e.g. $\measuredangle BAF$).

If a rewrite rule is defined such that its output cluster contains all distances and angles that are in its input clusters, then all input clusters are removed after the rule has been applied, and we say that the rewrite rule *merges* the input clusters. The scalable clusters in Figure 4.4(b) can be merged using Rule 3 from Appendix A.

Rule 3 Merge two scalable clusters with two shared points

Pattern: $Scalable(A = [p_1, p_2, \ldots]) \cup Scalable(B = [p_1, p_2, \ldots])$
$\rightarrow Scalable(A \cup B)$
Procedure: $c_1 \times c_2 \rightarrow c_R$
T = rotation, translation and scaling such that p_1 and p_2 in c_2 are mapped onto p_1 and p_2 in c_1
$c_R = c_1 \cup T(c_2)$

This rule basically takes two scalable clusters, and combines their configurations by rigidly transforming one of them such that the shared points between the configurations coincide. New configurations obtained in this way are associated with a new scalable cluster. The rule can be applied repeatedly, in the example problem, as follows:

$$Scalable([A, B, C]) \cup Scalable([A, C, D]) \rightarrow Scalable([A, B, C, D])$$
$$Scalable([A, D, E]) \cup Scalable([A, E, F]) \rightarrow Scalable([A, D, E, F])$$
$$Scalable([A, B, C, D]) \cup Scalable([A, D, E, F]) \rightarrow Scalable([A, \ldots, F])$$

Applying these rewrites results in the system shown in Figure 4.4(c). Now we can remove the clusters $Radial(A, [B, C, D, E, F])$ and $Radial(D, [C, A, E])$, because all angles in those clusters are also constrained by the newer cluster $Scalable([A, B, C, D, E, F])$, resulting in Figure 4.4(d). Finally the cluster $Scalable([A, B, C, D, E, F])$ can be merged with $Rigid([B, F])$, using the following rule from Appendix A.

Rule 8 Derive a rigid cluster from a scalable and a rigid cluster with two shared points

Pattern: $Scalable(A = [p_1, p_2, \ldots]) \cup Rigid([p_1, p_2, \ldots]) \rightarrow Rigid(A)$
Procedure: $c_1 \times c_2 \rightarrow c_R$
T = scale configuration by $\frac{\delta(c_2(p_2), c_2(p_1))}{\delta(c_1(p_2), c_1(p_1))}$
$c_R = T(c_1)$

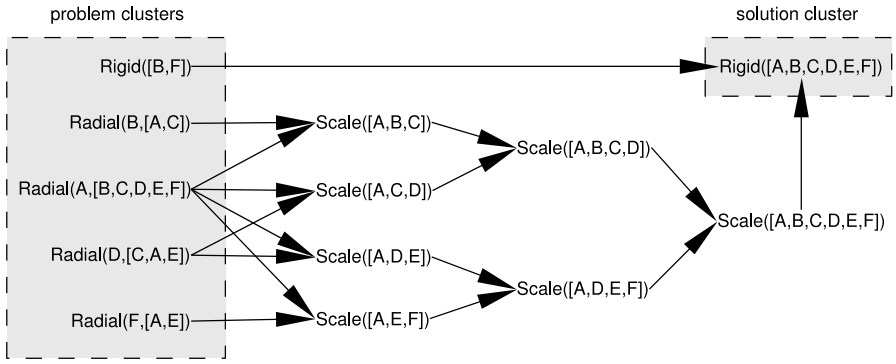


Figure 4.5: Generic solution for the problem in Figure 4.3. Note that $Scale(\ldots)$ represents a scalable cluster.

This rule basically scales the configuration of the scalable input cluster, such that the distance between the shared points becomes equal to the distance specified by the configuration of the rigid input cluster. New configurations obtained in this way are associated with a new rigid cluster.

The rule can be applied to the example system, resulting in a cluster $Rigid([A, B, C, D, E, F])$, which constrains all the variables of the problem. No other rewrite rules can be (nor need to be) applied.

The *generic solution* of a problem is represented by a directed acyclic graph (DAG) of clusters and rewrite rules. The generic solution of the problem of Figure 4.3 is shown in Figure 4.5. In this figure, arrows indicate dependencies between clusters created by rewrite rules (the rules are not explicitly represented). The clusters in the generic solution can be classified as *problem clusters*, i.e. the clusters specified in the original problem, intermediate clusters, and *solution clusters*, i.e. the clusters that are not used as input for any rewrite rule. In Figure 4.5, the clusters with no incoming arrows are problem clusters, and the cluster with no outgoing arrows is the solution cluster.

The configurations associated with the solution cluster are the *particular solutions* of the problem. If there are no configurations associated with the solution cluster, then the problem is *inconsistent*. If there are one or more configurations, then the problem is *consistent*.

From its generic solution, we can also determine whether a problem is underconstrained, overconstrained or well-constrained:

- A problem is *underconstrained* if its generic solution has more than one solution cluster or a single non-rigid solution cluster.

- A problem is *overconstrained* if any distance or angle constraint has more than one *source cluster*. This is elaborated below.

- A problem is *well-constrained* if it is not underconstrained and not overconstrained. Note that these conditions are not mutually exclusive.

If any distance or angle is constrained in two or more clusters in the generic solution, then the system may be overconstrained, depending on which clusters these distances and angles occur in.

In particular, if two or more problem clusters, i.e. clusters determined from constraints specified by the user, constrain the same distance or angle, then those distances and angles are generally not consistent, and thus the problem is overconstrained.

However, if some distance or angle is constrained in several intermediate clusters or solution clusters, then the problem is not necessarily overconstrained. The values of this distance or angle in different clusters, are derived by rewrite rules from the original problem clusters, and these values may in fact be the same in all clusters.

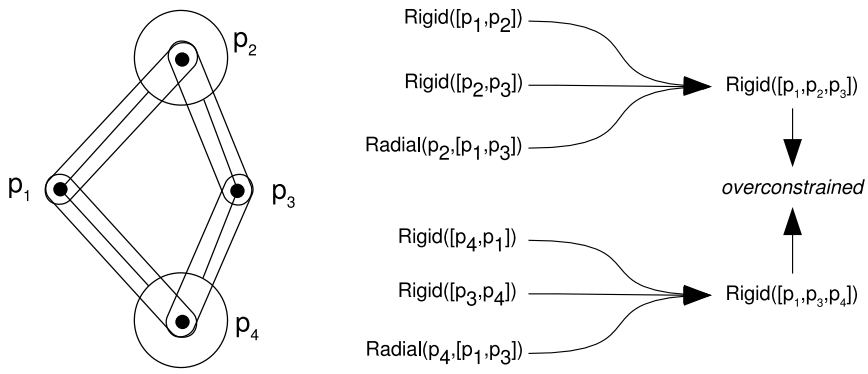
To determine whether a problem is overconstrained, we use the following procedure. When adding new clusters to the generic solution, we determine, for each distance or angle in that cluster, its *source clusters*, i.e. the first clusters in the generic solution that constrain that distance or angle.

The source cluster of a distance or angle in some cluster can be found by following dependencies in the generic solution in the reverse direction, checking for each cluster encountered whether it already constrains that distance or angle. A cluster that constrains a given distance or angle is a source cluster if it does not depend on any cluster that constrains that distance or angle.

If there is exactly one source for each distance or angle, then the system is not overconstrained, because each rewrite rule ensures that all distance/angle constraints in its input clusters are also satisfied in its output clusters. Otherwise, if there is more than one source for a distance or angle, then there is no guarantee that it will have the same value in different clusters, and therefore the system is overconstrained.

During the cluster rewriting process, sets of clusters may be created by the solver that constrain the same distances and angles, but do not result in an overconstrained system. Consider, for example, the system in Figure 4.4(c). Here, all the angles in the clusters $Radial(A, [B, C, D, E, F])$ and $Radial(D, [C, A, E])$ are also constrained by the cluster $Scalable([A, B, C, D, E, F])$. However, from the generic solution in Figure 4.5, we can infer that for each overconstrained angle, there is only one source, in these cases a single problem cluster. Thus, the system is not overconstrained.

In Figure 4.6(a), an overconstrained problem is shown. The generic solution for this problem is shown in Figure 4.6(b). Here, the clusters $Rigid([p_1, p_2, p_3])$ and $Rigid([p_1, p_3, p_4])$ both constrain the distance $\delta(p_1, p_3)$.



(a) problem (b) generic solution

Figure 4.6: An overconstrained 2D problem and its generic solution.

So, this distance is determined twice, by different rewrite rule applications, using different input clusters that do not constrain this distance. Thus, there are two source clusters for the distance: $Rigid([p_1, p_2, p_3])$ and $Rigid([p_1, p_3, p_4])$, and therefore the problem is overconstrained.

4.4 Incremental algorithm

The solving algorithm incrementally updates the generic solution of a constraint problem whenever changes are made to the problem, i.e. clusters are added or removed. The generic solution (symbol `G` in Algorithms 4.1, 4.2 and 4.3) is represented by a bi-partite graph, in which nodes are clusters or rewrite rules. Directed edges connect clusters and rewrite rules, i.e. if a cluster is an input cluster of a rewrite rule, then there is an edge from the cluster to the rewrite rule. If a cluster is the output cluster of a rewrite rule, then there is an edge from the rewrite rule to the cluster.

The solving algorithm also keeps track of the set of *active clusters*, i.e. the clusters that represent the problem after all rewriting steps so far (symbol `A` in Algorithms 4.1, 4.2 and 4.3).

The generic solution and the active set are initially empty. When the user adds a cluster to the problem (method `AddCluster`, see Algorithm 4.1), the cluster is added to the generic solution and to the active set. Because it is not the output of a rewrite rule, the cluster can be identified in the generic solution as a problem cluster. The algorithm then searches for possible rewrite applications on that new cluster, i.e. rewrite rule applications where the cluster is used as input (method `SearchRewrites`, see Algorithm 4.3).

When a cluster is removed (method `RemoveCluster`, see Algorithm 4.2), it is removed from the generic solution and the active set. All dependent clusters are also removed from the generic solution. The dependent

clusters (function `DependentClusters`) are all clusters in the generic solution that are directly or indirectly determined by rewrite rules that use the given cluster as input. The algorithm must then determine a new set of active clusters. For this purpose, it determines which clusters were removed from the active set when this particular cluster was added (function `DeactivatedClusters`). The active set is restored by re-adding those clusters to the active set. It is possible that after restoring the active set, combinations of clusters can be rewritten (method `SearchRewrites`, see Algorithm 4.3).

Searching for possible rewrite rule applications (see Algorithm 4.3) can be done efficiently because we search only for rewrites on newly added clusters. Since each rewrite rule involves a small number of overlapping clusters (i.e. clusters sharing one or more point variables), we construct a subset of the set of active clusters consisting only of the newly added cluster and the clusters that overlap with it (function `OverlappingClusters`), and search in that subset for possible rewrite rule applications. The pattern matching algorithm thus searches only through a small number of clusters and

Algorithm 4.1: Adding a cluster

```
method AddCluster (G,A,c)
  G: generic solution
  A: active set
  c: cluster
begin
  G.add(c)
  A.add(c)
  SearchRewrites(G,A,c)
end
```

Algorithm 4.2: Removing a cluster

```
method RemoveCluster (G,A,c)
  G: generic solution
  A: active set
  c: cluster
begin
  A.remove(c)
  G.remove(c)     (* also removes rewrite rules on c *)
  for each x in DependentClusters(G,c)
    RemoveCluster(x)
  for each y in DeactivatedClusters(G,c)
    A.add(y)
    SearchRewrites(G,A,y)
end
```

variables.

The pattern matching algorithm used in our implementation is basically a subgraph matching algorithm that finds all subgraph isomorphisms [Ullmann, 1976]. The input pattern specified by a rewrite rule is converted into a graph (function `PatternGraph`). The subset of the active set in which we look for rewrite rule applications is also converted to a graph (function `ReferenceGraph`). For each subgraph isomorphism returned by the graph matching algorithm (function `SubgraphIsomorphisms`), we determine which point variable is assigned to which pattern variable, and from that the actual rewrite rule can be instantiated and added to the generic solution.

For each possible rewrite rule application found, the algorithm first checks whether the rewrite rule application is *progressive* (function `IsProgressive`), and only if it is, the algorithm adds it to the generic solution. A rewrite rule application is progressive if it either increases the number of distances and angles constrained by the active set, or reduces the number of active clusters. This ensures that the algorithm does not add redundant clusters to the system, except to remove overconstrained clusters.

Generally, when a rewrite rule is added to the generic solution, its output cluster becomes part of the set of active clusters, and one or more input clusters may be removed from the active set. A cluster is removed from the active set if it is redundant, i.e. if all distances and angles constrained by it

Algorithm 4.3: Searching for rewrite rule applications

```
method SearchRewrites(G,A,c)
  G: generic solution
  A: active set
  c: cluster
begin
  subset := c + OverlappingClusters(A,c)
  reference := ReferenceGraph(subset)
  for each rule in AllRewriteRules
    pattern := PatternGraph(rule)
    matches := SubgraphIsomorphisms(pattern,reference)
    for each match in matches
      rewrite := instantiate rule from match
      if IsProgressive(rewrite) then
        G.add(rewrite)     (* also adds output cluster *)
        A.add(rewrite.output)
        for each i in rewrite.inputs
          if IsRedundant(i) then
            A.remove(i)
        SearchRewrites(rewrite.output)
end
```

intersection	condition
$Rigid(A) \cap Rigid(B) = Rigid(A \cap B)$	$\lvert A \cap B \rvert > 1$
$Rigid(A) \cap Scalable(B) = Scalable(A \cap B)$	$\lvert A \cap B \rvert > 2$
$Rigid(A) \cap Radial(p_c, B) = Radial(p_c, A \cap B)$	$p_c \in A, \lvert A \cap B \rvert > 2$
$Scalable(A) \cap Scalable(B) = Scalable(A \cap B)$	$\lvert A \cap B \rvert > 2$
$Scalable(A) \cap Radial(p_c, B) = Radial(p_c, A \cap B)$	$p_c \in A, \lvert A \cap B \rvert > 2$
$Radial(p_c, A) \cap Radial(p_c, B) = Radial(p_c, A \cap B)$	$\lvert A \cap B \rvert > 2$

Table 4.1: Pairwise cluster intersections. If none of the cases listed here matches, then the intersection is empty, i.e. the intersection contains no distances or angles.

cluster	distances	angles
$Rigid([p_1, \ldots, p_n])$	$\binom{n}{2}$	$3\binom{n}{3}$
$Scalable([p_1, \ldots, p_n])$	0	$3\binom{n}{3}$
$Radial(p_c, [p_1, \ldots, p_n])$	0	$\binom{n}{2}$

Table 4.2: Number of distance and angles constrained by clusters.

are already constrained by the other clusters in the active set.

To determine whether a cluster is redundant (function `IsRedundant`), the algorithm needs to determine whether the set of distances and angles constrained by the cluster is a subset of the set of distances and angles constrained by the other clusters in the active set. Determining these sets explicitly is too expensive. Instead, we determine the number of distances and angles constrained by the cluster and the number of distances and angles constrained by each intersection of the cluster with any other overlapping cluster in the active set.

We define the intersection of two clusters as a cluster that constrains only those distances and angles that are constrained by both clusters. The intersection can be determined efficiently using the rules listed in Table 4.1. For example, given two clusters, $Rigid([p_1, p_2, p_3])$ and $Rigid([p_2, p_3, p_4])$, the intersection is determined by the first rule in the table as $Rigid([p_2, p_3])$. The table shows how the type of the intersection cluster is determined by the types of the original clusters. The set of point variables constrained by the intersection cluster is the set of point variables shared by the original clusters. The set of shared points must satisfy additional conditions to ensure that the intersection cluster is a valid cluster, e.g. a minimum number of shared points is needed and the centre point of two radial clusters must be the same.

The number of distances and angles constrained by a cluster can be

determined from Table 4.2. Note that the number of distances and angles constrained by a cluster is larger than the number of constraints typically needed for a well-constrained system. However, because the allowable combinations of values of the distances and angles constrained by a cluster, are determined by a configuration, these values are always consistent.

If the number of distances and angles constrained by a cluster is larger than the total number of distances and angles constrained by the intersections of the cluster with each overlapping cluster in the active set, then the cluster is not redundant. Otherwise, the number of distances and angles in the cluster is equal to the total number of distances and angles in the intersections (it cannot be smaller), and the cluster is redundant.

The generic solution of a problem can be used to determine its particular solutions, by evaluating the computation part of each rewrite rule in the generic solution, for each combination of its input clusters' configurations. This may also be done in an incremental way. When the set of configurations associated with a problem cluster is changed, the dependent rewrite rules can be determined from the generic solution, i.e. the rules which use this cluster as input cluster. Only these rewrite rules need to be re-evaluated.

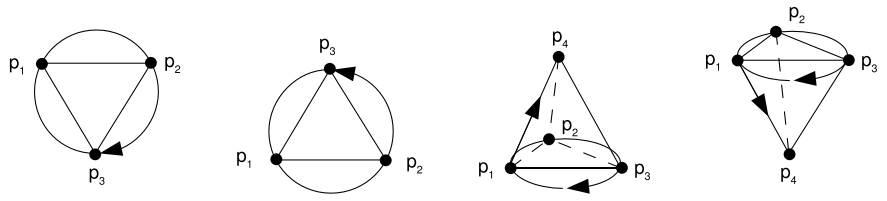
Note that the computation of particular solutions is repeated for every combination of input cluster configurations, and each rewrite rule may in turn generate several solutions for each such combination. The number of particular solutions may therefore be very high, and thus expensive to compute, whereas for most applications not all solutions are required or desirable. Thus, a solution selection mechanism is needed to reduce the number of solutions, and thus reduce the computation time.

4.5 Solution selection

In general, a geometric constraint problem can have several solutions. Solution selection is an important problem in geometric constraint solving, also known as the multiple-solution problem or the root identification problem.

In [Bettig and Shah, 2003] an overview is given of various solution selection schemes. The authors argue that declarative solution selectors are the most flexible and powerful approach. In this approach, the user specifies additional constraints (solution selectors), that narrow down the number of solutions until a single intended solution remains. Eleven basic selectors for various types of geometry have been identified, e.g. to specify that a point must be on a particular side of a curve.

Declarative solution selection is also supported by the solving algorithm presented in the previous sections. Selection constraints, like clusters, are defined on sets of point variables. These constraints, unlike clusters, can specify an arbitrary relation on those points that should be satisfied. The only requirement is that, given a configuration of those points, it should



(a) clockwise (b) counter-clockwise (c) left-handed (d) right-handed

Figure 4.7: Solution selection constraints

be possible to evaluate whether the constraint has been satisfied. However, these constraints can only be checked if the problem is well-constrained, and cannot be used to determine any DOF in the system.

In 2D, solutions can be selected based on whether a set of three points $\{p_1, p_2, p_3\}$ is oriented clockwise or counter-clockwise, as shown in Figure 4.7(a) and Figure 4.7(b), respectively. In 3D, the handedness of a set of points is a useful selection criterion. A set of points $\{p_1, p_2, p_3, p_4\}$ can be classified as left-handed or right-handed, as shown in Figure 4.7(c) and Figure 4.7(d), respectively.

In general, the orientation of a set of $n+1$ points in $\mathbb{R}^n$ can be determined by taking one point as a reference, and computing the determinant of the ordered set of offset vectors for the other points. We define the orientation of an ordered set of points $\{p_1, ..., p_{n+1}\}$ as:

$$Orientation(p_1, ..., p_{n+1}) = sign(Det[p_2 - p_1, ..., p_{n+1} - p_1])$$

The set of points is *positively oriented* if the determinant is positive, *negatively oriented* if the determinant is negative, and indeterminate if the determinant is zero. Note that the points may be represented as either the rows or the columns of the matrix of which the determinant is computed.

In a standard coordinate system, where the positive X-axis points to the right, the positive Y-axis points upwards, and the positive Z-axis points to the viewer, the $CounterClockWise$ and $RightHanded$ constraints are satisfied for an ordered set of points that is positively oriented, and the $Clockwise$ and $LeftHanded$ constraints are satisfied for an ordered set of points that is negatively oriented.

Because the number of solutions of a geometric constraint problem can be very large, and the computation of all the solutions expensive, solution selection should take place as soon as possible, i.e. as soon as enough information is available to evaluate the selection constraint. In general, a selection constraint can be evaluated as soon as a rigid cluster has been determined that involves a superset of the variables in the constraint. The specific 2D and 3D selection constraints discussed above can be evaluated

for the first determined scalable or rigid cluster on a superset of the variables in the constraint.

There are two problems with the declarative approach to solution selection. The first problem is that, in the worst case, all solutions of a geometric constraint problem have to be generated, before selection constraints can be evaluated. The number of solutions for geometric constraint problems is generally exponential to the number of geometric elements. In fact, the problem of finding all real solutions of a system of geometric constraints has been shown to be NP-complete [Fudos and Hoffmann, 1997]. Considering declarative solution selectors during the solving process is very likely to result in an NP-complete problem also [Bouma et al., 1995].

An alternative method to satisfy selection constraints is presented in [Joan-Arinyo et al., 2003]. A genetic algorithm selects a solution that satisfies a number of additional selection constraints, which determine on which side of a line a point should be. The drawbacks of this approach are that the genetic algorithm is not complete, i.e. it may not find the selected solution, and it is difficult to find optimal settings for the genetic algorithm.

The second problem with the declarative approach is that a large number of selection constraints generally needs to be specified, in order to determine a single solution. And this solution, as a function of the parameter of the problem, is often discontinuous. Discontinuous behaviour is undesirable for most parametrisations. In [van den Berg et al., 2003], requirements are identified for freeform feature classes and their instances. For users of the feature class, who generally have no knowledge of the constraints used for the parametrisation, discontinuities in the behaviour of the feature are unexpected and undesirable. Similar requirements hold for models of families of objects in general.

The most used alternative to the declarative approach is the prototype-based solution selection approach. A prototype is a configuration of all the geometric variables in the problem that 'looks like' the desired solution. Typically, the prototype is obtained from a sketch created by the user. Several solving algorithms use heuristic rules for solution selection based on the relative position and orientation of the geometry in a prototype, e.g. [Fudos and Hoffmann, 1997; Bouma et al., 1995; Podgorelec, 2002].

In [Essert-Villard et al., 2000] a formal framework is presented for sketch-based heuristics, based on homotopy theory. The S-homotopy relation is defined, which describes homotopic configurations that respect a system S of geometric constraints. By selecting a root for each triangular subproblem using local criteria, a solution is obtained that is S-homotopic to the sketch. In this way, the tree of possible solutions is pruned. Still, several branches may remain, resulting in different solutions. Some additional selection process is thus needed.

Some solving schemes allow the user to explore the tree of solutions

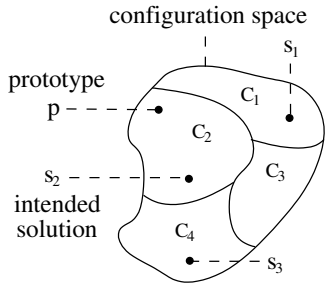


Figure 4.8: The resemblance relation partitions the configuration space into a number of equivalence classes C_i. The intended solution (s_2) is the solution that is in the same equivalence class (C_2) as the prototype (p).

when the search heuristics do not result in a single solution, e.g. [Bouma et al., 1995] and [Oung et al., 2001]. However, such a procedure can be cumbersome for large systems, or when systems are modified and need to be solved again, which is often the case in interactive modelling systems.

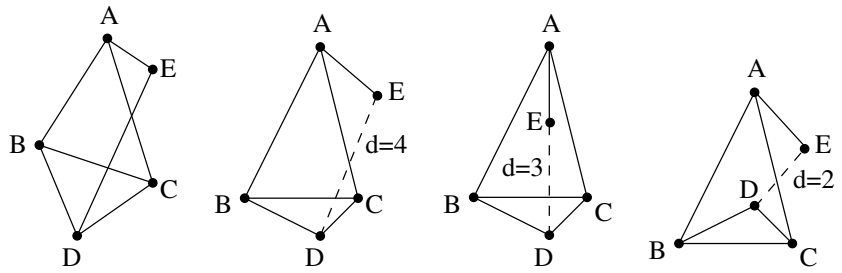
We present a new prototype-based solution selection mechanism that always determines at most one solution, the so-called *intended solution*. The intended solution satisfies the following properties:

- the intended solution is a continuous function of the parameters of the problem
- the intended solution uniquely resembles a given prototype.

The first property ensures that there is a predictable and intuitive relation between the intended solution and the parameters of the problem. The second property ensures that there is an intuitive relation between the intended solution and the prototype, so the user can control the selection process via the prototype. A unique resemblance between the intended solution and the prototype means that other solutions, found for the same parameter values, must not resemble the prototype, by some definition of resemblance. Thus, the intended solution is uniquely determined by the prototype.

Resemblance is defined by a resemblance relation. This relation is an equivalence relation, which partitions the configuration space into a number of equivalence classes. The intended solution is the solution in the same equivalence class as the prototype. This is illustrated in Figure 4.8.

For some combinations of the parameter and prototype, there may not be an intended solution, even though a real solution exists. Consider, for example, the 2D system in Figure 4.9. For any parameter value $d \geq 3$, the solution is a continuous function of d. If, however, we instantiate the problem with parameter $d = 2$, there is no solution in the same equivalence



(a) prototype (b) (c) (d)

Figure 4.9: A constraint problem with a prototype (a) and a distance parameter d. The intended solution exists only for $d \geq 3$ (b, c). For $d = 2$ (d) there is a solution, but there is no intended solution.

class, i.e. there is no solution for $d = 2$ that can be reached continuously from the previous solutions.

Note that when using the intended solution in the definition of a family of objects, i.e. in a DFOM, instead of all the solutions of the geometric constraint system, a smaller family is obtained. Using the intended solution implicitly imposes additional constraints, and consequently the DFOM has fewer realisations. The intended solution guarantees that if the geometric system is well-constrained, then at most one geometric solution is found, but several realisations may still be found due to the topological constraints in the model. Also note that, although the intended solution varies continuously with the parameters of geometric constraints, topological changes may still be observed in the realisations of the DFOM. Detecting such topological changes is discussed in Chapter 7.

The intended solution can be found by using the cluster rewriting algorithm presented in the previous sections. Basically, for each subproblem that is solved, selection constraints are generated, such that a single solution is selected for each subproblem. The selection constraints that are generated for a specific subproblem depend on the type of the subproblem, i.e. the specific rewrite rule, and the prototype. Since at most one solution is found for each subproblem, and no back-tracking search is needed, computing the intended solution in this way is inexpensive.

For example, consider the 2D system in Figure 4.9 again. This problem can be decomposed into three simple triangular problems: ABC, BCD and ADE. To solve each of these subproblems, the intersection of two circles is determined, as shown in Figure 4.10. The two solutions can be distinguished by the orientation of the points in the solution; either the points are counterclockwise oriented, i.e. ABC_1 or the points are clockwise oriented, i.e. ABC_2. If in the prototype the points ABC are oriented clockwise, then a selection constraint $Clockwise(A, B, C)$ is added to the system. If, on the other hand,

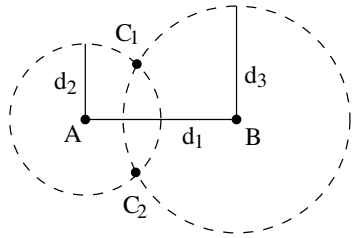


Figure 4.10: A simple triangular subproblem, where three distance constraints are given. The solution can be found by intersecting two circles.

the points in the prototype are oriented counter-clockwise, then a selection constraint $CounterClockwise(A, B, C)$ is added. If the orientation of the prototype points is clockwise nor counter-clockwise, i.e. the prototype points are on a line, then we must make an arbitrary choice, or warn the user. For the other subproblems in the example, the same procedure is followed.

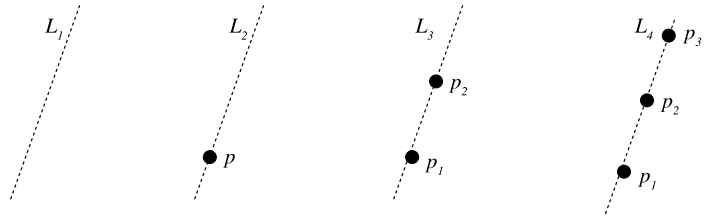
In general, for each type of subproblem, selection constraints can be generated that can distinguish between the possible solutions of the subproblem. The particular selection constraint that is satisfied by the prototype is added to the problem. For many 2D subproblems, the *Clockwise* and *Counterclockwise* selection constraints can be used, and for many 3D subproblems the $Lefthanded$ and *Righthanded* selection constraints. For some subproblems, inequality constraints on angles are used.

In Appendix B, we show for a somewhat simpler constraint solver, which only solves triangular and tetrahedral rigid subproblems, that the intended solution can be found using selection constraints. Basically, we formally define the properties that the resemblance relation must satisfy, in order to find the intended solution. For each type of subproblem, the selection constraints that are used define a resemblance relation. First we show, for several types of subproblems, that these resemblance relations satisfy the given properties. Then we show that by combining subproblems, a resemblance relation results for the whole problem, which also satisfies those properties. Thus, the solution found in this way is the intended solution. For the solver discussed in this chapter, which can also solve non-rigid clusters, we believe that a similar proof can be given.

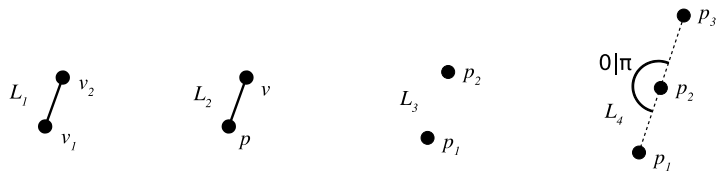
4.6 Constraints on 3D primitives

So far, in this chapter, we have only considered systems of distance and angle constraints on points, in particular, systems of clusters. In typical CAD models, and in the DFOM, however, constraints are imposed on other types of geometric primitives, e.g. lines, planes, spheres, cylinders, etc.

In this section, we present a mapping from constraint systems on primitives to a system of distance and angle constraints on points. Basically,



(a) lines with coincident points



(b) mapping

Figure 4.11: Mapping of lines ($L_1 - L_4$), with different numbers of coincident points

a primitive is represented by a small number of points, e.g. a line can be represented by two points. Constraints on primitives can then be expressed by distance and angle constraints on points. A system of distance and angle constraints on points, in turn, can be represented by a system of clusters, which can be solved with the solving algorithm presented in the previous sections.

The DOFs of a primitive cannot always be exactly represented by a set of point variables. For example, a line in 3D has 5 DOFs. It cannot be represented by a single point variable, which has 3 DOFs, or by two point variables, which together have 6 DOFs. However, we can correctly represent the DOFs of a *system* of primitives and constraints, at least in many cases where the system is well-constrained. In such cases, the solutions of a particular system of constraints on points can be used to construct the solutions of a particular system of constraints on primitives. The key idea is that the representation of a primitive depends on the number of points that are constrained to be coincident with it.

Figure 4.11 illustrates the mapping of lines with different numbers of coincident points. A line L_1, with no constraints imposed on it, is represented by two points v_1 and v_2, with an arbitrary distance constraint such that $\delta(v_1, v_2) \neq 0$. The represented line is the line through the points v_1 and v_2. By itself, this system is well-constrained. If a single point p is constrained coincident with a line L_2, by a constraint $Coincident(p, L_2)$, then line L_2 is represented by that point p and another point v. Again, this system is well-constrained. If two points are constrained coincident with a line L_3,

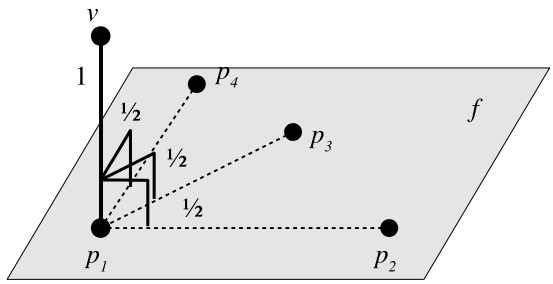


Figure 4.12: Mapping a plane f with four coincident points $p_1 - p_4$. An extra point variable v is introduced, representing the plane normal.

i.e. $Coincident(p_1, L_3)$ and $Coincident(p_2, L_3)$, then both points are used in the representation, i.e. line L_3 is represented by p_1 and p_2. If the system of points is well-constrained, then the line is also well-constrained. Note that the distance between the points is not constrained by the mapping, because if the system is well-constrained, then the distance between the points is already determined. If more than two points are constrained with a line L_4, i.e. $Coincident(p_i, L_4)$ for $1 <= i <= n$ with $n > 2$, then only the first two points, p_1 and p_2 are used in the representation. The other points are constrained to be coincident with the line, using an angle constraint that specifies that the angle between points is either 0 or π: $\measuredangle(p_1, p_2, p_i) = 0|\pi$ for $3 \leq i \leq n$. Such constraints with alternative values can be represented by clusters with several configurations.

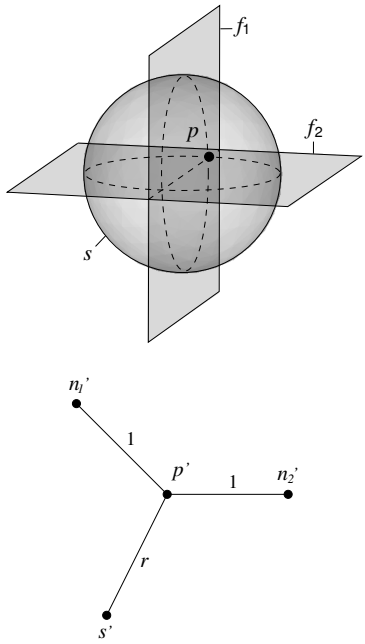
The mapping of planes is similar to the mapping of lines. A plane f, with no constraints imposed on it, is represented by two point variables, v_1 and v_2, and a distance constraint $\delta(v_1, v_2) \neq 0$. The represented plane is the plane through v_1 with normal $v_2 - v_1$. The first point that is constrained coincident with a plane, is used in the representation of the plane, i.e. if there is a constraint $Coincident(p, f)$ then the plane is represented by p and a point v, with a non-zero distance constraint between them. Any other points p_* constrained coincident with the plane are constrained in plane by a constraint $\measuredangle(v, p, p_*) = \frac{1}{2}\pi$.

The mapping of a system with four points $p_1 - p_4$ that are co-incident with a plane f, is illustrated in Figure 4.12. Effectively, a new point variable v is introduced, and some distance and angle constraints that force the points $p_1 - p_4$ to be in the plane.

For a sphere, we have to consider its centre point and its radius, since these are often constrained in applications. Fixed radius spheres are easily mapped. The centre of the sphere is represented by a point variable, and any point constrained coincident with the sphere is constrained with a distance equal to the given radius.

primitive	constraints	mapping ($\mathcal{R}$)
point p_1	-	$\mathcal{R}(p_1) = v_1$
point p_2	$Coincident(p_1, p_2)$	$\mathcal{R}(p_2) = \mathcal{R}(p_1)$
line l_1	-	$\mathcal{R}(l_1) = \{v_1, v_2\}$ $\delta(v_1, v_2) = 1$
line l_2	$Coincident(l_2, p)$	$\mathcal{R}(l_2) = \{\mathcal{R}(p), v\}$ $\delta(\mathcal{R}(p), v) = 1$
line l_3	$Coincident(l_3, p_1)$ $Coincident(l_3, p_2)$	$\mathcal{R}(l_3) = \{\mathcal{R}(p_1), \mathcal{R}(p_2)\}$
line l_4	$Coincident(l_4, p_1)$ $Coincident(l_4, p_2)$ $Coincident(l_4, p_*)$	$\mathcal{R}(l_4) = \{\mathcal{R}(p_1), \mathcal{R}(p_2)\}$ $\measuredangle(\mathcal{R}(p_1), \mathcal{R}(p_2), \mathcal{R}(p_*)) = 0 \vert \pi$
plane f_1	-	$\mathcal{R}(f_1) = \{v_1, v_2\}$ $\delta(v_1, v_2) = 1$
plane f_2	$Coincident(f_2, p)$	$\mathcal{R}(f_2) = \{\mathcal{R}(p), v\}$ $\delta(\mathcal{R}(p), v) = 1$
plane f_3	$Coincident(f_3, p_1)$ $Coincident(f_3, p_*)$	$\mathcal{R}(f_3) = \{\mathcal{R}(p_1), v\}$ $\delta(\mathcal{R}(p_1), v) = 1$ $\measuredangle(v, \mathcal{R}(p_1), \mathcal{R}(p_*)) = \frac{1}{2}\pi$
sphere s_1	$Radius(s_1) = r$ $Coincident(s_1, p_*)$	$\mathcal{R}(s_1) = v$ $\delta(v, \mathcal{R}(p_*)) = r$
sphere s_2	$Radius(s_2) = r$ $Center(s_2, p_1)$ $Coincident(s_2, p_*)$	$\mathcal{R}(s_2) = \mathcal{R}(p_1)$ $\delta(\mathcal{R}(p_1), \mathcal{R}(p_*)) = r$
cylinder c_1	$Radius(c_1) = r$ $Coincident(c_1, p_*)$	$\mathcal{R}(c_1) = \{v_1, v_2\}$ $\delta(v_1, v_2) = 1.0$ $\delta(v_*, \mathcal{R}(p_*)) = r$ $\measuredangle(v_1, v_*, \mathcal{R}(p_*)) = \frac{1}{2}\pi$
cylinder c_2	$Radius(c_2) = r$ $Axis(c_2, l)$ $Coincident(c_2, p_*)$	$\mathcal{R}(c_2) = \mathcal{R}(l)$ $Coincident(v*, l)$ $\delta(v_*, \mathcal{R}(p_*)) = r$

Table 4.3: Mapping of primitives with different incidence constraints. Points are represented by p_*, lines by l_*, planes by f_*, spheres by s_* and cylinders by c_*, where $*$ can be any integer.



point p
plane f_1
plane f_2
sphere s, radius r
p on s
p on f_1
p on f_2

$\mathcal{R}(p) = p'$
$\mathcal{R}(f_1) = \{p', n_1'\}$
$\delta(p', n_1') = 1$
$\mathcal{R}(f_2) = \{p', n_2'\}$
$\delta(p', n_2') = 1$
$\mathcal{R}(s) = s'$
$\delta(p', s') = r$

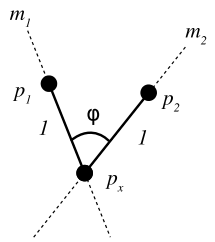
Figure 4.13: Mapping plane and sphere primitives with coincidences to constraints on points

Variable radius spheres cannot be easily represented, because all points coincident with the sphere must have an equal, unknown distance from its centre. Such equalities are not supported by our solver, because distance constraints must have a fixed parameter value. In some cases, however, variable radius circles, spheres or cylinders can be supported, by using propagation to solve the equality constraint. If the radius can be determined by first solving other constraints in the system, then this value can simply be propagated to those distance constraints that should be equal to the radius. However, these cases will not be further considered here.

Table 4.3 shows the different representations for primitives depending on the number of points constrained coincident with it. The function $\mathcal{R}$ maps a primitive to a set of point variables and constraints on those variables, depending on the constraints imposed on the primitive.

Figure 4.13 shows an example of how a set of primitives with coincidence constraints can be mapped to a system of constraints on points.

Constraints on primitives are mapped to a set of points, a set of coincidence constraints between the primitives and those points, and a set of distance and angle constraints between those points. The representation of the primitive is determined by the points that are constrained coincident with it.



$Coincident(p_1, m_1)$
$Coincident(p_2, m_2)$
$Coincident(p_x, m_1)$
$Coincident(p_x, m_2)$
$\delta(p_x, p_1) = 1.0$
$\delta(p_x, p_2) = 1.0$
$\angle(p_1, p_x, p_2) = \phi$

Figure 4.14: Mapping of an angle between lines, $\angle(m_1, m_2) = \phi$.

For example, an angle between two lines $\angle(m_1, m_2) = \phi$, where $0 < \phi < \pi$, is mapped to three new points, p_1, p_2 and p_x, and a number of constraints, as shown in Figure 4.14. The constraint may or may not affect the representations of the lines, depending on the constraints already present in the system. If no other constraints are imposed on the lines, m_1 is represented by p_1 and p_x, and m_2 is represented by p_2 and p_x. The angle between two planes can be constrained by constraining the angle between the normals of those planes, using the same construction.

In general, many useful constraints on primitives can be mapped in this way, and the resulting constraint systems can be solved using the cluster-based approach presented in this chapter. From the solutions of the latter systems, the geometry of the primitives can be completely determined.

In CAD systems, after geometric constraints have been solved, a geometric representation for the model is constructed that also contains topological information, e.g. a B-rep. In the case of the DFOM, this is a cellular model (CM). The CM only encodes the topological relations between the carriers in a realisation, which are completely determined by solving the geometric constraints in the model. The complete topology of a realisation is determined by the topology of the CM and the material values associated with the cells in the CM. These material values are determined in a subsequent step by solving the system of topological constraints in the model. A method for this is presented in the next chapter.

Chapter 5

Topological constraint solving

Solving topological constraints is a new and relatively unexplored subject. The closest related research area is topology optimisation. However, in that area, topology is mostly defined procedurally, e.g. by a shape grammar [Stiny and Gips, 1972], or defined implicitly, e.g. by a level-set function [Bendsoe, 1989], but not declaratively, with constraints, as in the DFOM.

In this chapter, we present a novel approach to solving topological constraints, which can be used to determine realisations for models with declarative topology, such as the DFOM. We assume here that a cellular model (CM) is available and that constraints are imposed on volumes and surfaces that can be represented by combinations of cells in the CM. The topological solver determines for each cell in a CM whether it contains material or not, such that these constraints are satisfied. To do this, each cell in a CM is represented by a Boolean variable, and topological constraints are mapped to a set of Boolean constraints (see Section 5.1). The system of Boolean constraints is then solved by a Boolean constraint solver (see Section 5.2).

Parts of this chapter have already been published in [van der Meiden and Bronsvoort, 2006b; van der Meiden and Bronsvoort, 2007a].

5.1 Mapping topological constraints

In this chapter, we consider topological constraints imposed on any volume or surface that can be represented by a combination of cells in a given CM. Topological constraints in a DFOM are imposed on constructs, from which a CM can be generated, thus we can use the method presented here to solve the topological constraints in a DFOM.

A solution to a topological constraint problem is a CM with a set of assignments, specifying for each cell in the CM whether the cell contains material. Thus a solution is equivalent to a realisation of a DFOM (see

Section 3.4).

All cells in the CM are mapped to Boolean variables. If a variable is TRUE, then the corresponding cell contains material; if it is FALSE, the corresponding cell does not contain material.

Topological constraints are mapped to Boolean constraints. A Boolean constraint is a predicate on a set of Boolean variables, which may be expressed by a Boolean function $p(v_1, v_2, \ldots, v_k)$, where $v_1 \ldots v_k$ are Boolean variables. If p evaluates to TRUE, then the constraint is satisfied. If p evaluates to FALSE, then the constraint is not satisfied.

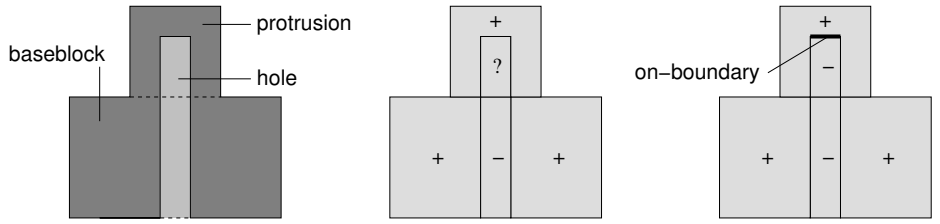
In this section, we present the mappings for the topological constraints in the DFOM, i.e. the nature constraints, boundary constraints and interaction constraints listed in Table 3.2 on page 41.

A constraint NATURE(M, ADDITIVE) or NATURE(M, SUBTRACTIVE) specifies that the nature of a volume, represented by a construct M that is part of a feature, is additive or subtractive, respectively. Basically, an additive volume determines that cells overlapping with it contain material, unless some subtractive volume is dependent on it and overlaps with those cells. Vice versa, a subtractive volume determines that cells overlapping with it do not contain material, unless some additive volume is dependent on it and overlaps with those cells.

In the SFM, dependency was defined for features only. We use a very similar definition to define dependency for volumes. A feature F_1 is directly dependent on a feature F_2 if there is a constraint in F_1 imposed on a variable in F_2. Thus, dependencies are created via interface constraints. In this way, we can construct a dependency graph of direct dependencies, and we can infer indirect dependencies from such a graph. A volume M_1 that is defined in a feature F_1, is dependent on a volume M_2 in a feature F_2, if F_1 is dependent on F_2.

Nature constraints are mapped to Boolean constraints as follows. For each cell c in the CM, we determine the set of volumes $\mathcal{M}$ that overlap with it. The dependencies between these volumes are analysed to find the set $\mathcal{D}$ of *dominant volumes*, which is the set of volumes on which no other volumes in $\mathcal{M}$ are dependent. The nature constraints of these volumes are dominant over the nature constraints of other volumes. Let $\mathcal{N}$ be the set of nature constraint values (ADDITIVE or SUBTRACTIVE) associated with the volumes in $\mathcal{D}$. If $\mathcal{N}$ contains only ADDITIVE, then the cell should contain material. This is mapped to a constraint $c =$ TRUE. If $\mathcal{N}$ contains only SUBTRACTIVE, then the cell should not contain material. This is mapped to a constraint $c =$ FALSE. If $\mathcal{N}$ contains no value, or both values, then no constraint is imposed on c.

Figure 5.1(a) shows a 2D model that is similar to a slice through the 3D model in Figure 2.2 on page 10, which illustrates the feature ordering problem. In this model, a blind hole feature cuts through a base block, into



(a) DFOM (b) CM (c) CM

Figure 5.1: A 2D model corresponding to Figure 2.2 on page 10. In (b) cell values have been determined by nature constraints, and in (c) cell values have been determined by nature and boundary constraints.

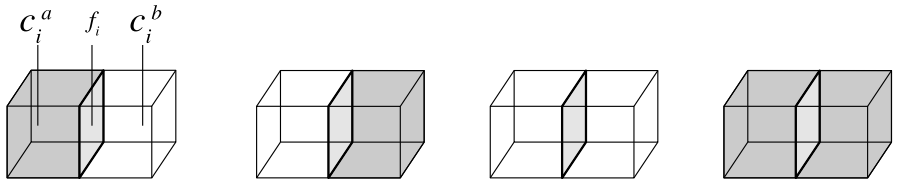
a protrusion feature. The blind hole has subtractive nature, whereas the block and the protrusion feature have additive nature.

Figure 5.1(b) shows the CM of the model, where the cell values are been determined by mapping nature constraints. In this figure, cells marked with '+' contain material, cells marked with '-' do not contain material, and the nature of one cell, marked with '?', has not been decided. The cell with the '?' corresponds to the intersection of the hole and the protrusion, which are independent of each other and have a different nature constraint. Therefore, the value of the cell can only be determined by solving other topological constraints. In the case of a blind hole, the bottom face of the hole should be on the boundary of the model. This can be expressed by a CompletelyOnBoundary constraint (see below), and satisfying this constraint results in the CM shown in Figure 5.1(c).

In this example, and also in general, most cell values are determined by nature constraints. Usually, only a relatively small number of cells is involved in feature interactions, such that their material value cannot be determined by nature constraints. Only for those cells, the other topological constraints in the model have to be solved.

Boundary constraints are imposed on surfaces, represented by constructs in a DFOM. Surfaces are constructs that are constrained ON one carrier, which determines the geometry of the surface, and IN or OUT several other carriers, which determine the extent of the surface. In the CM, faces overlap with the surfaces on which boundary constraints are be imposed. However, because the topological solver determines only which cells in the CM contain material, boundary constraints must be expressed in terms of the material values of cells.

A CompletelyOnBoundary(S) constraint specifies that the surface S must be completely on the boundary of the model. This constraint is mapped as follows. We first determine the set of cell faces $f_1, ..., f_k$ in the CM that are coincident with S. Each cell face f_i has two adjacent cells, c_i^a



(a) on-boundary (b) on-boundary (c) not on-boundary (d) not on-boundary

Figure 5.2: Representing on-boundary and not on-boundary cell faces in terms of cell values.

and c_i^b. A cell face f_i is on the boundary of the model if and only if exactly one adjacent cell contains material. This is illustrated in Figure 5.2.

To satisfy the constraint, all cell faces must be on the boundary. Thus, the constraint is expressed as follows:

$$\textsc{CompletelyOnBoundary}(S) = \bigwedge_{i=1...k} ((c_i^a \vee c_i^b) \wedge \neg(c_i^a \wedge c_i^b))$$

Note that this Boolean expression concerns the material values of the cells, not the geometry of the cells.

The mappings of the other boundary constraints, i.e. the CompletelyNotOnBoundary, PartiallyOnBoundary and PartiallyNotOnBoundary constraints, are similar to the mapping above:

$$\textsc{CompletelyNotOnBoundary}(S) = \bigwedge_{i=1...k} \neg((c_i^a \vee c_i^b) \wedge \neg(c_i^a \wedge c_i^b))$$

$$\textsc{PartiallyOnBoundary}(S) = \bigvee_{i=1...k} ((c_i^a \vee c_i^b) \wedge \neg(c_i^a \wedge c_i^b))$$

$$\textsc{PartiallyNotOnBoundary}(S) = \bigvee_{i=1...k} \neg((c_i^a \vee c_i^b) \wedge \neg(c_i^a \wedge c_i^b))$$

Interaction constraints are imposed on volumes. In some cases, the mapping to Boolean constraints involves only constraints on the cells that overlap with the volume. This is the case for the obstruction interaction and absorption interaction constraints. The splitting, disconnection and closure interaction constraints require that the connectedness of the whole CM is considered.

An obstruction interaction is defined as causing partial obstruction of the volume of a subtractive volume. An obstruction interaction constraint disallows this interaction. In other words: to satisfy an obstruction interaction constraint on a volume M, all cells $c_1, ...c_k$ that have M in their owner list, may not contain material:

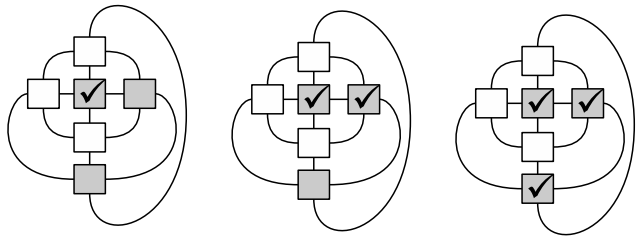


Figure 5.3: In each of the figures above, variables are represented by squares and adjacency information is represented by edges. Variables that are TRUE are indicated by gray squares. From left to right, marks are propagated to adjacent variables that are also TRUE.

$$\text{NoObstruction}(M) = \bigwedge_{i=1...k} (\neg c_i)$$

An absorption interaction is defined as causing a volume to cease completely its contribution to the model. In that case, the volume does not contribute to the boundary of the model. An absorption interaction constraint disallows this interaction. Thus, an absorption interaction constraint on a volume M is satisfied if any of the bounding surfaces $S_1...S_k$ of M is at least partially on the boundary of the model:

$$\text{NoAbsorption}(M) = \bigvee_{i=1...k} \text{PartiallyOnBoundary}(S_i)$$

The splitting, disconnection and closure interaction constraints require that the connectedness of the CM can be expressed as a Boolean constraint. We formulate an expression $\text{TrueConnected}(G)$ of a graph $G = (V, E)$, where $V = \{v_1, ..., v_n\}$ is a set of Boolean variables and E is a set of edges, representing adjacency of variables. This expression evaluates to TRUE if and only if all variables with the value TRUE are connected via edges of the graph.

The TRUECONNECTED expression is based on a simple propagation algorithm: first, a single variable that is TRUE is marked, and the mark is propagated to the adjacent variables that are TRUE. When no more marks can be propagated, either all TRUE variables have been marked, in which case the system is TRUECONNECTED (see Figure 5.3), or some TRUE variables have not been marked, in which case the system is not TRUECONNECTED.

Representing the propagation of marks in an expression requires that we model each mark in each iteration as a variable. For each $v_i \in V$, we define mark-variables $m_{i,t}$, where t is the iteration number. If and only if $m_{i,t}$ is TRUE, then variable v_i has been marked for iteration t. We know

that the maximum number of iterations for spreading marks is equal to the number of variables in the adjacency graph, thus we need $|V|^2$ variables to represent the state of the propagation algorithm. For each node, and for each iteration, we must specify an expression representing the propagation of marks from the previous iteration:

$$m_{i,t} = v_i \wedge \bigvee \{m_{j,t-1} | v_j \in Star(v_i)\}$$

Here, $Star(v_i)$ is the set of variables adjacent to v_i, i.e. all variables v_j for which there is an edge (v_i, v_j) in the adjacency graph G, plus v_i itself.

To begin propagation, a single node that evaluates to TRUE must be marked. However, when the expression is constructed, we cannot know which variables will evaluate to TRUE. We therefore order the variables and add constraints such that only the first node in the ordering that evaluates to TRUE is marked:

$$m_{i,0} = \bigwedge \{\neg v_j | j < i\}$$

The TRUECONNECTED constraint is formulated as an expression that tests whether all variables v_i that are TRUE have been marked after the last iteration:

$$\text{TRUECONNECTED}(G) = \bigwedge_{i=1...n} \{v_i = m_{i,T}\} \qquad T = |V| - 1$$

Because this formulation of connectedness requires quadratic memory space, it should not be used to formulate connectedness for the whole model. It is currently only used for parts of the model determined by volumes with a splitting interaction or disconnection interaction constraint.

Splitting interaction is defined as an interaction that causes the boundary of a volume to be split into two or more disconnected subsets. A splitting interaction constraint disallows this interaction. To satisfy the constraint for a volume M, all cell faces that correspond to a face of M and that are on the boundary of the model, must be connected. For each cell face $f_1...f_k$ of each cell that overlaps with M, we define a variable $v_i = \text{ONBOUNDARY}(f_i)$, $i = 1...k$. A graph $G_{ob} = (V, E)$ is constructed, where $V = \{v_1, ..., v_k\}$ and the edges E represent the adjacency between the corresponding cell faces. Then the spitting interaction constraint is simply:

$$\text{NOSPLITTING}(M) = \text{TRUECONNECTED}(G_{ob})$$

Disconnection interaction is defined as an interaction that causes the volume of an additive volume, or part of it, to become disconnected from the model. A disconnection interaction constraint disallows this interaction. In other words: all cells of a volume M that contain material, must be connected to the rest of the model. We construct a graph G_{adj} containing

variables V_M for all cells that overlap with M, plus variables V_{adj} for the cells that are adjacent to M. There is an edge in the graph for each pair of adjacent cells, i.e. cells that share a cell face. The rest of the model is represented by a single variable v_{model}, and there is an edge in the graph between v_{model} and each $v \in V_{adj}$. To test for disconnection interaction, we impose a TrueConnected constraint on this graph:

$$\text{NoDisconnect}(M) = \text{TrueConnected}(G_{adj})$$

A closure interaction is defined as causing some subtractive volume to become (part of) a closed void in the model. Combinations of subtractive volumes may form closed voids, whereas the volumes separately are not closed voids. A closed void is a set of empty cells which is surrounded by non-empty cells. A NoClosure constraint is satisfied if every empty cell, overlapping with the constrained volume, is connected to the space outside the model. Since we do not know before solving which cells contain material, the NoClosure constraint potentially affects all cells of the CM. It is possible to map this constraint to a Boolean expression, using the TrueConnected expression, but the size of the expression is quadratic with respect to the size of the model, and it is thus expensive to do this mapping. Therefore, we do not map this constraint, but simply verify it for each solution.

In general, if mapping a constraint is very expensive, it is probably cheaper to not map the constraint but to verify it for each solution, and selecting those solutions for which it is satisfied. As a rule, constraints involving global properties, e.g. connectedness of the whole model, are not mapped. The number of realisations for which such constraints must be verified is expected to be small, i.e. we expect most cells in the CM to be determined by solving other topological constraints.

5.2 Boolean constraint solving

The Boolean constraint problem to be solved is the following: given a set of n Boolean variables $V = \{v_1, v_2, \ldots, v_n\}$ and a set of m Boolean constraints $C = \{c_1, c_2, \ldots, c_m\}$, find an assignment $v_i := x_i$, where $x_i \in \{True, False\}$ for every $v_i \in V$, such that every constraint $c_j \in C$ is satisfied. This problem is known as the *Boolean satisfiability problem*, or SAT problem, which is an NP-hard problem [Papadimitriou, 1995]. The SAT problem has been well studied, and search algorithms exist that can find solutions efficiently for many instances. State-of-the-art solvers learn from earlier 'mistakes' and avoid search paths that are unlikely to lead to a solution.

The solver that we have used in our implementation is the MINISAT solver [Een and Sörensson, 2004], which is based on a solving technique called conflict-driven learning [Silva and Sakalla, 1996; Zhang et al., 2001].

The MINISAT solver, like most SAT solvers, is specialised to problems formulated in *conjunctive normal form* (CNF), represented by clauses. A clause is a set of literals, where a literal is a variable $v_i \in V$ or the negation of a variable, $\bar{v}_i$. A clause $(l_1, l_2, \ldots, l_j)$ is interpreted as $l_1 \vee l_2 \vee \ldots \vee l_j$. A set of clauses $\{c_1, c_2, \ldots, c_k\}$ is interpreted as $c_1 \wedge c_2 \wedge \ldots \wedge c_k$.

Some topological constraints can perhaps be formulated in CNF manually, but this is cumbersome for all but the most trivial constraints. It is, however, feasible to formulate topological constraints as expressions using the common Boolean operators, AND, OR, and NOT, as done in the previous section. Such expressions are represented in our system as trees. Each node of the tree represents either an operator or a variable. Operator nodes point to other nodes, which are used as inputs. Since creating small expressions for generic constraints can be difficult, large expressions should be expected. We use some simple reduction strategies to reduce the size of expressions: if an expression contains several sub-expressions that are lexically equivalent, they are represented by a single node that is referenced multiple times. Also, during construction of an expression, some local simplification rules are applied to reduce the number of nodes, e.g. removing double negations and evaluating expressions containing constants.

The Boolean expressions are converted to CNF clauses, using a method called *Tseitin encoding* [Tseitin, 1968]. This method replaces each subexpression with a new variable. The meaning of the operators between subexpressions, is expressed by CNF clauses on the new variables. For example, the expression $a \vee (b \wedge c)$ is expanded as follows. The subexpression $b \wedge c$ is substituted by a new variable x. Now the expression becomes $a \vee x$, which is already a CNF expression. We also represent the expression $x = b \wedge c$ in CNF, which becomes:

$$(b \vee c \vee \bar{x}) \wedge (b \vee \bar{c} \vee \bar{x}) \wedge (\bar{b} \vee c \vee \bar{x}) \wedge (\bar{b} \vee \bar{c} \vee x)$$

The complete expression thus corresponds to the following set of clauses:

$$(a, x), (b, c, \bar{x}), (b, \bar{c}, \bar{x}), (\bar{b}, c, \bar{x}), (\bar{b}, \bar{c}, x)$$

The system of CNF clauses is solved with the MINISAT solver. The standard implementation of this solver finds only one solution for a given system (enough to determine satisfiability). For applications such as the DFOM, we need to find all the solutions. Therefore, we have implemented an algorithm that can be used to finds all solutions, by systematically adding extra constraints to the system, forcing the solver to find different solution.

Our algorithm for finding all solutions (Algorithm 5.1) assumes that a solution has already been found by the MINISAT solver, and is given as an argument to the algorithm, along with the solver state, and a third argument representing the number of fixed variables (which should be zero initially

and will be incremented recursively). The algorithm attempts to fix the first free variable in the system to the complementary value of the value in the given solution. It does this by adding a unit-clause to the system, i.e. a clause with one literal. If this results in an unsatisfiable problem, then instead the variable is fixed to the value found in the solution. As long as no new solution is found, the next free variable in the system is fixed, until no free variables remain. If a complementary unit-clause was successfully added, then a new solution has been found, and more solutions may be found by recursively applying the algorithm with the incremented number of fixed variables. If no complementary unit-clause could be added to any variable, then the algorithm quits, and backtracks if it was called recursively. For correct functioning of the recursive algorithm, all unit-clauses added by the algorithm must be removed before returning the solutions. Thus, potentially all possible combinations of additional unit-clauses are tried, but unsatisfiable combinations are pruned early, because the MINISAT solver determines which additional constraints result in conflict.

To gain insight in the feasibility of solving topological constraints, we have experimented with solving several models that generate a large numbers of cells and topological constraints.

The first model, shown in 2D in Figure 5.4, contains a base block with a slot. On the base block, n protrusions are stacked. To the slot, m blind

Algorithm 5.1: Determine all subsequent solutions for a CNF problem (using MINISAT solver)

```
function NextSolutions(solution,solver,nfixed[=0])
  solution:  list of Booleans
  solver:    MINISAT solver instance
  nfixed:    number of fixed variables
begin
  newsolutions := empty list
  i := nfixed
  while i < length(solver.variables)
    solver.AddClause(solver.variables[i]=not(solution[i]))
    next := solver.Solve()
    if next != unsatisfiable then
      newsolutions.add(next + NextSolutions(next, solver, i+1))
    solver.RemoveClause(solver.variables[i]=not(solution[i]))
    solver.AddClause(solver.variables[i]=(solution[i])
    i := i + 1
  while i > nfixed
    solver.RemoveClause(solver.variables[i]=previous[i])
    i := i - 1
  return newsolutions
end
```

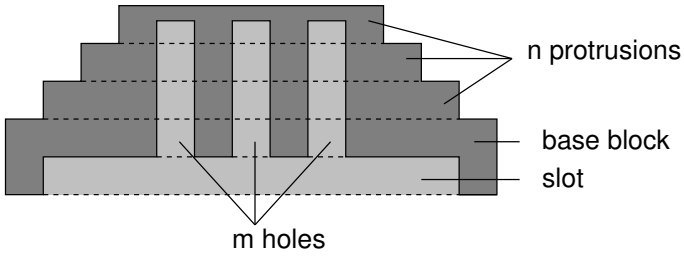


Figure 5.4: 2D model with $n = 3$ protrusion features and $m = 3$ hole features.

holes are attached, such that the holes intersect with the protrusions. This model has been solved for different values of m and n, to determine the solving times for increasing numbers of feature interactions. The hole features and the protrusion features are all dependent on the base block, but are independent of each other. Thus, the values of the intersection cells of the holes and the protrusions cannot be determined by feature dependency analysis. However, each hole has a topological constraint that it may not be split. The model has exactly one solution for any given number of holes and protrusions.

Table 5.1 shows, for different values of $m + n$, the number of cells in the CM, the number of variables and clauses in the constraint system, and the solving time in milliseconds. The number of cells in the model is roughly quadratic to the number of features in the model, because the model is designed to have a large number of feature interactions. The number of clauses is linear with respect to the number of variables and the number of cells in the model. Each solving time in the table is the sampled mean over 50 experiments. The standard deviation for the solving times is approximately 10% of the mean.

Figure 5.5 shows a plot of the solving times against the number of cells in the model. The latter quantity is representative for the complexity of the model, because it depends on the number of feature interactions. The plot suggests that solving time grows somewhat faster than linear with respect to the number of cells in the model, but solving times are fairly low, even for problems with a large number of variables and clauses.

We have also run the experiment on a model with more complex feature intersections, shown in 3D in Figure 5.6. This model, like the model in Figure 5.4, has a unique solution for each tested instance, but has a larger number of topological constraints per cell. A plot of the solving times against the number of cells for this model is shown in Figure 5.7. This plot again shows a somewhat more than linear increase of the solving time with the number of cells, but as before, solving times are fairly low given the size of the problems.

$m+n$	cells	variables	clauses	time(ms)
10+10	259	672	2252	3.9
16+16	601	1446	4616	8.1
21+21	996	2311	7191	11.9
25+25	1384	3147	9647	17.4
28+28	1717	3858	1172	23.1
31+31	2086	4641	13991	28.7
34+34	2491	5496	16460	39.4
37+37	2932	6423	19127	41.3
39+39	3246	7081	21015	50.7
41+41	3576	7771	22991	52.6
44+44	4101	8866	26120	64.1
46+46	4471	9636	28316	82.1
48+48	4857	10438	30600	77.5
49+49	5056	10851	31775	80.6
51+51	5466	11701	34191	96.4
53+53	5892	12583	36695	91.0
55+55	6334	13497	39287	122.0
56+56	6561	13966	40616	117.3
58+58	7027	14928	43340	151.4
59+59	7266	15421	44735	144.0

Table 5.1: Solving statistics for the model in Figure 5.4.

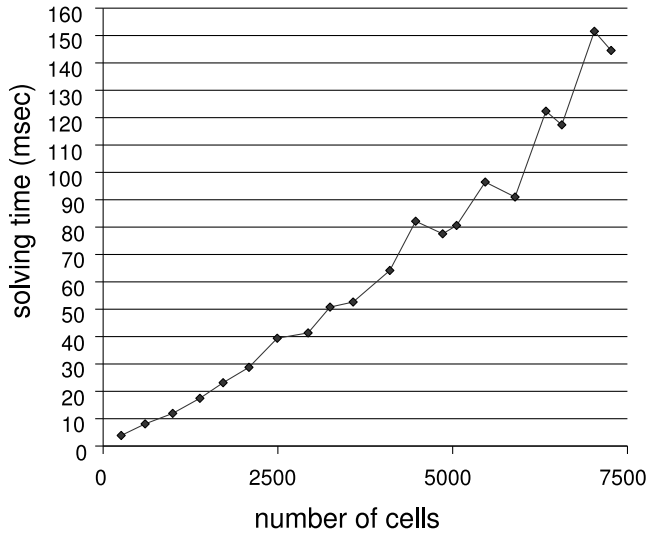


Figure 5.5: Plot of solving times against the number of cells for the model in Figure 5.4.

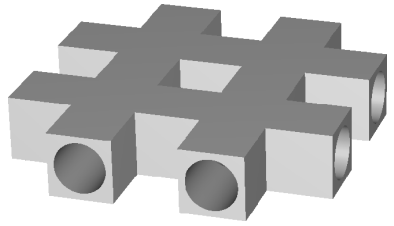


Figure 5.6: 3D model with $n + n$ intersecting blocks and $n + n$ through holes, shown for $n = 2$.

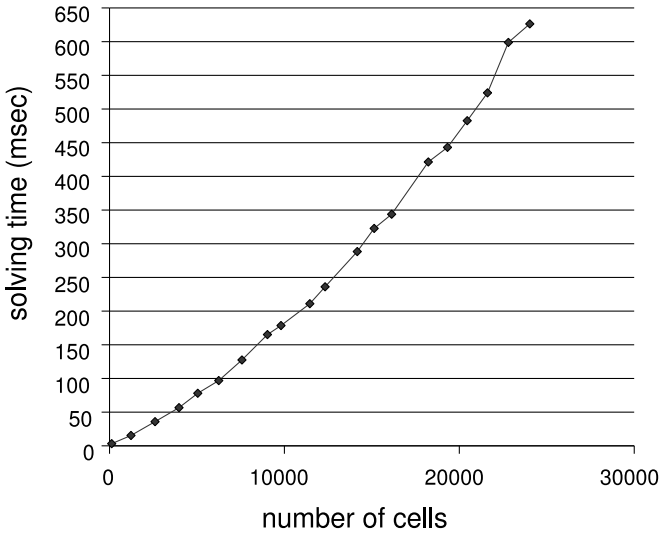


Figure 5.7: Plot of solving times against the number of cells for the model in Figure 5.6.

From the fast solving times in the benchmarks presented here, it would seem that the SAT problems generated from the example models are easy to solve. The number of clauses per variable is often cited in SAT literature as an indication for the difficulty of problems. For randomly created k-SAT problems (i.e. problems with k variables per clause), lower and upper bounds have been found for the number of clauses per variable, between which problems are generally considered hard to solve. Below the lower bound, problems have a high probability of being satisfiable, and above the upper bound, problems have a high probability of being unsatisfiable. Such problems are generally easy to solve with modern SAT solvers. Our topological solver maps constraints to 3-SAT clauses, and the tightest known lower and upper bounds for hard 3-SAT problems are currently 3.42 and 4.51, respectively [Achlioptas and Peres, 2003]. For the models in Figure 5.4 and Figure 5.6, the clauses per variable ratio is approximately 3.0, for all generated 3-SAT problem instances, which is below the lower bound for hard problems, and thus it is not surprising that these problems are easy to solve.

It should be noted that, in the models discussed above, the number of cells grows quadratically with the number of features, and the number of clauses quickly gets very large. This is because the example models were intentionally designed to maximise the number of feature interactions. In realistic models, the number of feature interactions will likely be much smaller, resulting in fewer cells and fewer clauses, and thus even better solving times can be expected.

We have not systematically tested the performance of the topological constraint solver for models with no realisations and for models with several realisations, however, tests with small models with these properties have not shown any performance problems.

For larger models with no realisations, no performance problems are expected, because the types of constraints used in our models so far have resulted in easy SAT problems, but this still needs to be verified experimentally.

For models with more than one solution, we expect performance to depend strongly on the number of solutions generated. For many applications, not all solutions have to be generated, e.g. for deciding DFOM membership, it is enough to know whether there are zero, one or more than one solutions. We expect that the difficulty of finding a first solution for problems with more than one solution, i.e. solving satisfiability for such problems, will in fact be easier than for problems with a unique solution, because the first problems typically have fewer clauses per variable, and are thus more likely to fall below the threshold for hard SAT problems.

The real cost of finding all solutions of a topological constraint problem, with respect to the size of the problem, is difficult to determine. At least in

the examples we have tried, which we believe represent typical models, the cost of finding a unique solution does not seem to grow exponentially with the number of cells in the problem, and this is confirmed by the analysis of the number of clauses per variable in the corresponding SAT problems. Let us assume for simplicity that for different models with the same number of cells, the algorithmic complexity is of the same order. We write $O(i(n))$ for the algorithmic complexity of finding the initial solution of a system with n cells, and $O(p(n))$ for the algorithmic complexity of incremental solving, i.e. finding a new solution, or a conflict, after adding or removing a unit-clause. We expect $O(p(n))$ to be smaller than $O(i(n))$, because of the incremental nature of the MINISAT solver, and because the solver remembers conflicts found while searching for earlier solutions.

Finding one additional solution for a problem, involves trying to add unit clauses to the system without causing the system to become unsatisfiable. In the worst case, we need to try and fix every variable and incrementally solve the system, before a new solution is found. However, we can optimise the algorithm by only fixing variables corresponding to cells in the model, since only these are relevant for constructing realisations. Thus, finding one additional solution takes $O(np(n))$ operations.

Finding all solutions, i.e. exhaustively searching for additional solutions, is generally expensive. In the worst case, the number of solutions, and therefore the cost, is exponential with respect to the number of variables in the problem. In the best case, the number of solutions is small and no backtracking is needed to find all solutions. In such cases, the algorithm needs $O(np(n))$ operations per additional solution, so for k solutions, we find a total algorithmic complexity of $O(i(n) + knp(n))$. To determine the actual complexity of our method, $i(n)$ and $p(n)$ should be determined experimentally, for a wide range of topological problems.

The topological constraint solving method presented here may also be usable for other applications than the DFOM, i.e. to determine realisations for other models with a declarative topology. Mappings were given only for the topological constraints currently defined in the DFOM, but many other topological constraints can be mapped to Boolean constraints, and thus solved with this method.

With the geometric constraint solver presented in the previous chapter and the topological constraint solver presented this chapter, we can determine realisations and decide family membership for declarative models such as the DFOM. However, for creating and using models of families of objects, we also need to be able to determine parameter ranges for which the constraints in such a model are satisfied, and critical values corresponding to topological changes. To to this, we must first determine the parameter ranges for systems with only geometric constraints. A method for this is presented in the next chapter.

Chapter 6

Computing geometric parameter ranges

To instantiate members of a family, e.g. when exploring a family of objects (see Section 2.5), parameter values have to be specified. For some parameter values, no members exist, because it is not possible to satisfy the constraints in the model. In general, a user of a CAD system does not know exactly for which parameter values members exist, and for which values no members exist. In current systems, the user is informed that the constraints could not be satisfied only after regenerating the model with the new parameter values. Specifying parameter values thus becomes a trial-and-error process.

It would be very helpful if the range of parameter values for which members exist, could be calculated and presented to the user beforehand.

In this chapter, we present a method to compute parameter ranges for a system of geometric constraints. Parameter ranges for which both geometric and topological constraints are satisfied, are considered in Chapter 7.

Parts of this chapter have already been published in [van der Meiden and Bronsvoort, 2005a; van der Meiden and Bronsvoort, 2006a].

6.1 Basic approach

Prior work concerning parameter ranges in geometric constraint systems is scarce.

A solving approach for interval-based geometric constraints is presented in [Joan-Arinyo and Mata, 2001]. This approach can be used to find bounds for a parameter of a system of constraints such that a solution is feasible. However, it cannot deal with parameter ranges that consist of several disjoint intervals, and, because it is based on sampling, it cannot determine the exact intervals.

We present a method to determine the exact range of a parameter, represented by a set of intervals. The method computes the range for a

single parameter, referred to as the variant parameter. We assume the range to be computed is the range such that the intended solution of the system exists (see Section 4.5), although with the presented method we can also compute the parameter range for which some other solution, indicated by selection constraints, exists, or for which any solution exists. We do not consider the case where several parameters are varied simultaneously (see Section 2.5).

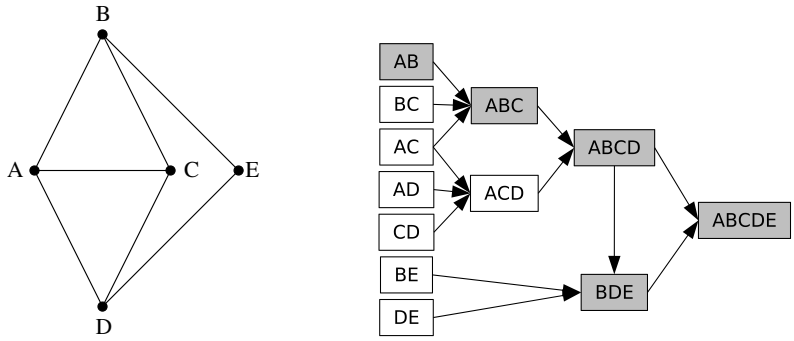
The considered constraint problems are systems of distance and angle constraints on points in 2D or 3D that are well-constrained. It is assumed that a geometric constraint solver is available that can find a decomposition of the system into subproblems, and the intended solutions for each subproblem and the system as a whole. The solver presented in Chapter 4 satisfies these criteria, because it determines a generic solution consisting of clusters, which represent subproblems, and for each cluster the intended solution can be specified by selection constraints.

Basically, our method first determines all critical values of the variant parameter. A critical value of a parameter is defined as any value c for which there is an arbitrarily small value ϵ, with $|\epsilon| > 0$, such that for c and $c + \epsilon$ the system has a different number of solutions. The idea behind this approach is that the constructibility of the system can only change at critical values, and not between two subsequent critical values. However, we cannot completely determine constructibility, because no complete solving method is currently known for 3D geometric constraint systems (see Chapter 4). Instead, we determine the solvability of the system, i.e. whether we can find solutions using our solver, which is indicative of constructibility. Our method determines the solvability of the system, for each interval between two subsequent critical values, by picking a value in each interval and solving the system with that parameter value. The parameter range is the set of intervals for which solutions can be found for the system.

Critical parameter values are related to degenerate subproblems. A subproblem is degenerate, for some parameter value, if for some arbitrarily close value it has a different number of solutions. In general, the solutions of a subproblem form a continuous class, i.e. a connected subset of the solution space. For parameter values for which the subproblem degenerates, solutions are found on the boundary of this class.

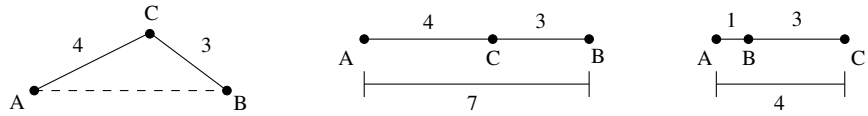
Only subproblems that are dependent on the variant parameter can degenerate. A subproblem is directly dependent on the variant parameter if the variant parameter is the parameter of one of the constraints in the subproblem. A subproblem is indirectly dependent on the variant parameter if it is dependent on another subproblem that is directly or indirectly dependent on the variant parameter.

The dependent subproblems can be found by analysing the generic solution of a problem. Consider the constraint problem in Figure 6.1(a) and the



(a) constraint problem (b) generic solution

Figure 6.1: A 2D constraint problem and its generic solution. Distance $\delta(A, B)$ is the variant parameter. The corresponding problem cluster (AB) and dependent clusters are shown in gray.



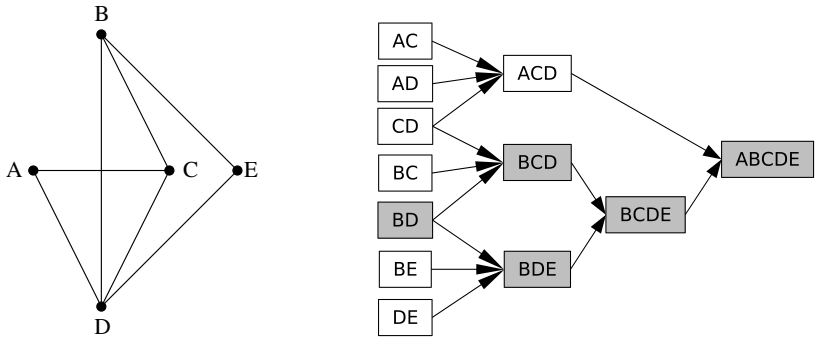
(a) $1 < \delta(A, B) < 7$ (b) $\delta(A, B) = 7$ (c) $\delta(A, B) = 1$

Figure 6.2: Degenerate cases for subproblem ABC with a single variant parameter $\delta(A, B)$, and $\delta(A, C) = 4$ and $\delta(B, C) = 3$.

generic solution for this system in Figure 6.1(b). Each cluster in the generic solution that is determined by a rewrite rule corresponds to a subproblem. In this case, all subproblems solutions are rigid clusters (see Section 4.2), so the point variables of a cluster are sufficient to identify a subproblem. From the generic solution we can infer that to solve this system, first the subproblems ABC and ACD are solved, independently, and merged into subproblem $ABCD$. When subproblem $ABCD$ has been solved, subproblem BDE can be solved and then merged with $ABCD$ into the final solution, represented by cluster $ABCDE$. If we consider the distance $\delta(A, B)$, constrained by cluster AB, to be the variant parameter, then subproblem ABC depends directly on the variant parameter, and subproblems $ABCD$, BDE and $ABCDE$ indirectly. Subproblem ACD does not depend on the variant parameter.

For each subproblem that is dependent on the variant parameter, several degenerate cases may exist, i.e. there can be several ways in which the subproblem degenerates. The degenerate cases of subproblem ABC are illustrated in Figure 6.2. The triangle ABC exists only if the triangle inequality is satisfied, i.e. if

$$\delta(A, C) + \delta(B, C) \geq \delta(A, B) \geq |(\delta(A, C) - \delta(B, C)|$$



(a) modified constraint problem (b) new generic solution

Figure 6.3: The problem of Figure 6.1 after removing the constraint on $\delta(A, B)$ and adding a constraint on $\delta(B, D)$. Clusters shown in gray are dependent on the new constraint.

If the distance $\delta(A, B)$ is the variant parameter, and $\delta(A, C) = 4$ and $\delta(B, C) = 3$, then there are two critical values, $\delta(A, B) = 1$ and $\delta(A, B) = 7$. For these values, ABC degenerates to a configuration of three points on a line (see Figure 6.2(b) and Figure 6.2(c)). In general, for each distance $\delta(A, B)$, $\delta(A, C)$ or $\delta(B, C)$ we can determine the critical values if the two other distance values are known.

Subproblem BDE in figure 6.1(b) is similar to the triangular subproblem ABC discussed above. However, the variant parameter, $\delta(A, B)$, is not in subproblem BDE and therefore the degenerate cases of this subproblem do not directly correspond to critical parameter values. Instead, we can determine the values of $\delta(B, D)$ for each degenerate case, and add a corresponding distance constraint to the system.

By removing the constraint on $\delta(A, B)$, the variant parameter, and adding a constraint on $\delta(B, D)$ that corresponds to a degenerate solution of BDE, we obtain the constraint problem in Figure 6.3(a). The generic solution for this system is shown in Figure 6.3(b). As can be seen, subproblem ABC has disappeared, subproblems ACD and BDE have remained, and a new subproblem BCD has appeared. From the solutions of this system, we can determine the critical parameter values, by measuring $\delta(A, B)$ in each solution.

In general, finding critical values is achieved by first removing the constraint corresponding to the variant parameter. Then, for each degenerate case of each dependent subproblem, constraints are added such that the subproblem degenerates, resulting in a new system of constraints for each case. Each such system is solved, and the critical values are the values of the variant parameter, measured in the solutions of each system.

Next, in Section 6.2, we describe a general approach to represent the

degenerate cases of different types of subproblems, using constraints. In Appendix A, the degenerate cases are listed for all rewrite rules used by the constraint solver described in Chapter 4, i.e. the degenerate cases are given for all types of subproblems supported by this solver. The details of the algorithm for computing parameter ranges are discussed in Section 6.3. The application of the algorithm to an example 3D constraint problem is demonstrated in Section 6.4.

6.2 Degenerate subproblems

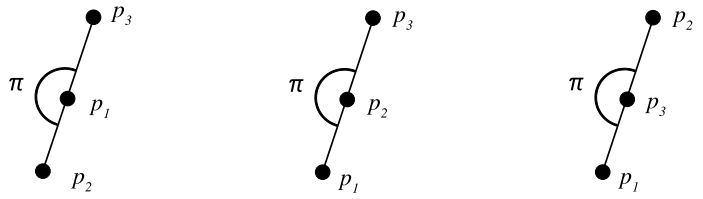
The term degenerate typically refers to limiting cases where a class of mathematical objects changes its nature to become another, usually simpler class [Weisstein, 2008a]. For example, a configuration of three points generally forms a triangle, but it can degenerate to a configuration of three points on a line, or to a configuration with one or more coincident points, as in Figure 6.2.

Subproblems that depend on the variant parameter have one or more degrees of freedom, i.e. infinitely many solutions. In general, the solutions of a subproblem with one or more degrees of freedom form a continuous class, i.e. a connected subset of the solution space. We use the term degenerate case to refer to a set of solutions on the boundary of such a class, i.e. a set of degenerate solutions.

For each type of subproblem, several degenerate cases may be distinguished. These degenerate cases can be represented by geometric constraints. If the subproblem under consideration has only one degree of freedom, as was the case for subproblems ABC and BDE in Figure 6.1(b), then it has a finite number of degenerate solutions. We can first determine these degenerate solutions, and then, for each solution, add a single constraint to the underconstrained system. This constraint may be any distance or angle constraint that is not in the original system, and the distance or angle value can be calculated from the degenerate solution.

The above approach fails if a subproblem has more than one degree of freedom, because it will then have an infinite number of degenerate solutions, so we cannot generate them one by one. This situation may occur if a subproblem depends indirectly on the variant parameter via two or more independent subproblems. In other words, the variant parameter induces more than one degree of freedom in the subproblem, via several paths in the generic solution.

Therefore, we represent the degenerate cases of any subproblem, regardless of whether it is directly or indirectly dependent on the variant parameter and whether it has one or more degrees of freedom, in the same way. Each degenerate case is represented by a constraint that is independent of the original constraints in the subproblem, in particular, independent of the



(a) $\measuredangle(p_3, p_1, p_2) = \pi$ (b) $\measuredangle(p_1, p_2, p_3) = \pi$ (c) $\measuredangle(p_2, p_3, p_1) = \pi$

Figure 6.4: Constraints corresponding to degenerate case $colinear(p_1, p_2, p_3)$ for a triangular subproblem.

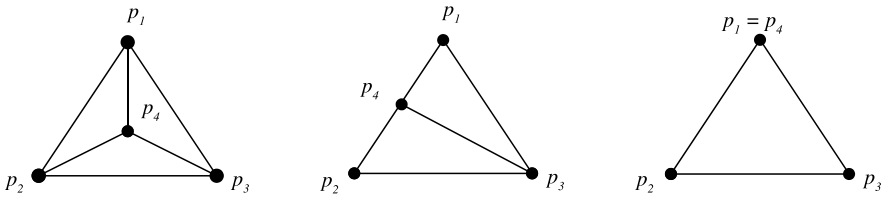
distance and angle values of constraints in the subproblem. A degenerate case can thus represent a finite or an infinite set of degenerate solutions.

The degenerate cases of the basic triangular subproblem with three known distances, e.g. ABC in Figure 6.2, can be represented by constraints as follows. In general, the solutions of this subproblem are configurations of the points A, B, and C such that ABC is a triangle. The solution can degenerate to a set of points on a line. If $\delta(A, B)$ is variable, then there are two such degenerate cases, one where the order of the points on the line is ACB (Figure 6.2(b)) and one where the order of the points on the line is ABC (Figure 6.2(c)). The first case can be enforced by a constraint $\measuredangle(A, C, B) = \pi$, such that point C is between point A and point B. The second case can be enforced by a constraint $\measuredangle(A, B, C) = \pi$, such that point B is between point A and point C.

More in general, the degenerate cases of a triangular subproblem occur when its points are co-linear. The constraint $colinear(p_1, p_2, p_3)$ is satisfied if one of the following constraints is satisfied: $\measuredangle(p_3, p_1, p_2) = \pi$, $\measuredangle(p_1, p_2, p_3) = \pi$ or $\measuredangle(p_2, p_3, p_1) = \pi$. All three constraints should be tried separately to solve the degenerate case. This is illustrated in Figure 6.4.

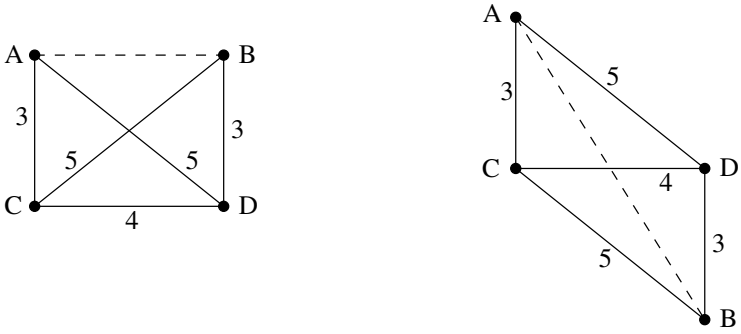
Alternatively, the co-linear constraint can be simply implemented by introducing a line L and constraints $coincident(L, p_1)$, $coincident(L, p_2)$ and $coincident(L, p_3)$. The mapping algorithm for geometric primitives discussed in Section 4.6 will effectively introduce the same angle constraints as presented above.

The basic triangular subproblem discussed above also degenerates if any two points in the problem are coincident. In that case, the problem has solutions only if the other distances in the triangle are of equal value. These degenerate cases can be enforced by constraining each of the distances to zero. For example, for the system in Figure 6.4, the extra degenerate cases are: $coindicent(p_1, p_2)$, $coincident(p_1, p_3)$ and $coincident(p_2, p_3)$. In total there are six degenerate cases for the basic triangular subproblem. These degenerate cases are listed for the corresponding rewrite rule, Rule 4, in



(a) $coplanar(p_1, p_2, p_3, p_4)$ (b) $colinear(p_1, p_2, p_4)$ (c) $coincident(p_1, p_4)$

Figure 6.5: Degenerate cases for a tetrahedral subproblem



(a) degenerate case corresponding to minimum value of $\delta(A, B) = 4$

(b) degenerate case corresponding to maximum value for $\delta(A, B) = 2\sqrt{13}$

Figure 6.6: Degenerate cases of a tetrahedral problem.

Appendix A.

The basic 3D subproblem is the tetrahedral problem with six distance constraints. The solution of this subproblem, in general, is a configuration of four points that form a tetrahedron with a non-zero volume. The subproblem degenerates if the points are co-planar, as illustrated in Figure 6.5(a), if any three points are co-linear (Figure 6.5(b)), or if any two points are coincident (Figure 6.5(c)). If permutations of the point variables are considered, then in total there are 11 degenerate cases. These degenerate cases are listed for the corresponding rewrite rule, Rule 12, in Appendix A.

An example tetrahedral problem is shown in Figure 6.6. Suppose that the distance $\delta(A, B)$ is the variant parameter. With five known distances, the problem is underconstrained in 3D. However, in 2D the problem is well-constrained and has two solutions, corresponding to $\delta(A, B) = 4$ and $\delta(A, B) = 2\sqrt{13}$, shown in Figures 6.6(a) and 6.6(b) respectively. These solutions correspond to the co-planar degenerate case of the tetrahedral problem.

To solve the degenerate case $coplanar(p_1, p_2, p_3, p_4)$, a new plane P is introduced and the constraints $coincident(P, p_i)$, $1 \leq i \leq 4$. As with the co-linear degenerate case, these constraints are mapped using the methods described in Section 4.6 to angle constraints on points.

A subproblem that merges two rigid clusters with shared points can also degenerate, when the clusters cannot be rotated and translated such that the shared points in one cluster are mapped onto the shared points in the other cluster. In 2D, two rigid clusters with two shared points, p_1 and p_2, can be rigidly merged, unless the points are coincident in the configuration of one or both clusters, i.e. if the subproblem is incidentally underconstrained. So, for the corresponding rewrite rule (Rule 1 in Appendix A), there is one degenerate case: $coincident(p_1, p_2)$.

In 3D, two rigid clusters with three shared points, p_1, p_2 and p_3, can be rigidly merged, unless two or three points are coincident, or the three points are co-linear. For the corresponding rewrite rule (Rule 9 in Appendix A), there are four degenerate cases: $coincident(p_1, p_2)$, $coincident(p_1, p_3)$, $coincident(p_2, p_3)$ and $colinear(p_1, p_2, p_3)$.

If the distance between the shared points is not equal in the configurations to be merged, then the problem is overconstrained. However, since we require that the original system of constraints is structurally well-constrained, this distance must always have the same value (see Section 4.3), and this degenerate case does not have to be considered when searching for critical values.

The subproblems considered, so far, are only a subset of the subproblems solved by the rewrite rules used by the geometric constraint solver discussed in Chapter 4. In Appendix A, for each rewrite rule the degenerate cases are listed and expressed by constraints.

6.3 Parameter range computation algorithm

In this section, we discuss the details of our approach to parameter range computation.

A pseudo-code algorithm for parameter range computation is presented in Algorithm 6.1. The algorithm takes two input arguments, a geometric constraint problem (argument GCP), and a variant parameter (argument VP). The variant parameter is mapped to a geometric constraint when a value is set for it (method `SetParameter`). If the value is cleared (method `FreeParameter`), then the geometric constraint is removed. Geometric constraints can be added to (method `AddConstraint`) and removed from the GCP (method `RemoveConstraint`). We assume that the generic solution (function `GenericSolution`) is automatically updated by the GCP, resulting in an efficient algorithm, but this is not strictly required (see Section 4.4 for an incremental solving algorithm).

The algorithm returns a set of intervals (R) that represents the parameter range. The set may contain open intervals (e.g. (y_1, y_2)), half-open intervals (e.g. $[y_1, y_2)$) and closed intervals (e.g. $[y_1, y_2]$). When intervals are added to the set, overlapping intervals are automatically combined, e.g. $(0, 1) + [1, 2] \rightarrow (0, 2]$.

To start, the algorithm asserts that the GCP is well-constrained, otherwise the result found by the algorithm is not a correct parameter range. It then determines the generic solution of the problem, and from the generic solution it determines which subproblems are dependent on the variant parameter. The algorithm then modifies the system of constraints by removing the constraint corresponding to the variant parameter. Now the GCP is underconstrained.

For each degenerate case of each subproblem that is dependent on the variant parameter, we add constraints to the GCP that correspond to the degenerate case, and we determine the generic solution of the modified sys-

Algorithm 6.1: Geometric parameter range computation

```
function GeometricParameterRange (GCP,VP)
  GCP: geometric constraint problem
  VP:  variant parameter
begin
  assert WellConstrained(GCP)
  generic := GenericSolution(GCP)
  dependent := DependentSubproblems(generic,VP)
  FreeParameter(VP, GCP)
  Y := empty set of critical values
  R := empty set of intervals
  for each subproblem in dependent
    cases := DegenerateCases(subproblem)
    for each case in cases
      AddConstraints(case, GCP)
      newgeneric := GenericSolution(GCP)
      cluster := find Rigid in newgeneric containing VP and case
      for each configuration in cluster
        y := calculate VP from configuration
        add y to Y
        add [y,y] to R
      RemoveConstraints(case, GCP)
  for each subsequent interval (y1,y2) in Y
    x := (y1+y2)/2
    SetParameter(VP=x,GCP)
    if WellConstrained(GCP) then add (y1,y2) to R
  return R
end
```

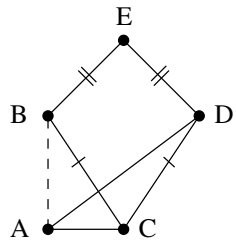


Figure 6.7: System with underconstrained degenerate cases.

tem (preferably by an incremental solving method). The subproblems that depend on the variant parameter will have disappeared in the new generic solution, and new subproblems that depend on the newly added constraints will have appeared.

If the modified constraint system is well-constrained, then the particular solutions of the system are found as configurations of the solution cluster of the generic solution. In general, the particular solutions are determined from the cluster that contains the variant parameter and the constraints imposed by the degenerate case. For each particular solution of the system, we determine a critical parameter value (`y`), which is stored in a set (`Y`), such that any value can occur in the set only once. We also add this value as a closed interval (`[y,y]`) to the parameter range (`R`).

If the modified system is structurally or incidentally overconstrained, then no solutions are found and the current degenerate case cannot occur for any value of the parameter.

If the modified system is structurally or incidentally underconstrained, then it may still be possible to find a critical value for the variant parameter, depending on which subproblems are underconstrained. If a rigid cluster exists in the generic solution that contains the constraints corresponding to the variant parameter and the constraints corresponding to the current degenerate case, then from the configurations of that cluster, the values of the variant parameter are determined, and stored as critical parameter values. If no such cluster exists, then no value is determined for the variant parameter, and thus no critical values are found.

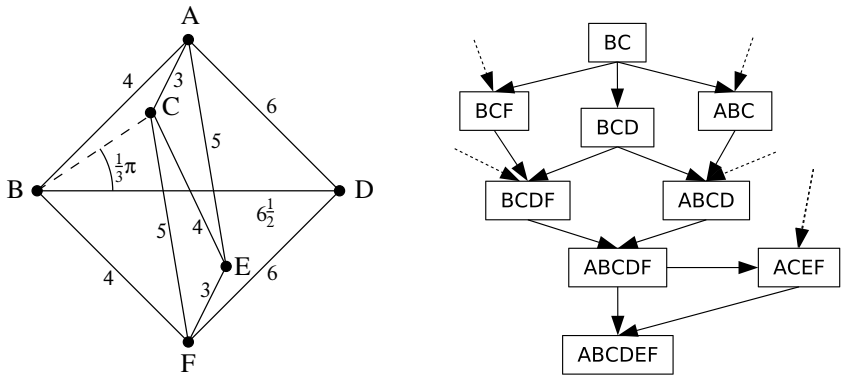
For example, consider the system of Figure 6.7, where $\delta(A, B)$ is the variant parameter. Given that $\delta(B, E) = \delta(D, E)$, then for the degenerate case of triangle BDE we find $\delta(B, D) = 0$. By propagating this value, we can solve subsystem $ABCD$, and calculate the critical value for $\delta(A, B)$. When applying this critical value to the original system, we find that points B and D are again coincident, such that merging $ABCD$ and BDE results in an incidentally underconstrained situation. So this critical value should not be part of the parameter range, and the intervals adjacent to it are half

open intervals.

After all critical values have been determined, the parameter range is determined by repeatedly solving the original constraint system (which has been restored after computing critical values). For each interval (y_1, y_2) between two subsequent critical parameter values, y_1 and y_2, a value in the middle of the interval is chosen, and assigned to the variant parameter (SetParameter). If the system is well-constrained, then the open interval is added to the parameter range (R). If the new interval is adjacent to any interval(s) already in the parameter range, then these intervals are merged into a new, open, half-open or closed interval. This is the result returned by the algorithm.

From the discussion above, it is clear that geometric parameter range computation is more expensive than merely solving a given geometric constraint problem. To compute the parameter ranges for a problem, several constraint problems of similar complexity, representing degenerate cases, have to be solved. Also, when solving the original problem, we can use selection constraints to find the intended solution of the problem quickly (see Section 4.5). However, when the problem is modified to represent degenerate cases, subproblems need to be solved that are not in the original problem. For these new subproblems, we cannot generate solution selection constraints, since these extra constraints do not occur in the original problem. If we would do this, then some critical values might be missed. This means that for these subproblems, all solutions must be generated.

In a worst case scenario, all the subproblems solved to find a particular degenerate case are new, i.e. they do not occur in the original problem. Then the number of solutions generated in the parameter range computation is exponential with respect to the number of constraints in the problem. However, it seems unlikely that such worst cases will occur in practice, in particular for typical CAD models. Such models are often composed of features of which the shapes are determined independently by shape parameters. The decomposition of the corresponding geometric constraint problem typically yields a decomposition with rigid subproblems that correspond to these features. Changing a parameter value of one feature typically effects only the relative position and orientation of a few other features, and modifying the system of constraints to find critical values will result in only few new subproblems. So, we expect that for typical CAD problems, we can use the intended solution to reduce the cost of constraint solving during parameter range computation. However, the feasibility of our method for large CAD models used in practice must still be verified experimentally.



(a) constraint problem with variant parameter $\delta(B, C)$

(b) part of generic solution dependent on $\delta(B, C)$

Figure 6.8: 3D example problem

6.4 Example constraint problem

The example 3D constraint problem considered in this section is shown in Figure 6.8(a), and its generic solution is shown in Figure 6.8(b). The distance between point B and point C is chosen to be the variant parameter, i.e. the parameter for which the range is to be computed. From the generic solution, we infer that the following subproblems are dependent on $\delta(B, C)$: ABC, BCD, BCF, $ABCD$, $BCDF$ and $ABCDF$, $ACEF$, and $ABCDEF$. For these subproblems, we must find the degenerate solutions and the corresponding critical values of the variant parameter.

Subproblem ABC is the simplest type of triangular subproblem, and degenerates for $\delta(B, C) = 1$ and $\delta(B, C) = 7$. Subproblem BCF is similar and degenerates for $\delta(B, C) = 1$ and and $\delta(B, C) = 9$. However, for these values, the complete system is overconstrained, thus no critical values are recorded.

Subproblem BCD degenerates for $\delta(B, C) = 0$. Again, for this value, the complete system is overconstrained, thus no critical values are recorded.

Subproblem $ABCD$ is shown in Figure 6.9. This subproblem degenerates for $\delta(B, C) \approx 1.01$ and $\delta(B, C) \approx 6.97$. Once again, for these values, the complete system is overconstrained, and no critical values are recorded. Subproblem $BCDF$ is similar to $ABCD$, and no critical values are found for this subproblem either. These clusters are merged into subproblem $ABCDF$, for which also no critical values are found.

Subproblem $ACEF$ has two degenerate cases, which correspond to a constraint $\delta(A, F) = 4$ and $\delta(A, F) = 2\sqrt{13}$. This subproblem depends indirectly on the variant parameter, therefore we modify the system of constraints by removing the constraint on $\delta(B, C)$, and adding a constraint

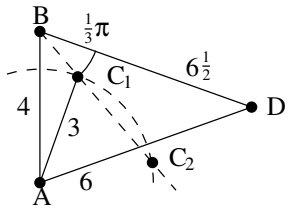
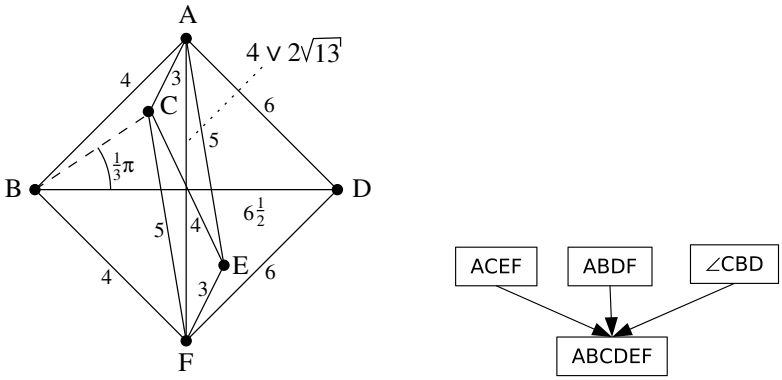


Figure 6.9: Subproblem $ABCD$.



(a) modified constraint problem (b) generic solution

Figure 6.10: Modified constraint problem corresponding to the degenerate solutions of subproblem $ACEF$.

$\delta(A, F) = 4$ or $\delta(A, F) = 2\sqrt{13}$. The modified constraint problem is shown in Figure 6.10(a), and the new generic solution for this problem in Figure 6.10(b). This generic solution is used to find the critical parameter values for both degenerate cases of the subproblem. In the generic solution, there is a new subproblem $ABDF$, for which no intended solution has been defined. For each of the two degenerate cases, this subproblem has two solutions, resulting in four critical values for $\delta(B, C)$: $c_1 \approx 1.49$, $c_2 \approx 1.90$, $c_3 \approx 2.50$ and $c_4 \approx 6.76$.

For subproblem $ABCDEF$, again no critical values are found. No other subproblems need to be tested for degenerate cases.

Now that we have determined all critical values for $\delta(B, C)$, we test the solvability of the system by picking a parameter value in each interval between two subsequent critical values, and solving the system. We find that the intended solution for this problem exists for $\delta(B, C) \in [c_1, c_2] \cup [c_3, c_4]$.

To determine this parameter range, the problem was solved for a relatively small number of degenerate cases and intervals. In comparison, a naive sampling approach, i.e. an algorithm that simply tries to solve the system for different values, would need to solve the problem a very large

number of times to achieve any reasonable accuracy. Also, a sampling approach may not find all intervals in the parameter range, since the minimum and maximum parameter values and the minimum sampling resolution are not known.

The method presented in this chapter can compute parameter ranges for systems of geometric constraints only. There are many useful applications for such a method. However, for many CAD applications, in particular the design of families of objects, we also need to consider the topology of the model and the topological constraints in the model. This is discussed in the next chapter.

Chapter 7

Tracking topological changes

When instantiating members of a family, e.g. when exploring a family of objects, being able to relate the parameters with the topology of a model is important (see Section 2.5). In particular, we want to determine for which specific parameter values the topology of the model changes, i.e. the critical parameter values corresponding to topological changes. Also, we want to determine the parameter ranges for which the topological constraints in a model are satisfied.

The method presented in this chapter tracks topological changes in a parametric model, such that the critical parameter values and parameter ranges can be determined. It can be used for parametric or feature-based models, e.g. the DFOM, as long as (1) geometry is specified declaratively, using geometric primitives and constraints, and (2) every topological entity in the model is determined by an intersection of geometric primitives.

Parts of this chapter have already been published in [van der Meiden and Bronsvoort, 2007b].

7.1 Relating parameters and topology

The relation between parametric representations and geometric representations has been studied by Shapiro et al., e.g. [Shapiro and Vossler, 1995; Raghothama and Shapiro, 1998; Raghothama and Shapiro, 2000]. They focus on finding a mapping between the parameter-space family of a model, representing the objects that can be obtained by modifying parameters, and the representation-space family of the model, representing the objects that can be obtained by continuity-preserving operations on the geometric representation. However, no method is given that explicitly represents such mappings, such that it can be determined for which parameter values the topology of the geometric representation changes.

In [Hoffmann and Kim, 2001], the parameter range problem is considered for a 2D polygon with only horizontal and vertical line segments and distance constraints between them. They determine the range for a single distance parameter such that the topology of the polygon does not change. The considered constraint system is very simple, but the authors do make some useful observations concerning the problem in general. In particular, they observe that a system of geometric constraints in general has a large number of solutions, exponential with respect to the number of constraints in the problem, and that a different parameter range may be found for each solution. Also, they suggest that only one parameter at a time should be considered, because the combined parameter range for n parameters is a subset of n-dimensional space which will be very difficult to determine and to present to the user.

To be able to relate parameters and topology, we must consider the relation with feature geometry also, because the topology of a model is determined by the geometry of the features, which in turn are determined by the parameter values. If the geometry of the features in the model is determined procedurally, as in most CAD systems, e.g. by creating 3D extrusions from 2D sketches, then it will be difficult, in general, to relate changes in the topology of model, via the geometry of the features, back to the parameters. The method we present here therefore requires that all feature geometry is specified declaratively, using geometric variables and constraints. Such systems can be analysed and solved with the methods presented in Chapter 4.

The topology of a model is typically represented by a cell-complex representation, e.g. a B-rep or a cellular model. The models contain topological entities (simply referred to as entities), e.g. vertices, edges, faces and volume cells, and topological relations between entities, e.g. which faces are on the boundary of a volume cell. These representations are often constructed from feature shapes by means of regularised Boolean operations, i.e. the regularised union, difference and intersection operations. Typically, these operations are implemented by fairly complex procedures, but topological entities are always determined by intersecting feature geometry. In particular, every entity is a subset of the intersection of the feature geometry from which it was determined. Not every entity can be uniquely described by an intersection of feature geometry, because the entity may be a real subset of the intersection, and thus other entities with the same description may exist [Rappoport, 1997]. However, we can state that if the intersection disappears, the entity must also disappear from the model.

For tracking topological changes, we thus need to generically represent topological entities as intersections of feature geometry, with the understanding that the representation is not unique. Examples of such generic representations of topology can be found in the Generic Geometric Complex

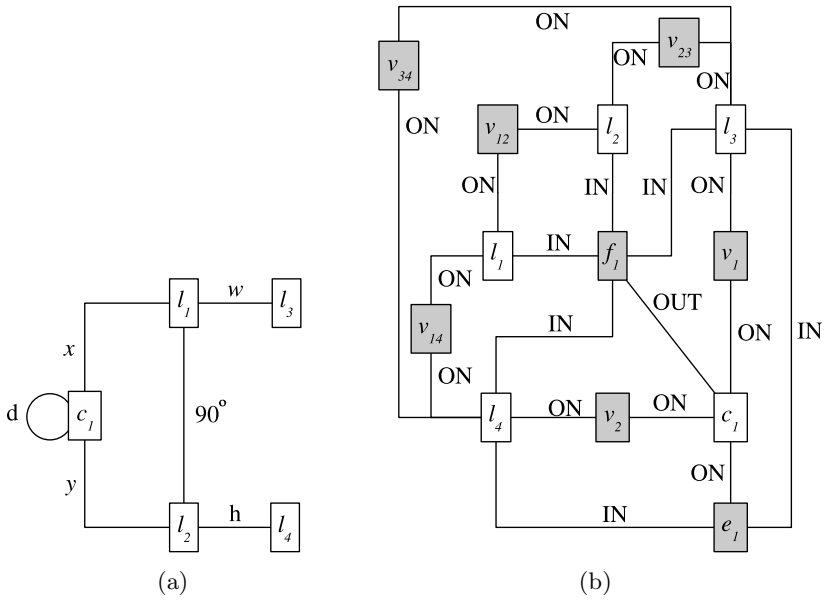


Figure 7.1: Constraint graphs relating (a) parameters and carriers, (b) carriers and entities. White boxes represent carriers and gray boxes represent entities. Edges in the graphs represent constraints, labelled with the name of a parameter or the type of the constraint.

(see Section 2.1 and [Rappoport, 1997]), the Constructive Topological Representation (see Section 2.2 and [Raghothama, 2006]) and the DFOM (see Section 3.2). Most CAD systems do not create such generic topological representations, but this can be done for most geometric representations, e.g. for the B-rep or the cellular model, by generating each entity procedurally and then relating it to the original features that determined it.

In the description of our method, we assume that there is a parametric model (PM) with geometric constraints between parameters and carriers, that realisations can be generated and represented by a cellular model (CM), and that each entity in the CM can be related to the carriers in the PM with subspace constraints (see Section 3.2). The relations between parameters, carriers and topological entities can be represented by two constraint graphs, such as shown in Figure 7.1, which corresponds to the DFOM model shown in Figure 3.7 on page 38.

The parametric model is represented by the constraint graph in Figure 7.1(a). The parametric model defines four linear carriers, $l_1 ... l_4$, and one circular carrier, c_1. The carriers are related by geometric constraints to the parameters h, w, x, y and d.

A CM, corresponding to a realisation that can be generated from the PM, is represented by the constraint graph in Figure 7.1(b). This constraint

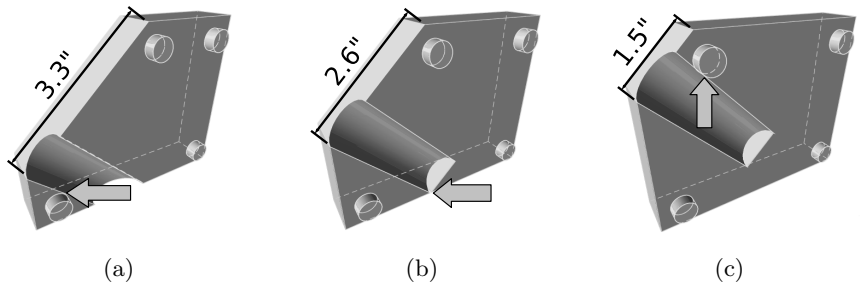


Figure 7.2: Critical parameter values for an example model. Arrows indicate where features are touching.

graph contains subspace constraints that relate CM entities to carriers. The geometry of entities is determined by intersections of subspaces of carriers, just like constructs in the DFOM (see Section 3.2). For example, edge e_1 is determined by the intersection of the circular carrier c_1 with the half-spaces induced by the carriers l_3 and l_4. Vertices v_1 and v_2 are determined by intersections of carrier c_1 with carriers l_3 and l_4 respectively. Face f_1 is determined by the intersection of the half-spaces induced by the carriers c_1, l_1, l_2, l_3 and l_4.

Note that in a DFOM, relations between parameters, carriers and constructs are represented, but not the relations between these and CM entities. The latter relations can only be determined after a CM has been generated from a geometrically well-constrained DFOM (see Section 3.4).

7.2 Computing critical values

A critical value of the variant parameter is here defined as any value c for which there is an arbitrarily small value ϵ, with $|\epsilon| > 0$, such that for c and $c + \epsilon$ the realisations of the model have different topologies. Critical values thus correspond to changes in the topology of the realisations of a model, when a parameter is varied continuously.

Critical values often correspond to features that are touching or tangent, as shown in Figure 7.2. The figure shows the realisations of a model for different critical parameter values. For each such critical value, model topology is different compared to the model topology for values around it.

The topology of a realisation is represented by a CM. However, we do not explicitly compare CMs to detect topological changes, because the entities in one realisation's CM cannot be easily mapped onto the entities of another realisation's CM. Instead, topological changes are associated with degenerate entities.

Degenerate entities are entities that should not occur in the geometric representation of a model, i.e. the CM. Examples of degenerate entities are

edges of length 0, faces with area 0, and cells with volume 0. Such entities should be represented by lower-dimensional entities instead. Entities that represent disjoint point sets, or point sets that can be decomposed into point sets of different dimension, should be split into several entities (see Section 3.4), and are therefore also degenerate.

Although degenerate entities should never occur in geometric representation, we can impose constraints on the carriers that determine an entity, such that the entity degenerates if the constraints are satisfied. For each entity in the geometric representation, one or more degenerate cases can be formulated in terms of geometric constraints. For example, a vertex, determined by the intersection of two lines, degenerates when the intersection of the two lines no longer exists. This corresponds to a constraint specifying that the lines should be parallel. Degenerate cases of entities are further discussed in Section 7.3.

The basic approach for computing critical values is as follows. First we remove the constraint corresponding to the variant parameter, i.e. it is considered to be a variable without a fixed value. Effectively, a constraint will be removed from the geometric constraint system. We then determine which entities were dependent on the variant parameter and may therefore degenerate. For each degenerate case of each entity, we add specific constraints to the system. By solving the modified system, we obtain values of the variant parameter for which an entity degenerates, and thus the topology of the model must change, i.e. critical parameter values. By repeating this process for every entity that is dependent on the variant parameter, we can obtain all critical values.

To be able to track all topological changes, it is essential that the entities of the CM partition the complete Euclidean space into volumetric cells. Because space is then covered by cells, any topological change corresponds to a degenerate cell or a degenerate lower-dimensional entity on the boundary of a cell. Thus, the CM must also represent space outside the object with one or more cells. This may be accomplished by considering the empty space in the initial CM as an entity. When entities are added to the CM, this initial entity is split. Another possibility is to add a volume entity to the CM corresponding to a bounding box or the convex hull of all entities.

Note that we assume here that all features in the model are represented in the initial CM by at least one entity. If some feature is not represented by any entity in the CM, then the feature's geometry cannot be related to the topology of the model, and it cannot be guaranteed that all critical parameter values will be found.

As stated earlier, only entities that are dependent on the variant parameter can degenerate, i.e. entities of which the geometry changes when the value of the variant parameter is changed. To determine whether an entity is dependent on the variant parameter, we inspect the decomposition

of the system of geometric constraints into its rigid subsystems, i.e. rigid clusters (see Section 4.2). If we remove the constraint corresponding to the variant parameter from the PM, then the system of constraints defined by the PM becomes underconstrained. In the decomposition, the carriers that have relative degrees freedom will be in different clusters. Entities that are determined by carriers in different clusters are dependent on the variant parameter. A pseudo-code algorithm to determine dependent entities is presented as Algorithm 7.1.

Figure 7.3 shows the decomposition of the system of constraints on carriers from Figure 7.1(a), given that parameter x is the variant parameter, combined with the entities from Figure 7.1(b). Here, c_1 is a cluster, and l_1, l_2, l_3, and l_4 form a cluster. Entities v_{12}, v_{23}, v_{34} and v_{14} are dependent on only one cluster. Entities v_1, v_2, e_1 and f_1 are dependent on both clusters, and therefore these entities are dependent on the variant parameter.

The CM, when generated for different parameter values, may contain different entities, and therefore, different critical values may be found. Thus, to obtain all critical values, the CM must be regenerated for several values of the variant parameter, such that degenerate cases can be solved for all possible entities.

The algorithm therefore tracks topological changes as follows. For some value of the variant parameter, the *current parameter value*, it generates the CM and the system of constraints, relating entities to parameters. Next, it determines which entities are dependent on the variant parameter, and it solves all degenerate cases for these entities. The algorithm then determines the interval for which the topology of the current CM does not change, called a *stable interval*. This is the interval between the largest of the critical values smaller than the current parameter value, and the smallest critical value larger than the current parameter value (see Figure 7.4).

Algorithm 7.1: Determine the dependent entities for a given parameter

```
function DetermineDependentEntities(p, G)
  p:  parameter
  G:  constraint graph
begin
  E := empty set of entities
  R := empty set of clusters
  remove p from G
  R = RootClusters(G)
  for each entity e in G
    if e constrained to carriers in more than one cluster in R then
      add e to E
  return E
end
```

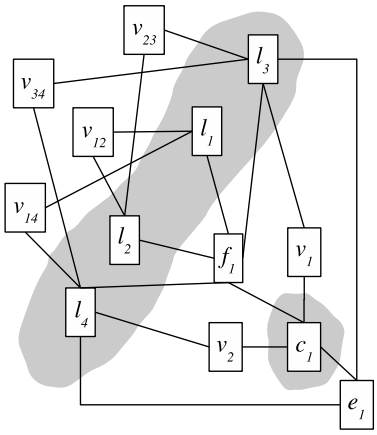


Figure 7.3: Root clusters, shown by the gray areas, for the constraint system from Figure 7.1 with variant parameter x.

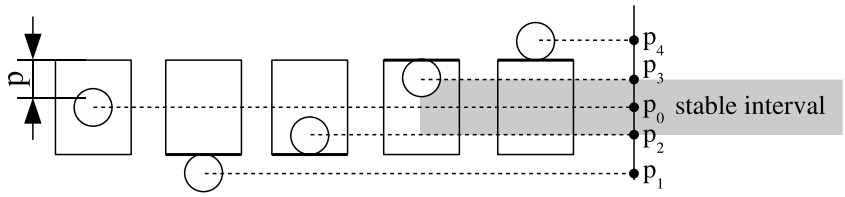


Figure 7.4: Stable interval for a model with a box and a circle, with a variant parameter p. The current parameter value is p_0. Four degenerate cases are found, resulting in four critical parameter values: $p_1 ... p_4$. The stable interval is $[p_2, p_3]$.

After a stable interval has been determined, the tracking algorithm picks a new current parameter value arbitrarily, but outside any previously determined stable interval (not just outside the last determined stable interval). A new CM is generated for this parameter value, and critical values are determined as before. This process is repeated until the domain of the variant parameter is completely covered by stable intervals. By definition, new critical values are never found in any previously determined stable interval. Thus, when the whole parameter domain is covered by stable intervals, all critical values have been found.

Algorithm 7.2 determines the stable interval for a model and the current parameter value, and Algorithm 7.3 the critical values for a given PM and variant parameter.

The computational complexity of this method is very high, since in the worst-case scenario, we must solve degenerate cases for every combination of the carriers in the model. However, we believe that such worst cases will not occur in practice, because parameters will typically affect the geometry and topology of the model only locally, and consequently only for dependent entities do we have to solve degenerate cases. The efficiency of the method is improved by using an incremental constraint solving method, as discussed

Algorithm 7.2: Determine stable interval for a given parameter value

```
function ComputeStableInterval(PM, p, x)
  PM: parametric model
  p:  variant parameter
  x:  current value of p
begin
  V := empty set of parameter values
  CM := regenerate CM from PM
  G := ConstraintGraph(PM) + Constraintgraph(CM)
  E := DetermineDependentEntities(p, G)
  V := [-∞, ∞]
  for each entity e in E
    C := DegenerateCases(e)
    for each case c in C
      add constraints of c to G
      solve G
      for each solution s of G
        v := compute value of p from s
        add v to V
      remove constraints in c from G
  c1 = max(v in V | v < x)
  c2 = min(v in V | v > x)
  return [c1, c2]
end
```

in Section 4.4. Also, the cellular model can be updated efficiently, because for each new parameter value, the cellular model only needs to be updated locally [Bidarra et al., 2005a]. Finally, in Algorithm 7.3, stable intervals are computed for arbitrarily chosen values of the variant parameter. It may be possible to make a better choice for these values, so that fewer degenerate cases have to be considered, but such optimisations have not yet been considered.

Algorithm 7.3: Determine critical values for a given parameter

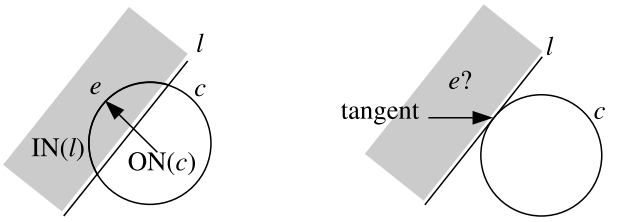
```
function FindCriticalValues(PM, p)
  PM: parametric model
  p:  parameter
begin
  v := current value of p
  I := empty set of intervals
  C := empty set of critical values
  do
    [c1,c2] := ComputeStableInterval(PM, p, v)
    add c1 to C; add c2 to C
    add [c1,c2] to I
    v := arbitrary value not in I, or None
  until v = None
  return C
end
```

7.3 Degenerate entities

An entity of a particular type, i.e. vertex, edge, face or cell, degenerates when its generic representation no longer represents a point set that is valid for that type of entity. For example, a face that is ON a carrier and IN one or more other carriers, degenerates when the geometry of the carriers is changed such that the combination of those carriers represents a curve, a point or an empty set. For any entity that is dependent on the variant parameter, one or more degenerate cases can be formulated and enforced via constraints on its carriers.

Figure 7.5 shows an example of a degenerate case. An edge is determined by the intersection of a circular carrier and a half-plane defined by a linear carrier. In general, the intersection is a curve, or does not exist. The intersection cannot be a point, because the entity is constrained IN and not ON the linear carrier. If we add constraints to the system such that the carriers are tangent, then the entity is degenerate.

The carriers on which constraints are imposed are determined from the decomposition of the system of constraints in the model. For each pair of carriers in different root clusters, i.e. each pair of carriers with relative



(a) generic case (b) degenerate case

Figure 7.5: (a) shows an edge e determined by the intersection of a circular carrier c and a half-plane defined by a linear carrier l. (b) shows that the edge degenerates when the carriers are tangent.

degrees of freedom, we add constraints such that, if the new system is well-constrained, then the entity is degenerate, i.e. the combination of the carriers no longer represents a point set that is valid for the given type of the entity.

Table 7.1 lists all possible topologically different intersections of two carriers, where a carrier is either a plane, sphere or cylinder. Each intersection in the table can be enforced by imposing the given constraints on the carriers. The same can be done for combinations of other geometric primitives, e.g. cones and tori. With more carrier types, however, enumerating all combinations would be rather elaborate, and a more generic approach to finding degenerate intersections would be preferable. This will not be considered here further.

For each pair of carriers in a dependent cluster, we first determine the type of the intersection of the carriers by verifying the constraints in the table. Next, for all other intersections listed in the table, we add the given constraints to the system of constraints in the model, and determine whether the new system is well-constrained.

Consider, for example, a curve that is constrained ON a spherical carrier and ON a cylindrical carrier (known as Viviani's curve, see [Weisstein, 2008b]). The variant parameter, d, is the distance of the centre of the sphere to the axis of the cylinder. Given are the radius of the cylinder, r_c, and the radius of the sphere, r_s, and in this example $r_c < r_s$. Suppose that in the current configuration, the intersection of the carriers is a closed curve, represented by a single edge in the CM. From Table 7.1 we can infer the constraints to force the carriers into different configurations. We find that the curve degenerates to a point for $d = r_s + r_c$ and to a 'figure 8' for $d = r_s - r_c$. For these configurations, adding the corresponding constraints results in a well-constrained system. For the other configurations of the sphere-cylinder intersection listed in Table 4.1, the system of constraints is either under- or overconstrained.

An entity that is constrained IN or OUT with respect to some carriers

intersection	**constraints**
plane-plane	
none	parallel, $d \neq 0$
line	not parallel
coincident	parallel, $d = 0$
plane-sphere	
none	$d > r_2$
point	$d = r_2$
circle	$d < r_2$
plane-cylinder	
none	parallel, $d > r_2$
line	parallel, $d = r_2$
ellipse	not parallel
two lines	parallel, $d < r_2$
sphere-sphere	
none	$d > r_1 + r_2$
point	$d = r_1 + r_2$
circle	$d < r_1 + r_2$
coincident	$d = 0$, $r_1 = r_2$
sphere-cylinder	
none	$d > r_1 + r_2$
point	$d = r_1 + r_2$
closed curve	$r_{max} - r_{min} < d < r_1 + r_2$
'figure 8'	$d = r_1 - r_2$, $r_1 > r_2$
2× closed curve	$d < r_1 - r_2$, $r_1 > r_2$
point	$d = r_2 - r_1$, $r_2 > r_1$
none	$d < r_2 - r_1$, $r_2 > r_1$
cylinder-cylinder	
none	parallel, $d < r_{max} - r_{min}$
line	parallel, $d = r_{max} - r_{min}$
two lines	parallel, $r_{max} - r_{min} < d < r_1 + r_2$
line	parallel, $d = r_1 + r_2$
coincident	parallel, $d = 0$, $r_1 = r_2$
none	$d > r_1 + r_2$
point	$d = r_1 + r_2$
closed curve	$r_{max} - r_{min} < d < r_1 + r_2$
'figure 8'	$d = r_{max} - r_{min}$
2× closed curve	$d < r_{max} - r_{min}$
2× 'figure 8'	$d = 0$, $r_1 = r_2$

Table 7.1: Intersections of pairs of carriers and corresponding constraints. Legend: r_1 = the radius of carrier 1, if it is a sphere or cylinder. r_2 = the radius of carrier 2, if it is a sphere or cylinder. $r_{min} = min(r_1, r_2)$. $r_{max} = max(r_1, r_2)$. d = distance between any two of: a plane, the centre of a sphere or the axis of a cylinder.

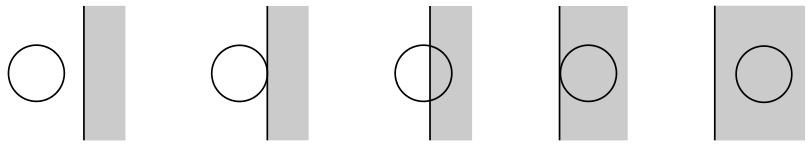


Figure 7.6: Five different configurations of a half-plane and a circle, corresponding to three different intersections of the corresponding carriers, i.e. the carriers have either zero, one or two intersection points.

does not always degenerate when the intersection of the carriers changes. For example, the entity that we have seen illustrated in Figure 7.5, is constrained ON with respect to a circular carrier and IN with respect to a linear carrier. Figure 7.6 shows the topological variants of a circle, defined by a circular carrier, and a half-plane, defined by a linear carrier. In this example, there are five different topological configurations, corresponding to only three different intersections of the carriers, such that the carriers intersect in zero, one or two points. The entity defined above degenerates if the carriers are constrained tangent, such that there is only one intersection point. This corresponds to two different topological configurations, i.e. the circle is either outside or inside the linear half-plane. In the first configuration, the entity is degenerate, in the second configuration, it is not.

Thus, depending on the subspaces of the carriers to which an entity is constrained, it may or may not degenerate when the configuration of those carriers changes. Therefore, we regenerate the entity from the carriers after solving the modified system of constraints, and we verify whether or not the resulting entity is of the same type as the original entity. If it is of a different type, then the entity degenerates for the given constraints, and the value of the variant parameter is a critical value.

For solving the systems of constraints corresponding to degenerate cases, a 3D constraint solver is needed that supports the constraints listed in Table 7.1, i.e. distances and angles on planes, cylinders and spheres. Inequalities do not have to be solved but should be used to select solutions. The geometric constraints solver presented in Chapter 4 satisfies these requirements.

7.4 Parameter range computation

The range of a given parameter is the set of values corresponding to valid models. One possible definition for the set of valid models is the set of models with the same topology as a given prototype [Shapiro and Vossler, 1995], corresponding to a representation-space family. The corresponding parameter range is equivalent to the stable interval, computed for the parameter value in the prototype.

However, assuming that only a single representation-space family is of interest, is rather limiting from a designer's perspective. A designer is not interested in one topological variant of the model per se, but rather wants the topology of the model to correspond with his or her design intent. Topological aspects of the design intent can be expressed by topological constraints. We define the parameter range as the set of parameter values for which all constraints in the model are satisfied, including geometric and topological constraints.

For a given variant parameter, the parameter range can be determined using Algorithm 7.4. The algorithm first determines the geometric parameter range, as discussed in Chapter 6. Then we determine all critical parameter values corresponding to topological changes, as discussed in Section 7.2. Because we have first computed the geometric parameter range, we can reduce the number of times the topological tracking algorithm needs to compute the stable interval for different parameter values, by discarding those parameter values for which the system has no geometric solutions.

When all critical values have thus been determined, then for each interval between two subsequent critical values, we pick a parameter value in that interval and we generate the CM for that value. For this CM we test whether the topological constraints in the model can be satisfied. If so, then the whole interval is part of the parameter range, because all objects in the interval have the same topology, and thus satisfy the topological constraints. Each interval between two subsequent critical values can thus be marked as part of the parameter range, or not part of the parameter range. For the

Algorithm 7.4: Determine the parameter range for a given parametric model and parameter

```
function FindParameterRange(PM, p)
  PM: parametric model
  p:  parameter
begin
  R := empty set of intervals
  G := GeometricParameterRange(PM, p)
  C := FindCriticalValues(PM, p)
  for each subsequent pair c1,c2 in G+C
    if TopologicalConstraintsSatisfied(PM, p, c1) then
      add [c1] to R
    if TopologicalConstraintsSatisfied(PM, p, c2) then
      add [c2] to R
    if TopologicalConstraintsSatisfied(PM, p, (c1+c2)/2) then
      add (c1, c2) to R
  return R
end
```

critical values we can also determine whether the topological constraints in the model are satisfied. Thus, we have determined exactly for which values and intervals the topological constraints are satisfied.

Knowing the parameter range can help a designer to maintain model validity while changing a parameter of the model. This can be very helpful when fine-tuning a model or when exploring a family of objects. Also, presenting critical parameter values to the designer, may help to understand the family model better, in terms of the different objects that it represents. Although the relation between several parameters cannot be explicitly represented, the effect of one parameter on the range and critical values of another parameter can be explored, by computing these for different values for the first parameter.

Parameter ranges and critical values are therefore very useful tools for modelling families of objects.

Chapter 8

Conclusions and future research

This thesis addresses the question of how to model families of objects. Current CAD systems are not adequate for modelling families of objects, because they create history-based models, which cannot be used to fully specify and maintain feature semantics, and they offer insufficient support for exploring families of objects. To overcome the limitations of current systems, we have proposed that (1) families of objects should be modelled declaratively, using features with geometric and topological constraints, and (2) for creating and using models of families of objects, methods for computing parameter ranges and critical values are needed.

To demonstrate the feasibility and advantages of a declarative model for families of objects, we have presented such a model, the DFOM. We have also presented methods for solving systems of geometric and topological constraints, which are needed to determine family membership and to find realisations of such models. Finally, we have presented methods that can compute parameter ranges and critical values. These methods can be used for the DFOM, but also for other models, as long as geometry and topology can be related to parameters via constraints.

In this final chapter, we first discuss the feasibility and advantages of our solution. Then we discuss our implementation and other possible applications. Finally, we discuss the limitations of our solution and some directions for future research.

8.1 Feasibility and advantages of the approach

The DFOM does not have the main problems associated with history-based models. In particular, it can correctly maintain feature semantics, and is not affected by the feature ordering problem. The model is based on the Semantic Feature Model, but does not suffer from the ambiguity of feature

dependency analysis, which always determines a single realisation. This realisation may not satisfy the topological constraints in the model, even though other realisations exists that do. Instead, we find all possible realisations that satisfy the topological constraints, using a topological constraint solver.

The DFOM allows a large variety of new features and topological constraints to be defined, which results in the required flexibility to model many useful families of objects.

For solving geometric constraints, we have introduced a new method based on rewriting systems of rigid and non-rigid clusters. With non-rigid clusters, we can solve a larger class of problems than is possible with only rigid clusters. Additional advantages of this approach over other solving approaches, such as DOF-based decomposition, are that we can solve large problems efficiently, and that an incremental solving algorithm is easy to implement.

Systems of constraints on primitives such as lines, planes, blocks, cylinders and spheres, which are typically found in CAD models and models of families of objects, including the DFOM, can be solved by first mapping them to systems of constraints on points, and solving the latter systems with the cluster rewriting approach.

We have also presented a mechanism to select a single solution for a system of geometric constraints on points, using a prototype configuration. The use of a prototype has been given a theoretical basis by introducing a formal resemblance relation. This allows us to select a solution in a meaningful way, i.e. the selected solution is really the intended one. The intended solution can be found by generating selection constraints for the subproblems solved by the constraint solver. With this approach, we can find the intended solution in polynomial time, making it suitable for large systems of constraints.

To solve systems of topological constraints, we map them to instances of the boolean satisfiability problem. The boolean satisfiability or SAT problem is NP-hard. Our experiments with the MINISAT solver, however, show that the types of SAT problems generated from DFOMs, even large DFOMs with many feature interactions, are easy to solve. It should be noted that for models with no realisations and models with more than one realisation, performance has not been systematically tested. However, at least for models with unique solutions, solving topological constraints is feasible.

Furthermore, a method has been presented to calculate the range of allowable values for a single parameter of a geometric constraint system. The method uses the decomposition of the constraint system to find the critical parameter values for which subproblems degenerate, and computes the range from these values. By solving for the intended solution, defined

by a prototype, the number of solutions generated and the complexity of the parameter range computation is reduced. The worst-case complexity of the method is, nevertheless, still exponential. However, we believe that such worst cases will not occur in practice, because parameters will typically affect the constraint system only locally.

Finally, we have presented a method to compute the critical values, for which the topology of the model changes, when a single parameter of a feature model is varied. The method can also be used to compute the parameter range corresponding to all valid models, i.e. the models for which all geometric and topological constraints in the model are satisfied. The computational complexity of the method is high, but we believe that the worst cases are not likely to occur in practice.

To summarise, declarative models for families of objects are feasible and desirable. Feasible because we have shown that solving systems of constraints in such models is possible. Desirable because such models can be used to specify feature semantics and create verifiable models of families of objects. Furthermore, for declarative models it is feasible to compute critical values corresponding to topological changes, and to compute parameter ranges corresponding to valid models. Knowing these properties can be very helpful when exploring the members of a family and analysing its behaviour.

8.2 Implementation and possible applications

The DFOM has been implemented in the prototype feature modelling system Spiff, developed at Delft University of Technology. This system originally implemented the Semantic Feature Model. Features and constraints defined by the system have been re-implemented so they can be used in the DFOM. Originally, a single realisation of the model was determined by feature dependency analysis. With the implementation of the DFOM, instead, a set of realisations is determined by solving the topological constraints in the model.

The geometric and topological solving methods have been implemented in independent modules, which are used by the implementation of the DFOM. The implementation of the method for geometric parameter range computation is based on the implementation of the geometric constraint solver. Methods for tracking topological changes have been implemented separately for simplified models, and are currently being implemented for the DFOM implementation in Spiff.

These methods can be applied to a broad range of applications that involve geometry and/or topology. The geometric constraint solver that we have presented is particularly useful in CAD, but systems of geometric constraints are found in many other applications, and we believe that here too our method may have significant advantages over other geometric constraint

solving methods.

Our topological constraint solving approach allows for a wide variety of topological constraints to be specified and solved, and may also be applicable in other modelling areas, e.g. in VLSI electronic circuit design, to specify which components should interact and which should not interact.

Calculating geometric parameter ranges may also be useful in other applications where geometric constraints are used to parameterise a model. Changing a parameter is a common operation in such applications, and knowing the allowable range of a parameter beforehand can prevent this from becoming a trial-and-error process.

The presented methods for tracking topological changes, i.e. the method to compute critical values for which the topology of a model changes and the method to compute the parameter range such that topological constraints are satisfied, were motivated by important problems in CAD, but may also be useful in other applications where topology matters, e.g. computed-aided manufacturing and robot motion planning. The methods do, however, require that geometry is specified declaratively, using geometric primitives and constraints, and that topological entities are determined by intersections of primitives.

8.3 Limitations and future research

In the DFOM, in our geometric constraint solving method and in the method for tracking topological changes, the carrier geometry considered is currently limited to planes, cylinders and spheres. For many real modelling applications, more general algebraic geometry and parametric geometry, such as NURBS, should also be considered.

The set of rewrite rules for 3D problems presented in Appendix A is not complete; there are known well-constrained geometric constructions that cannot be solved with the current rule set, e.g. the octahedral problem [Durand and Hoffmann, 2000]. It is not even known whether there exists a complete set of rewrite rules that can be used to solve every 3D system of rigid, scalable and radial clusters. This is related to the more fundamental problem whether a generic, combinatorial description can be given of rigidity in 3D [Sitharam, 2006].

It may be difficult to extend our prototype-based solution selection approach to general geometric constraint systems. In particular, rules for larger subproblems, e.g. the octahedral subproblem, should be found.

Currently, we have not yet done any extensive comparison of our solver to other solving algorithms. It would be interesting to see how it compares in terms of the class of problems that can be solved and in terms of algorithmic complexity.

The boolean constraint systems derived from systems of topological con-

straints can be very large. Although these systems are easy to solve for the MINISAT solver, generating these problems is expensive. A more efficient algorithm for mapping and solving topological constraints, would therefore be very useful. The MINISAT solver can be extended with new types of constraints, but the effectiveness of the solver's optimisation strategy for such constraints is unknown.

Some improvements on the performance of the geometric parameter range computation algorithm may be possible. For example, it may not be necessary to compute all geometric solutions for all degenerate cases, because not all critical values contribute to the actual intervals of the parameter range. For complex models, it would therefore be useful to look for optimisation strategies.

We believe that the basic approach to geometric parameter range computation, i.e. finding critical values by introducing degenerate cases of subproblems in the constraint system, can be applied to any decomposable system, but more general methods would be needed to determine degenerate cases for a wider range of subproblems.

The methods for geometric and topological parameter range computation consider only one variant parameter. The parameter space corresponding to valid models when two parameters, or perhaps three parameters, are varied simultaneously, may also be useful for a designer, since this range can be presented to a user graphically. Considering even more parameters simultaneously may be useful for automated model adjustment [Noort and Bronsvoort, 2001] and model optimisation algorithms though. Our method cannot be easily generalised to solve problems with several simultaneously varying parameters, but it may be useful to help reduce the search space of such problems.

The declarative modelling approach plays a central role in this thesis. We believe that declarative models, in particular models with both geometric and topological constraints, can be useful in many areas where complex design problems are found. Requirements can be directly specified in such models, and no arbitrary or implicit choices have to be made that are difficult to undo. Thus, semantics can be specified in declarative models, which makes it easy to verify, maintain and reuse such models. Finally, global properties of declarative models, e.g. parameter ranges and critical parameter values, can be derived to help users with complex modelling tasks.

Whether the declarative approach to modelling families of objects will be used in future commercial CAD systems, depends on whether CAD vendors, and users, are willing to move away from history-based models. One of the challenges for future CAD research is therefore to show that declarative modelling systems can be used to do everything that can be done with current, history-based modelling systems, and much more.

Appendix A

Rewrite rules for clusters

A collection of rewrite rules for clusters is presented here, which can be used for solving systems of geometric constraints, as discussed in Chapter 4.

Each rewrite rule consists of a pattern and a procedure. The pattern is of the form: $C_1 \cup C_2 \cup \ldots \cup C_{n-1} \rightarrow C_n$. Here C_1 to C_{n-1} represent input clusters and C_n represents the output cluster. Each cluster in the pattern specifies a type, i.e. *Rigid*, *Scalable* or *Radial*, and a set of variables. A cluster with a fixed number of variables may be specified, e.g. $Rigid([p_1, p_2])$ or a cluster with an unknown number of variables, using an ellipsis, e.g. $Rigid([p_1, p_2, \ldots])$. A radial cluster specifies one center variable and a set of radial variables, e.g. $Radial(p_1, [p_2, p_3])$. A set of variables of input clusters may also be given a name, e.g. $Rigid(A = [p_1, p_2, \ldots])$. Such named sets of variables may be combined in the specification of the output cluster, e.g. $Rigid(A) \cup Rigid(B) \rightarrow Rigid(A \cup B)$.

The procedure specified by a rule is a function $c_1 \times c_2 \times \ldots \times c_{n-1} \rightarrow c_n$. Here c_1 to c_{n-1} represent input configurations and c_n represents the ouput configuration. The number of configurations in the procedure is always equal to the number of clusters specified in the pattern. In a procedure, the union of two configurations, represented as $c_1 \cup c_2$, is a configuration containing all the point variables in c_1 and c_2. Point variables shared by c_1 and c_2 take the value specified by c_1.

We present a set of rules applicable only in 2D (Section A.1), a set of rules applicable in 2D and 3D (Section A.2), and a set of rules applicable only in 3D (Section A.3).

A.1 2D rewrite rules

Rule 1 *Merge two rigid clusters with two shared points*

Pattern: $Rigid(A = [p_1, p_2, \ldots]) \cup Rigid(B = [p_1, p_2, \ldots])$
$\rightarrow Rigid(A \cup B)$
Procedure: $c_1 \times c_2 \rightarrow c_R$
T = *rotation and translation such that* p_1 *and* p_2 *in* c_2 *are mapped onto* p_1 *and* p_2 *in* c_1
$c_R = c_1 \cup T(c_2)$
Degenerate cases:
$coincident(p_1, p_2)$

Rule 2 *Merge two radial clusters with two shared points*

Pattern: $Radial(p_x, A = [p_1, \ldots]) \cup Radial(p_x, B = [p_1, \ldots])]$
$\rightarrow Radial(p_x, A \cup B)$
Procedure: $c_1 \times c_2 \rightarrow c_R$
T = *rotation, translation and scaling, such that* p_x *and* p_1 *in* c_2 *are mapped onto* p_x *and* p_1 *in* c_1
$c_R = c_1 \cup T(c_2)$
Degenerate cases: none

Rule 3 *Merge two scalable clusters with two shared points*

Pattern: $Scalable(A = [p_1, p_2, \ldots]) \cup Scalable(B : [p_1, p_2, \ldots])$
$\rightarrow Scalable(A \cup B)$
Procedure: $c_1 \times c_2 \rightarrow c_R$
T = *rotation, translation and scaling such that* p_1 *and* p_2 *in* c_2 *are mapped onto* p_1 *and* p_2 *in* c_1
$c_R = c_1 \cup T(c_2)$
Degenerate cases: none

A.2 2D/3D rewrite rules

Rule 4 *Derive a triangle from three distances*

Pattern: $Rigid([p_1, p_2, \ldots]) \cup Rigid([p_1, p_3, \ldots]) \cup Rigid([p_2, p_3, \ldots])$
$\rightarrow Rigid([p_1, p_2, p_3])$
Procedure: $c_1 \times c_2 \times c_3 \rightarrow c_R$
$c_R(p_1) = c_1(p_1)$
$c_R(p_2) = c_1(p_2)$
$c_R(p_3) =$ *intersection*
circle centre $c_R(p_1)$ *radius* $\delta(c_2(p_3), c_2(p_1))$
circle centre $c_R(p_2)$ *radius* $\delta(c_3(p_3), c_3(p_2))$
Degenerate cases:
$colinear(p_1, p_2, p_3)$
$coincident(p_1, p_2)$
$coincident(p_1, p_3)$
$coincident(p_2, p_3)$

Rule 5 *Derive a triangle from two distances and an angle (by rotation)*

Pattern: $Rigid([p_1, p_2, \ldots]) \cup Radial(p_2, [p_1, p_3, \ldots]) \cup Rigid([p_2, p_3, \ldots])$
$\rightarrow Rigid([p_1, p_2, p_3])$
Procedure: $c_1 \times c_2 \times c_3 \rightarrow c_R$
$\phi = \measuredangle(c_2(p_1), c_2(p_2), c_2(p_3))$
$c_R(p_1) = (\delta(c_1[p_1], c_1[p_2]), 0)$
$c_R(p_2) = (0, 0)$
$c_R(p_3) = \delta(c_3[p_3], c_3[p_2])(cos(\phi), sin(\phi))$
Degenerate cases: none

Rule 6 *Derive a triangle from two distances and an angle (by intersection)*

Pattern: $Rigid([p_1, p_2, \ldots]) \cup Radial(p_2, [p_1, p_3, \ldots]) \cup Rigid([p_1, p_3, \ldots])$
$\rightarrow Rigid([p_1, p_2, p_3])$
Procedure: $c_1 \times c_2 \times c_3 \rightarrow c_R$
$c_R(p_1) = c_1(p_1)$
$c_R(p_2) = c_1(p_2)$
$c_R(p_3) = intersection$
circle centre $c_1(p_1)$ *radius* $\delta(c_3(p_3), c_3(p_1))$
ray from $c_1(p_2)$ *direction* $\measuredangle(c_2(p_1), c_2(p_2), c_2(p_3))$
Degenerate cases:
$\measuredangle(p_2, p_3, p_1) = \frac{1}{2}\pi$

Rule 7 *Derive a scalable cluster from two radial clusters*

Pattern: $Radial(p_1, [p_3, p_2, \ldots]) \cup Radial(p_2, [p_1, p_3, \ldots])$
$\rightarrow Scalable([p_1, p_2, p_3])$
Procedure: $c_1 \times c_2 \rightarrow c_R$
$c_R(p_1) = (0, 0, 0)$
$c_R(p_2) = (1, 0, 0)$
$c_R(p_3)$ = *intersection*
ray from $c_R(p_1)$ *direction* $\measuredangle(c_1(p_3), c_1(p_1), c_1(p_2))$
ray from $c_R(p_2)$ *direction* $\pi - \measuredangle(c_2(p_1), c_2(p_2), c_2(p_3))$
Degenerate cases:
$\measuredangle(p_3, p_1, p_2) = \pi - \measuredangle(p_1, p_2, p_3)$

Rule 8 *Derive a rigid cluster from a scalable and a rigid cluster with two shared points*

Pattern: $Scalable(A = [p_1, p_2, \ldots]) \cup Rigid([p_1, p_2, \ldots]) \rightarrow Rigid(A)$
Procedure: $c_1 \times c_2 \rightarrow c_R$
T = *scale configuration by* $\frac{\delta(c_2(p_2), c_2(p_1))}{\delta(c_1(p_2), c_1(p_1))}$
$c_R = T(c_1)$
Degenerate cases: none

A.3 3D rewrite rules

Rule 9 *Merge two rigid clusters with three shared points*

Pattern: $Rigid(A = [p_1, p_2, p_3, \ldots]) \cup Rigid(B = [p_1, p_2, p_3, \ldots])$
$\rightarrow Rigid(A \cup B)$
Procedure: $c_1 \times c_2 \rightarrow c_R$
T = *rotation-translation such that* p_1, p_2 *and* p_3 *in* c_2
are mapped onto p_1, p_2 *and* p_3 *in* c_1
$c_R = c_1 \cup T(c_2)$
Degenerate cases:
$colinear(p_1, p_2, p_3)$
$coincident(p_1, p_2)$
$coincident(p_1, p_3)$
$coincident(p_2, p_3)$
$coincident(p_1, p_2, p_3)$

Rule 10 *Merge two scalable clusters with three shared points*

Pattern: $Scalable(A = [p_1, p_2, p_3, \ldots]) \cup Scalable(B = [p_1, p_2, p_3, \ldots])$
$\rightarrow Scalable(A \cup B)$
Procedure: $c_1 \times c_2 \rightarrow c_R$
T = *rotation-translation-scaling such that* p_1, p_2 *and* p_3 *in* c_2 *are mapped onto* p_1, p_2 *and* p_3 *in* c_1
$c_R = c_1 \cup T(c_2)$
Degenerate cases:
$colinear(p_1, p_2, p_3)$

Rule 11 *Merge two radial clusters with three shared points*

Pattern: $Radial(p_x, A = [p_1, p_2, \ldots]) \cup Radial(p_x, B = [p_1, p_2, \ldots])$
$\rightarrow Radial(p_x, A \cup B)$
Procedure: $c_1 \times c_2 \rightarrow c_R$
T = *rotation-translation-scaling such that* p_x, p_1 *and* p_2 *in* c_2 *are mapped onto* p_x, p_1 *and* p_2 *in* c_1
$c_R = c_1 \cup T(c_2)$
Degenerate cases:
$colinear(p_1, p_2, p_3)$

Rule 12 *Derive a tetrahedron from three rigid clusters*

Pattern: $Rigid([p_1, p_2, p_3, \ldots]) \cup Rigid([p_1, p_2, p_4, \ldots]) \cup Rigid([p_3, p_4, \ldots])$
$\rightarrow Rigid([p_1, p_2, p_3, p_4])$
Procedure: $c_1 \times c_2 \times c_3 \rightarrow c_R$
$c_R(p_1) = c_1(p_1)$
$c_R(p_2) = c_1(p_2)$
$c_R(p_3) = c_1(p_3)$
$c_R(p_4) = intersection$
sphere centre $c_R(p_1)$ *radius* $\delta(c_2(p_4), c_2(p_1))$
sphere centre $c_R(p_2)$ *radius* $\delta(c_2(p_4), c_2(p_2))$
sphere centre $c_R(p_3)$ *radius* $\delta(c_3(p_4), c_3(p_3))$
Degenerate cases:

$coincident(p_1, p_2)$	$coplanar(p_1, p_2, p_3, p_4)$
$coincident(p_1, p_3)$	$colinear(p_1, p_2, p_3)$
$coincident(p_1, p_4)$	$colinear(p_1, p_2, p_4)$
$coincident(p_2, p_3)$	$colinear(p_1, p_3, p_4)$
$coincident(p_2, p_4)$	$colinear(p_2, p_3, p_4)$
$coincident(p_3, p_4)$	

Rule 13 *Derive a radial cluster from three angles*

Pattern: $Radial(p_x, [p_1, p_2, \ldots]) \cup Radial(p_x, [p_1, p_3, \ldots])$
$\cup Radial(p_x, [p_2, p_3, \ldots]) \rightarrow Radial(p_x, [p_1, p_2, p_3])$
Procedure: $c_1 \times c_2 \times c_3 \rightarrow c_R$
$c_R(p_x) = c_1(p_x)$
$c_R(p_1) = c_1(p_1)$
$c_R(p_2) = c_1(p_2)$
$c_R(p_3) = intersection$
cone apex $c_R(p_x)$ *axis* $c_R(p_1) - c_R(p_x)$ *angle* $\measuredangle(c_2(p_1), c_2(p_x), c_2(p_3))$
cone apex $c_R(p_x)$ *axis* $c_R(p_2) - c_R(p_x)$ *angle* $\measuredangle(c_3(p_2), c_3(p_x), c_3(p_3))$
sphere centre $c_R(p_x)$ *radius 1*
Degenerate cases:
$colinear(p_1, p_2, p_x)$
$colinear(p_1, p_3, p_x)$
$colinear(p_2, p_3, p_x)$
$coplanar(p_1, p_2, p_3, p_x)$

Rule 14 *Derive a rigid cluster from two rigid clusters with two shared points and an angle*

Pattern: $Rigid([p_1, p_2, p_3, p_4, \ldots]) \cup Rigid([p_3, p_4, p_5, \ldots])$
$\cup Radial(p_1, [p_2, p_5, \ldots]) \rightarrow Rigid([p_1, p_2, p_3, p_4, p_5])$
Procedure: $c_1 \times c_2 \times c_3 \rightarrow c_R$
$c_R(p_1) = c_1(p_1)$
$c_R(p_2) = c_1(p_2)$
$c_R(p_3) = c_1(p_3)$
$c_R(p_4) = c_1(p_4)$
$c_R(p_5) = intersection$
cone apex $c_R(p_1)$ *axis* $c_R(p_2) - c_R(p_1)$ *angle* $\measuredangle(c_3(p_5), c_3(p_1), c_3(p_2))$
cylinder axis $c_R(p_4) - c_R(p_3)$ *radius* $distance(c_2(p_5), line(c_2(p_4), c_2(p_3)))$
plane normal $c_R(p_4) - c_R(p_3)$
through $c_R(p_3) + (c_2(p_5) - c_2(p_3)) \cdot (c_2(p_4) - c_2(p_3))$
Degenerate cases:
$parallel(line(p_1, p_5), line(p_3, p_4))$
$colinear(p_1, p_2, p_5)$
$colinear(p_3, p_4, p_5)$
$coincident(p_1, p_2)$
$coincident(p_3, p_4)$

Appendix B

Selection criteria for the intended solution

In this appendix, we will show that the intended solution of a geometric constraint problem, as discussed in Section 4.5, can be found by generating appropriate selection criteria for its subproblems.

The constraint problem considered here is a system of distance and angle constraints on points in 3D, similar to the problem considered in Chapter 4. However, the solving approach used here is a simpler one, which does not determine non-rigid clusters, but only rigid clusters, corresponding to triangular and tetrahedral subproblems.

First, in Section B.1, we formally define the intended solution. Next, in Section B.2, we show that for any triangular or tetrahedral subproblem, we can select the intended solution using simple selection criteria. Finally, we show in Section B.3 that by combining intended solutions of subproblems, we find the intended solution for the whole problem.

Parts of this appendix have already been published, see [van der Meiden and Bronsvoort, 2005b].

B.1 Definition of the intended solution

We consider a geometric constraint problem with distance constraints and angle constraints on points in 3-dimensional Euclidean space.

Definition 1 *A geometric constraint problem $G(V, C, x)$ consists of n point variables $V = \{v_1 \dots v_n\}$ and m constraints $C = \{c_1 \dots c_m\}$. A constraint $c_i \in C$ is either a distance constraint or an angle constraint with parameter $x_i \in \mathbb{R}$, $x = (x_1, ...x_m)$.*

The parameters of the problem are represented by the parameter vector $x = (x_1, ...x_m) \in X$, where $X = \mathbb{R}^m$ is referred to as the *parameter space* of the problem.

A solution for the problem is a configuration of points, i.e. a vector $y = (y_1, ..., y_n)$, $y_i \in \mathbb{R}^3$. The space $Y = \mathbb{R}^{n\times 3}$ is referred to as the *configuration space* of the problem.

Definition 2 *The generic solution $S(x)$ for a geometric constraint problem $G(V, C, x)$ is a set of solutions as a function of the parameters of the problem, such that $S(x) = \{y; y \therefore G(V, C, x)\}$. Here $y \therefore G(V, C, x)$ means that y satisfies the constraints in $G(V, C, x)$.*

Definition 3 *The intended solution for a problem $G(V, C, x)$ and a prototype p is a function $s : X \times Y \rightarrow Y$ such that: (1) $s(x, p) \in S(x)$ and (2) $s(x, p) \equiv_G p$, where $\equiv_G \subseteq Y \times Y$ is called the resemblance relation for G.*

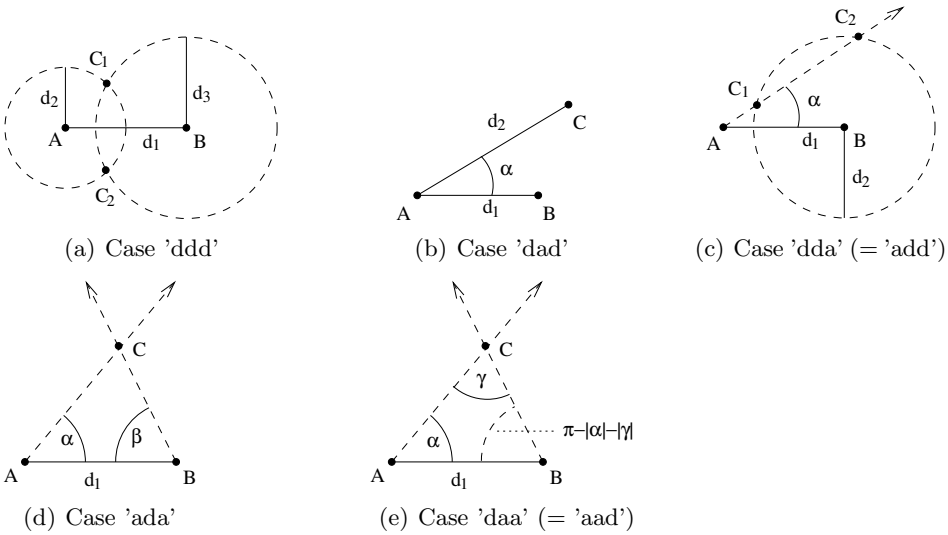
The resemblance relation defines when a solution resembles a given prototype configuration, and is thus the intended solution. The resemblance relation is an equivalence relation, which partitions the configuration space into a number of equivalence classes. The intended solution is then, by Definition 3, the solution in the same equivalence class as the prototype. The equivalence relation must be defined such that the following properties hold:

Property 1 *$x \rightarrow s(x, p)$ is a continuous mapping from X' to Y, for any $p \in Y$, where $X' = \{x^* \in X : s(x^*, p)$ exists $\}$.*

Property 2 *Different solutions for the same parameter value do not resemble each other, i.e. they are in different equivalence classes. Formally: $\forall x \in X, \forall s_1, s_2 \in S(x) : s_1 \neq s_2 \Rightarrow s_1 \not\equiv_G s_2$.*

Property 3 *Every equivalence class is a connected set, i.e. for any two configurations $y_1, y_2 \in Y$ such that $y_1 \equiv_G y_2$, there exists a continuous function $f : [0, 1] \rightarrow Y$ such that $f(0) = y_1$, $f(1) = y_2$ and $\forall \phi \in (0, 1) : f(\phi) \equiv_G y_1$ (and $f(\phi) \equiv_G y_2$).*

In words, Property 1 states that there is a continuous relation between the parameters of the constraints and the intended solution. Property 2 ensures that a solution can be selected unambiguously; given any combination of a prototype and a parameter vector, there is at most one solution. Property 3 states that all configurations in an equivalence class are connected; configurations that resemble each other are geometrically 'close' to each other. We believe this to be a reasonable interpretation of the intuitive meaning of resemblance.



(a) Case 'ddd' (b) Case 'dad' (c) Case 'dda' (= 'add')

(d) Case 'ada' (e) Case 'daa' (= 'aad')

Figure B.1: Triangular subproblems

B.2 Subproblem analysis

In this section, we consider how the intended solution can be determined for basic triangular and tetrahedral subproblems. In Section B.3 we consider the construction of the solutions of the whole problem by cluster merging.

There are five well-constrained triangular subproblems to consider, illustrated in Figure B.1. Each case is identified by a 3-letter code, where 'd' stands for a known distance, and 'a' stands for a known angle. The order of these letters indicates the configuration of the known distances and angles. Because three points are always in a plane, these problems are solved in 2D.

Case 'ddd': three distances are known, see Figure B.1(a).
constraints: $d(A,B) = d_1$, $d(A,C) = d_2$, $d(B,C) = d_3$
solution: $A = (0,0)$, $B = (d_1, 0)$, $C = (x, y)$
$x = \frac{(d_2)^2 - (d_3)^2 + (d_1)^2}{2d_1}$, $y = \pm\sqrt{(d_2)^2 - x^2}$

Case 'dad': two distances and the enclosed angle are known, see Figure B.1(b).
constraints: $d(A,B) = d_1$, $a(B,A,C) = \alpha$, $d(A,C) = d_2$
solution: $A = (0,0)$, $B = (d_1, 0)$, $C = (x, y)$
$x = d_2 \cdot cos(\alpha)$, $y = \pm d_2 \cdot sin(\alpha)$

Case 'dda' (= 'add'): two distances and one adjacent angle are known, see Figure B.1(c).
constraints: $d(A,B) = d_1$, $d(B,C) = d_2$, $a(B,A,C) = \alpha$
solution: $A = (0,0)$, $B = (d_1, 0)$, $C = (x,y)$
$x = t \cdot cos(\alpha)$, $y = \pm t \cdot sin(\alpha)$
$t = q \pm \sqrt{4q^2 - 4r}, \quad t \geq 0$
$q = d_1 \cdot cos(\alpha), \quad r = (d_1)^2 - (d_2)^2$

Case 'ada': one distance and two adjacent angles are known, see Figure B.1(d).
constraints: $a(C,A,B) = \alpha$, $d(A,B) = d_1$, $a(A,B,C) = \beta$
solution: $A = (0,0)$, $B = (d_1, 0)$, $C = (x,y)$
$x = t \cdot cos(\alpha)$, $y = \pm t \cdot sin(\alpha)$
$t = \frac{d_1 \cdot cos(\beta)}{sin(\alpha) \cdot cos(\beta) + sin(\beta) \cdot cos(\alpha)}$

Case 'daa' (= 'aad'): one distance, one adjacent angle and the opposite angle are known, see Figure B.1(e).
constraints: $a(C,A,B) = \alpha$, $a(B,C,A) = \gamma$, $d(A,B) = d_1$
solution: $A = (0,0)$, $B = (d_1, 0)$, $C = (x,y)$
$x = t \cdot cos(\alpha)$, $y = \pm t \cdot sin(\alpha)$
$t = \frac{d_1 \cdot cos(\beta)}{sin(\alpha) \cdot cos(\beta) + sin(\beta) \cdot cos(\alpha)}, \quad \beta = \pi - |\alpha| - |\gamma|$

For each subproblem, the intended solution is represented by a function $s(x,p)$, where x is the parameter vector containing only the parameters of the constraints in the subproblem, and p is the subconfiguration of the prototype for the point variables involved in the subproblem. The prototype is used as a solution selector in this function, i.e. depending on p, a particular solution is returned. The selected solution should have a resemblance relation with the prototype (see Definition 3), so that it is the intended solution.

For each subproblem case, a specific resemblance relation will be defined in this section. For the 'ddd', 'dad', 'ada' and 'daa' cases, the same resemblance relation is used, denoted by $\equiv^*$. For the 'dda' case, the resemblance relation is denoted by $\equiv^{dda}$, and for the tetrahedral subproblem by $\equiv^{tet}$. The implementation of the different functions $s(x,p)$ is trivial given these resemblance relations. For our analysis it is sufficient to note that, depending on the specific subproblem case, $s(x,p) \equiv^* p$, $s(x,p) \equiv^{dda} p$ or $s(x,p) \equiv^{tet} p$.

The intended solution and resemblance relation for each subproblem must satisfy the properties defined in Section B.1. In the following, we will

define the resemblance relations and prove the required properties in each case.

First note that for each of the triangle cases described above, there are mirror-symmetrical solutions, i.e. solutions for $y > 0$ and $y < 0$.

Lemma 1 *Mirror-symmetrical solutions in 2D are congruent in 3D.*

Proof: The plane in which these solutions are constructed is arbitrary. In 3D, the solutions may be rotated 180° about the symmetry axis, so they are congruent.

Lemma 2 *For the 'ddd', 'dad', 'ada' and 'daa' cases, $x \to s(x,p)$ is a continuous mapping, satisfying Property 1.*

Proof: For each case, there are only two solutions, which are congruent in 3D (Lemma 1). In the solution formulas given above, the sign of y may be chosen arbitrarily. In each case, the solution is a continuous function of the constraint parameters, and thus satisfies Property 1.
Due to Lemma 1, for all the trianguluar cases except the 'dda' case, there is only one solution. The definition of the resemblance relation for these cases is therefore trivial:

Definition 4 *For the 'ddd', 'dad', 'ada' and 'daa' cases, the resemblance relation $\equiv^* \subseteq Y \times Y$ is defined by:*

$$y_1 \equiv^* y_2 \iff y_1, y_2 \in Y$$

This relation is obviously reflexive ($a \equiv^* a$), symmetrical ($a \equiv^* b \iff b \equiv^* a$) and transitive ($a \equiv^* b$ and $b \equiv^* c \Rightarrow a \equiv^* c$). Thus it is an equivalence relation.

Lemma 3 *The resemblance relation $\equiv^*$ satisfies Properties 2 and 3.*

Proof: For any given parameter value, there is at most one solution, thus Property 2 is satisfied. The $\equiv^*$ relation defines only one equivalence class, equal to $Y = \mathbb{R}^{3n}$, which is connected. Thus Property 3 is also satisfied.

In the 'dda' case, there may be up to four solutions in 2D. One pair of solutions may be discarded due to symmetry (Lemma 1). From the remaining pair, one solution is selected using the prototype configuration. Given a 'dda' problem on a triangle ABC, as shown in Figure B.1(c), and a prototype $p = (A_p, B_p, C_p)$, then the solution $s = (A_s, B_s, C_s)$ is selected such that angle $\angle B_p C_p A_p$ and angle $\angle B_s C_s A_s$ are either both acute or both non-acute. The *acuteness* Γ of an angle $\angle Q$ is defined as:

$$\Gamma(\angle Q) = \begin{cases} \textit{acute} & \iff & 0 \le \angle Q < \frac{\pi}{2} \\ \textit{non-acute} & \iff & \frac{\pi}{2} \le \angle Q \le \pi \end{cases}$$

Definition 5 *For the 'dda' case, the resemblance relation* $\equiv^{dda} \subseteq Y \times Y$ *is defined by:*

$$ABC \equiv^{dda} A'B'C' \iff \Gamma(\angle BCA) = \Gamma(\angle B'C'A')$$

Because equality is an equivalence relation, $\equiv^{dda}$ is also an equivalence relation.

Lemma 4 *For the 'dda' case,* $x \to s(x, p)$ *is a continuous mapping, satisfying Property 1.*

Proof: The sign of the term $t = q \pm \sqrt{4q^2 - 4r}$ in the general 'dda' solution, presented at the beginning of this section, is determined only by the prototype p. Both solutions are continuous and real for $4q^2 > 4r$. By substitutions for q and r, we obtain:

$$cos^2(\alpha) > 1 - (\frac{d_2}{d_1})^2$$

By substituting $cos^2(\alpha) + sin^2(\alpha) = 1$, we obtain:

$$sin^2(\alpha) < (\frac{d_2}{d_1})^2$$

The parameter domain corresponds to:

$$(|\alpha| < sin^{-1}(\frac{d_2}{d_1}) \cap d_1 \geq d_2) \cup (d_1 < d_2)$$

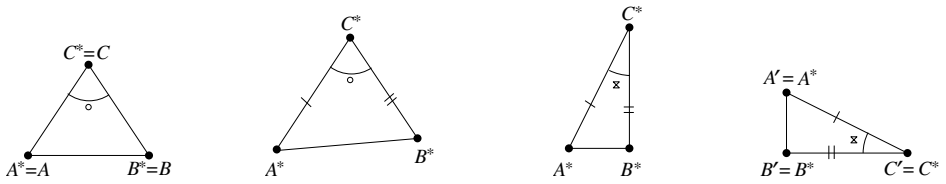
For any parameter vector in this domain, Property 1 is satisfied.

Lemma 5 *The resemblance relation* $\equiv^{dda}$ *satisfies Property 2.*

Proof: There are two distinct solutions of the 'dda' case, for the same parameter values, corresponding to $t = q - \sqrt{4q^2 - 4r}$ and $t = q + \sqrt{4q^2 - 4r}$. Given that ABC is a solution corresponding to $0 \leq t \leq q$, then $\|AC\| = t \leq q = d_1 cos(\alpha)$ and $\angle BCA \geq \frac{\pi}{2}$. Note that if $t = q$, then $\|AC\| = t = q = d_1 cos(\alpha)$, and thus $\angle BCA = \frac{\pi}{2}$. In this case there is only one solution, and no selection is needed.

The other solution, corresponding to $t > q$ is a triangle $A'B'C'$, where $\angle B'C'A' < \frac{\pi}{2}$. Angle $\angle B'C'A'$ is always acute, whereas $\angle BCA$ is always non-acute, i.e. $ABC \not\equiv^{dda} A'B'C'$. Thus, Property 2 is satisfied.

Lemma 6 *The resemblance relation* $\equiv^{dda}$ *satisfies Property 3.*



(a) $f(0)$ (b) $f(\frac{1}{3})$ (c) $f(\frac{2}{3})$ (d) $f(1)$

Figure B.2: A continuous acuteness-preserving transformation

Proof: Given a configuration ABC (Figure B.2a) such that $\angle BCA$ is acute and a configuration $A'B'C'$ (Figure B.2d) such that $\angle B'C'A'$ is also acute. Then there is a continuous function f, such that $f(0) = ABC$, $f(1) = A'B'C'$, and $\forall \phi \in (0,1)$: $f(\phi) = A^*B^*C^*$ such that $\angle B^*C^*A^*$ is acute.

This function may be constructed as follows. From $\phi = 0$ to $\phi = \frac{1}{3}$, the edges A^*C^* and B^*C^* are scaled continuously from $\|AC\|$ to $\|A'C'\|$ and from $\|BC\|$ to $\|B'C'\|$. The angle $\angle B^*C^*A^*$ remains constant (Figure B.2b). Then, angle $\angle B^*C^*A^*$ is scaled continuously from $\angle BCA$ at $\phi = \frac{1}{3}$ to $\angle B'C'A'$ at $\phi = \frac{2}{3}$. The edges A^*C^* and B^*C^* remain of constant length, but the edge A^*B^* is now scaled (Figure B.2c). $\angle B^*C^*A^*$ remains acute during this transformation, because $\angle BCA < \frac{\pi}{2}$ and $\angle B'C'A' < \frac{\pi}{2}$. $A^*B^*C^*$ is now congruent with $A'B'C'$. From $\frac{2}{3} < \phi < 1$, the triangle $A^*B^*C^*$ undergoes a rigid motion such that at $\phi = 1$ it is equal to $A'B'C'$ (Figure B.2d). $\angle B^*C^*A^*$ remains constant during this motion. Thus for any $\phi \in [0,1]$: $\angle B^*C^*A^*$ is acute. The equivalence class corresponding to an acute angle is thus connected, satisfying Property 3. For $\angle BCA$ and $\angle B'C'A'$ being non-acute, a similar proof can be constructed.

The basic 3D subproblem is the tetrahedral subproblem. This problem involves four points. Only the case where six distances between these points are known needs to be considered. Angle constraints are solved in triangular subproblems.

A tetrahedron $ABCD$ is constructed using six known distances: d_{AB}, d_{AC}, d_{AD}, d_{BC}, d_{BD} and d_{CD}. First, a triangle ABC is constructed. Point D is determined by the intersection of three spheres, centred in A, B and C, with radii d_{AD}, d_{BD} and d_{CD} respectively.

There may be zero, one or two solutions for the intersection. If there are two solutions, then these are symmetrical, mirrored in the plane through ABC (see Figure B.3). To distinguish these solutions, we determine the *handedness* Θ of the corresponding tetrahedra. For a tetrahedron $PQRS$ this is defined as:

$$\Theta(PQRS) = \begin{cases} right & \iff & (\overrightarrow{PQ} \times \overrightarrow{PR}) \cdot \overrightarrow{PS} \geq 0 \\ left & \iff & (\overrightarrow{PQ} \times \overrightarrow{PR}) \cdot \overrightarrow{PS} < 0 \end{cases}$$

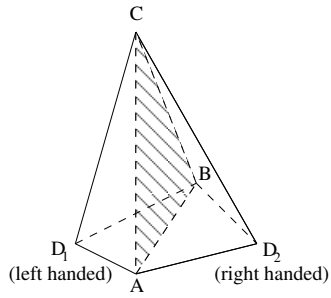


Figure B.3: Lefthanded and righthanded solutions $ABCD_1$ and $ABCD_2$, given d_{AB}, d_{AC}, d_{BC}, d_{AD}, d_{BD} and d_{CD}.

Definition 6 *For the tetrahedral subproblem, the resemblance relation* $\equiv^{tet} \subseteq Y \times Y$ *is defined by:*

$$ABCD \equiv^{tet} A'B'C'D' \iff \Theta(ABCD) = \Theta(A'B'C'D')$$

Because equality is an equivalence relation, $\equiv^{tet}$ is also an equivalence relation.

Lemma 7 *For tetrahedral subproblems,* $x \to s(x,p)$ *is a continuous mapping, satisfying Property 1.*

Proof: For any combination of the parameters, there are two symmetrical solutions, there are no solutions, or there is one degenerate solution. In the latter two cases, no solution selection is needed. If there are two solutions, then one solution is lefthanded and the other is righthanded, due to symmetry. The solution of which the handedness is equal to the handedness of the prototype, is the intended solution. Thus, the parameter domain for which the intended solution is continuous, is equal to the parameter domain for which a solution exists.

This domain can be characterised as follows. For each triangle in the tetrahedron, a solution must exist, which is expressed by the triangle inequality. Each parameter corresponds to an edge in two triangles, thus for each parameter two inequalities can be formulated:

$$d_{pq} < d_{pr} + d_{qr}$$

$$d_{pq} < d_{ps} + d_{qs}$$

where p, q, r and s should be replaced by any combination of A, B, C and D, resulting in a total of twelve inequalities. Each of these inequalities can be described geometrically as a half-space $X_i \in X$, $i = 1 \ldots 12$, where X is the parameter space of the problem. The domain of the parameters for which a solution exists is $X' = \bigcap X_i$. It can easily be shown, using elementary

calculus, that this is a continuous domain. Thus, for any parameter vector $x \in X'$, the solution is a continuous mapping, satisfying Property 1.

Lemma 8 *The resemblance relation $\equiv^{tet}$ satisfies Property 2.*

Proof: The two solutions for a tetrahedron $ABCD$ are mirror-symmetrical. If the triangle ABC is first constructed in the plane $z = 0$, then the solutions D_1 and D_2 for point D are mirror images on different sides of this plane. Suppose $ABCD_1$ is righthanded, then $(\overrightarrow{AB} \times \overrightarrow{AC}) \cdot \overrightarrow{AD_1} > 0$, and $(\overrightarrow{AB} \times \overrightarrow{AC}) \cdot \overrightarrow{AD_2} < 0$, and thus $ABCD_2$ is lefthanded. If $ABCD_1$ is lefthanded, then vice versa. In both cases $\Theta(ABCD_1) \neq \Theta(ABCD_2)$ and $ABCD_1 \not\equiv^{tet} ABCD_2$, thus Property 2 is satisfied.

Lemma 9 *The resemblance relation $\equiv^{tet}$ satisfies Property 3.*

Proof: Each equivalence class in $\equiv^{tet}$ should be a connected set, i.e. all righthanded tetrahedra should be connected and all lefthanded tetrahedra should be connected. Given two righthanded configurations $ABCD$ and $A'B'C'D'$. Then there is a function f such that $f(0) = ABCD$, $f(1) = A'B'C'D'$, and $\forall \phi \in (0,1)$: $f(\phi) = A^*B^*C^*D^*$, such that $A^*B^*C^*D^*$ is righthanded. Such a function may be constructed as follows. At $\phi = 0$, the edges of $A^*B^*C^*D^*$ are equal to the corresponding edges in $ABCD$. The edges are then scaled continuously such that they have the same length as the corresponding edges in $A'B'C'D'$ at $\phi = \frac{1}{2}$. To scale each edge, a scaling/shearing transformation is applied to the tetrahedron, which never scales an edge negatively, and thus the handedness of the tetrahedron does not change. From $\phi = \frac{1}{2}$ to $\phi = 1$, a rigid rotation/translation transforms $A^*B^*C^*D^*$ such that it exactly matches $A'B'C'D'$. This also does not change the handedness of the tetrahedron. Thus, we have shown that all righthanded tetrahedra are connected. For lefthanded tetrahedra, the same function may be used to proof the lemma.

B.3 Construction analysis

The intended solution for a problem $G(V, C, x)$ is obtained by solving a sequence of subproblems $\mathcal{H} = \{H_1, \ldots H_k\}$, and merging the intended solutions of the subproblem.

For each $H_i \in \mathcal{H}$ we have $H_i = (V_i, C_i)$, where $V_i \subset V$, and the constraints in C_i either are in C, or result from previously solved subproblems. The parameter vector of a subproblem H_i is denoted x_i and consists of the parameters of the constraints C_i. The prototype p_i is a subconfiguration of the prototype p, corresponding to the variables in V_i. The intended solution for each subproblem H_i is represented by a vector s_i.

Subproblems are solved in a specific order, such that for $i < j$, H_i is independent of H_j. If H_j depends on H_i, then they share two point variables,

of which the relative position is determined by H_i, and the distance between these points is needed to solve H_j.

Lemma 10 *Given is that $H_i = (C_i, V_i)$ and $H_j = (C_j, V_j)$ such that $V_i \cap V_j = \{v_a, v_b\}$. Also given is that in V_i we find $|v_a - v_b| = d_{ab}$ and in C_j there is a constraint $d(v_a, v_b, d_{ab})$. If $x_i \rightarrow s_i$ satisfies Property 1 and $x_j \rightarrow s_j$ satisfies Property 1, then $x_i \rightarrow s_j$ satisfies Property 1.*

Proof: If $x_i \rightarrow s_i$ is continuous, then $x_i \rightarrow d_{ab}$ is also continuous. Since $d_{ab} \in x_j$ and $x_j \rightarrow s_j$ is continuous, $x_i \rightarrow s_j$ is a composition of continuous functions, and thus also a continuous function.

Given a subproblem $H_i = (V_i, C_i)$ and a solution s_i, where $V_i = \{v_{i1}, \ldots, v_{ik}\}$ and $s_i = (s_{i1}, \ldots, s_{ik})$, then a cluster $K_i = \{(v_{il}, s_{il}); l = 1 \ldots k\}$ can be constructed. The cluster represents a collection of (variable, value) pairs, corresponding to assignments that satisfy the constraints in the problem.

A pair of clusters K_i and K_j can be merged if $V_i \cap V_j = \{v_a, v_b, v_c\}$, and the configuration of these points in both clusters is the same. If the three points are on a line or in a single point, then the problem is underconstrained. Otherwise, a new cluster $K_m = K_i \otimes K_j$ is obtained. Here $K_m = \{(v, s^*); v \in V_i \cup V_j, s^* \in s_i \cup T(s_j)\}$. For each pair $(v, s) \in K_i$, there is a corresponding pair $(v, s) \in K_m$. For each pair $(v, s) \in K_j$ and $(v, s) \notin K_i$, there is a pair $(v, T(s)) \in K_m$. The transformation T is a rotation/translation such that the configuration of v_a, v_b and v_c in K_j is mapped onto the configuration of these variables in K_i. Thus, if (v_a, s_{ia}), (v_b, s_{ib}), $(v_c, s_{ic}) \in K_i$ and (v_a, s_{ja}), (v_b, s_{jb}), $(v_c, s_{jc}) \in K_j$, then $T(s_{ja}) = s_{ia}$, $T(s_{jb}) = s_{ib}$ and $T(s_{jc}) = s_{ic}$.

Lemma 11 *Let $K_m = K_i \otimes K_j$. If $x_i \rightarrow s_i$ and $x_j \rightarrow s_j$ are continuous, then $x_i \rightarrow s_m$ and $x_j \rightarrow s_m$ are also continuous.*

Proof: For each pair $(v, s) \in K_i$, there is a corresponding pair $(v, s) \in K_m$. Since $x_i \rightarrow s_i$ is continuous, $x_i \rightarrow s_m$ is also continuous. For each pair $(v, s) \in K_j$ and $(v, s) \notin K_i$, there is a pair $(v, T(s)) \in K_m$. Since $x_j \rightarrow s_j$ is continuous, and T is a continuous transformation, $x_j \rightarrow s_m$ is also continuous.

Theorem 1 *The intended solution $s(x, p)$ satisfies Property 1.*

Proof: $s(x, p)$ is obtained by solving a sequence of subproblems, by propagating distances, and by merging clusters. From Lemmas 2, 4 and 7, we infer that all subproblem solutions satisfy Property 1. Using Lemma 10 and Lemma 11, we infer that the final solution also satisfies Property 1.

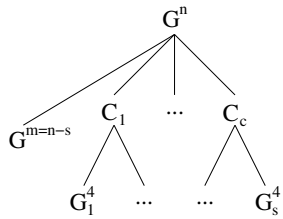


Figure B.4: Relations between subproblems and clusters used in proof of Theorem 2.

Definition 3 states that $s(x,p) \equiv_G p$, where $s(x,p)$ is the intended solution for a problem G. We can now define the resemblance relation $\equiv_G$ in terms of the resemblance relations $\equiv_i$ of the subproblems H_i that are solved to solve G.

Definition 7 *Given a problem G and a set of subproblems $\mathcal{H} = \{H_1, ..., H_k\}$ then $\equiv_G$ is defined as:*

$$y \equiv_G z \iff \forall i \in 1 \ldots k : y_i \equiv_i z_i$$

Here, y_i and z_i are the subconfigurations of y and z, for the point variables V_i of each subproblem $H_i \in \mathcal{H}$, and each relation $\equiv_i$ is appropriately chosen as $\equiv^*$, $\equiv^{dda}$ or $\equiv^{tet}$, defined in Definitions 4, 5 and 6 respectively.

Theorem 2 *The resemblance relation $\equiv_G$ is uniquely defined by G.*

Proof: First, consider problems with no angle constraints. A problem G^n consisting of n points and $k = 3n-6$ distance constraints is well-constrained and can be solved by decomposing it into $t = n-3$ tetrahedral subproblems (to see this, consider that for four points, six distances are needed to solve one tetrahedron, and that for each extra point, three more constraints and one more tetrahedron are needed).

We construct the proof by induction. First, if $n = 4$, then $k = 6$ and $t = 1$, and the theorem obviously holds: G^4 is the tetrahedral subproblem. Next, assume that for any $4 \leq m < n$ the set $\mathcal{H}^m$ of subproblems used for solving G^m is uniquely determined by G^m.

Suppose the G^n problem contains s independent tetrahedral subproblems $G_i^4 \subset G^n$, where $1 \leq i \leq s$ and $s \leq t$. Two subproblems are independent if they can be solved in any order. The independent subproblems are completely determined by the system of constraints in G^n. Suppose also that the solutions of the G^4 subproblems can be merged into c rigid clusters $C_1 \ldots C_c$. Figure B.4 shows the relations between subproblems and clusters used in this proof. Because the problem is well constrained, $s \geq 1$ and $c \geq 1$. If $s = t$, then all tetrahedral solutions can be merged into one

cluster, and the problem is solved. If $s < t$, then $c > 1$, and we construct a new problem that shares three points with each cluster, so all clusters can be merged into a single cluster after this new problem has been solved. This problem is constructed as follows. Every constraint that is not in any of the G_i^4 subproblems, and the points those constraint are imposed on, are included in the new problem. Any distance, between a pair of points in the new problem that is determined by one of the clusters, is added as a constraint to the new problem. The new problem is thus completely determined by G^n. Each G_i^4 subproblem fixes one point relative to the new problem, because it shares three points with it, or because it shares three points with a cluster that shares three points with the new problem. The new subproblem thus contains $m = n - s$ points, and because $s > 0$, $m < n$.

Because the G^n problem is well-constrained, the new problem is a well-constrained G^m problem, with $m < n$, for which we have assumed that $\mathcal{H}^m$ is uniquely determined by G^m. As already stated above, the independent G^4 subproblems are also uniquely determined by G^n, thus $\mathcal{H}^n = \mathcal{H}^m + \{G_i^4; 1 \leq i \leq s\}$ is uniquely determined by G^n.

Now consider angle constraints; these are solved in triangular subproblems. The distances determined by these subproblems are then used to solve tetrahedral subproblems. The configuration of angle constraints in the problem G determines which triangular patterns (i.e. dda, dad, daa or ada) are used. A triangular subproblem can be solved directly using the constraints in G, or a triangular subproblem can be solved using distances determined by merging clusters. In both cases, the configuration of the angles in the problem determines the pattern used.

We therefore conclude that $\mathcal{H}$ is uniquely determined by the constraints in G. From Definition 7 it is clear that the resemblance relation $\equiv_G$ depends only on the set of subproblems in $\mathcal{H}$, and not on the order in which subproblems are solved, so $\equiv_G$ is uniquely determined by G.

Lemma 12 *The resemblance relation $\equiv_G$ is an equivalence relation.*

Proof: For two configurations y and z, if $y = z$, then for each subproblem H_i, $y_i = z_i$. From Definitions 4, 5 and 6, we infer that each $\equiv_i$ is a resemblance relation, and therefore $y_i \equiv_i z_i$. From Definition 7 follows $y \equiv_G z$, thus $\equiv_G$ is reflexive. Also, for each subproblem, if $y_i \equiv_i z_i$ then $z_i \equiv_i y_i$, therefore, if $y \equiv_G z$ then $z \equiv_G y$. Thus, $\equiv_G$ is symmetrical. Given three configurations y, z and r such that $y \equiv_G z$ and $z \equiv_G r$, then, for every subproblem, $y_i \equiv_i z_i$ and $z_i \equiv_i r_i$. Because each $\equiv_i$ is transitive, we also have $y_i \equiv_i r_i$. From Definition 7 follows $y \equiv_G r$, thus the relation $\equiv_G$ is transitive. The relation $\equiv_G$ is thus reflexive, symmetrical and transitive, i.e. an equivalence relation.

Theorem 3 *The resemblance relation $\equiv_G$ satisfies Property 2.*

Proof: Given two solutions $s \therefore G$ and $t \therefore G$, for the same parameter vector, and a decomposition $\mathcal{H}$ of G. The first solution, s, is constructed by merging solutions $s_i \therefore H_i$, and t is constructed by merging solutions $t_i \therefore H_i$. If $s \neq t$ then there must be a subproblem H_i for which $s_i \neq t_i$, because merging clusters involves a rotation/translation transformation, which preserves the relative positions of the points in each cluster. From Lemmas 3, 5 or 8, we obtain $s_i \not\equiv_i t_i$. From Definition 7, we obtain $s \not\equiv_G t$, and thus $\equiv_G$ satisfies Property 2.

Theorem 4 *The resemblance relation $\equiv_G$ satisfies Property 3.*

Proof: Given two configurations $y, z \in Y$, $y \equiv_G z$, then by Definition 7, for every subproblem $H_i \in \mathcal{H}$, $y_i \equiv_i z_i$. According to Lemmas 3, 6 and 9, all $\equiv_i$ satisfy Property 3, i.e. the equivalence classes in these relations are connected, and thus each pair y_i, z_i is connected. Then y and z are also connected. Thus the relation $\equiv_G$ satisfies Property 3.

The solution found thus satisfies all three properties given in Section B.1, and is therefore the intended solution.

Bibliography

Achlioptas, D. and Peres, Y. (2003). The threshold for random k-sat is 2k (ln 2 - o(k)). In *STOC '03: Proceedings of the Thirty-fifth Annual ACM Symposium on Theory of Computing, June 9–11, San Diego, California*, pages 223–231. ACM Press, New York, NY, USA.

Bendsoe, M. P. (1989). Optimal shape design as a material distribution problem. *Structural Optimization*, 1(1):193–202.

Bettig, B. and Shah, J. (2003). Solution selectors: a user-oriented answer to the multiple solution problem in constraint solving. *Journal of Mechanical Design*, 125(3):443–451.

Bidarra, R. (1999). *Validity Maintenance in Semantic Feature Modeling*. PhD thesis, Delft University of Technology.

Bidarra, R. and Bronsvoort, W. F. (2000a). On families of objects and their semantics. In *Proceedings Geometric Modeling and Processing Conference, April 10–12, Hong Kong, China*, pages 101–111. IEEE Press, USA.

Bidarra, R. and Bronsvoort, W. F. (2000b). Semantic feature modelling. *Computer-Aided Design*, 32(3):201–225.

Bidarra, R., de Kraker, K. J., and Bronsvoort, W. F. (1998). Representation and management of feature information in a cellular model. *Computer-Aided Design*, 30(4):301–313.

Bidarra, R., Madeira, J., Neels, W., and Bronsvoort, W. F. (2005a). Efficiency of boundary evaluation for a cellular model. *Computer-Aided Design*, 37(12):1266–1284.

Bidarra, R., Nyirenda, P. J., and Bronsvoort, W. F. (2005b). A feature-based solution to the persistent naming problem. *Computer-Aided Design and Applications*, 2(1):517–526.

Bonnefoi, P.-F. and Plemenos, D. (2000). Constraint satisfaction techniques for declarative scene modelling by hierarchical decomposition. In *Proceedings 3IA'2000, International Conference on Computer Graphics and Aritifical Intelligence, May 3–4, Limoges, France*. Pergamon Press, Elmsford, NY, USA.

Bonnefoi, P.-F., Plemenos, D., and Ruchard, W. (2004). Declarative modelling in computer graphics,: current results and future issues. In Bubak, M., editor, *Proceedings ICCS 2004, International Conference on Computational Science, June 6–9, Krakow, Poland*, volume 3039 of *Lecture Notes in Computer Science*, pages 80–89. Springer, Berlin, Germany.

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

Bronsvoort, W. F. and Noort, A. (2004). Multiple-view feature modelling for integral product development. *Computer-Aided Design*, 36(10):929–946.

Capoyleas, V., Chen, X., and Hoffmann, C. (1996). Generic naming in generative, constraint-based design. *Computer-Aided Design*, 28(1):17–26.

Chau, H., Chen, X., McKay, A., and De Pennington, A. (2004). Evaluation of a 3D shape grammar implementation. In Gero, J. S., editor, *Design Computing and Cognition '04, July 19–21, MIT, Cambridge, USA*, pages 357–376. Elsevier Science, The Netherlands.

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

Coyne, B. and Sproat, R. (2001). Wordseye: An automatic text-to-scene conversion system. In *Proceedings SIGGRAPH 2001, 28th annual conference on Computer graphics and interactive techniques, August 12–17, Los Angeles, California, USA*, pages 487–496. ACM Press, New York, NY, USA.

Durand, C. and Hoffmann, C. M. (2000). A systematic framework for solving geometric constraints analytically. *Journal of Symbolic Computation*, 30(5):493–519.

Een, N. and Sörensson, N. (2004). An extensible SAT solver. In Giunchiglia, E. and Tacchella, A., editors, *Theory and Applications of Satisfiability Testing, 6th International Conference, SAT 2003, Santa Margherita Ligure, Italy, May 5–8, 2003, Selected Revised Papers*, volume 2919 of *Lecture Notes in Computer Science*. Springer, Berlin, Germany.

Essert-Villard, C., Schreck, P., and Dufourd, J.-F. (2000). Sketch-based pruning of a solution space within a formal geometric constraint solver. *Artificial Intelligence*, 124(1):139–159.

Fudos, I. and Hoffmann, C. M. (1996). Correctness proof of a geometric constraint solver. *International Journal of Computational Geometry and Applications*, 6(4):405–420.

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.

Gaildrat, V. (2007). Declarative modelling of virtual environments, overview, issues and applications. In Dimitri Plemenos, G. M., editor, *Proceedings 3IA'2007, International Conference on Computer Graphics and Artificial Intelligence, May 30–31, Athens, Greece*, volume 10. Pergamon Press, Elmsford, NY, USA.

Gao, X., Lin, Q., and Zhang, G. (2006). A C-tree decomposition algorithm for 2D and 3D geometric constraint solving. *Computer-Aided Design*, 38(1):1–13.

Hoffmann, C. M. and Joan-Arinyo, R. (1998). On user-defined features. *Computer-Aided Design*, 30(5):321–332.

Hoffmann, C. M. and Joan-Arinyo, R. (2002). Parametric modelling. In an Josef Hoschek, G. F., editor, *Handbook of Computer-Aided Design*, pages 519–541. Elsevier Science, The Netherlands.

Hoffmann, C. M. and Kim, K.-J. (2001). Towards valid parametric CAD models. *Computer-Aided Design*, 33(1):81–90.

Hoffmann, C. M., Lomonosov, A., and Sitharam, M. (2001a). Decomposition plans for geometric constraint systems, Part I: performance measures for CAD. *Journal of Symbolic Computation*, 31(4):376–408.

Hoffmann, C. M., Lomonosov, A., and Sitharam, M. (2001b). Decomposition plans for geometric constraint systems, Part II: new algorithms. *Journal of Symbolic Computation*, 31(4):409–427.

Hoffmann, C. M. and Vermeer, P. J. (1995). Geometric constraint solving in $\Re^2$ and $\Re^3$. In Du, D. and Hwang, F., editors, *Computing in Euclidean Geometry, Second Edition*, pages 266–298. World Scientific Publishing, Singapore.

Joan-Arinyo, R., Luzón, M. V., and Soto, A. (2003). Genetic algorithms for root multiselection in constructive geometric constraint solving. *Computers & Graphics*, 27(1):51–60.

Joan-Arinyo, R. and Mata, N. (2001). Applying constructive geometric constraint solvers to geometric problems with interval parameters. *Nonlinear Analysis – Theory, Methods & Application*, 47(1):213–224.

Joan-Arinyo, R. and Soto, A. (1997). A correct rule based geometric constraint solver. *Computers & Graphics*, 21(5):599–609.

Joan-Arinyo, R., Soto, A., Vila-Marta, S., and Vilaplana-Pasto, J. (2004). Revisiting decomposition analysis of geometric constraint graphs. *Computer-Aided Design*, 36(2):123–140.

Kramer, G. A. (1992). *Solving Geometric Constraint Systems: a Case Study in Kinematics*. The MIT Press, Cambridge, MA, USA.

Kripac, J. (1995). A mechanism for persistently naming topological entities in history-based parametric solid models. *Computer-Aided Design*, 29(2):113–122.

Kwaiter, G., Gaildrat, V., and Caubet, R. (1997). Dem2ons: A high level declarative modeler for 3D graphics applications. In *Proceedings CISST97, International Conference on Imaging Science Systems and Technology, June 30–July 3, Las Vegas, Nevada, USA*, pages 149–154.

Laman, G. (1970). On graphs and rigidity of plane skeletal structures. *Journal of Engineering Mathematics*, 4(4):331–340.

Lequette, R. (1997). Considerations on topological naming. In Pratt, M., Sriram, R., and Wozny, M., editors, *Product Modeling for Computer Integrated Design and Manufacturing – Proceedings TC5/WG5.2 International Workshop on Geometric Modeling in Computer-Aided Design, May 19–23, Airlie, VA, USA*, pages 394–403. Chapman & Hall, London.

Marcheix, D. and Pierra, G. (2002). A survey of the persistent naming problem. In Lee, K. and Patrikalakis, N., editors, *Proceedings Solid Modeling '02 - Seventh Symposium on Solid Modeling and Applications, 17–21 June, Saarbrücken, Germany*, pages 13–22. ACM Press, New York, NY, USA.

Müller, P., Wonka, P., Haegler, S., Ulmer, A., and Gool, L. V. (2006). Procedural modeling of buildings. *ACM Transactions on Graphics*, 25(3):614–623.

Noort, A. and Bronsvoort, W. F. (2001). Enforcing model validity by automatic adjustment. *Journal of Computing and Information Science in Engineering*, 1(4):311–319.

Oung, J., Sitharam, M., Moro, B., and Arbree, A. (2001). FRONTIER: fully enabling geometric constraints for feature-based modeling and assembly. In Anderson, D. C. and Lee, K., editors, *Proceedings Solid Modeling '01, Sixth ACM Symposium on Solid Modeling and Applications, June 4–8, Ann Arbor, USA*, pages 307–308. ACM Press, New York, NY, USA.

Papadimitriou, C. H. (1995). *Computational Complexity*. Addison-Wesley.

Podgorelec, D. (2002). A new constructive approach to constraint-based geometric design. *Computer-Aided Design*, 34(11):769–785.

Raghothama, S. (2006). Constructive topological representations. In Kobbelt, L. and Wang, W., editors, *Proceedings ACM Symposium on Solid and Physical Modeling, June 6–8, Cardiff, Wales, UK*, pages 39–51. ACM Press, New York, NY, USA.

Raghothama, S. and Shapiro, V. (1998). Boundary representation deformation in parametric solid modeling. *ACM Transactions on Graphics*, 17(4):259–286.

Raghothama, S. and Shapiro, V. (2000). Consistent updates in dual representation systems. *Computer-Aided Design*, 32(8):463–477.

Raghothama, S. and Shapiro, V. (2002). Topological framework for part families. *Journal of Computing and Information Science in Engineering*, 2(4):246–255.

Rappoport, A. (1997). The Generic Geometric Complex (GGC): a modeling scheme for families of decomposed pointsets. In Hoffmann, C. M. and Bronsvoort, W. F., editors, *Proceedings Solid Modeling '97, Fourth ACM Symposium on Solid Modeling and Applications, May 14–16, Atlanta, Georgia, USA*, pages 19–30. ACM Press, New York, NY, USA.

Rossignac, J. (2007). Solid and physical modeling. In Webster, J., editor, *Wiley Encyclopedia of Electrical and Electronics Engineering*. J. Wiley & Sons, Malden, MA, USA.

Rossignac, J. R. and O'Connor, M. A. (1988). SGC: a dimension-independent model for pointsets with internal structures and incomplete boundaries. In Wozny, M., Turner, J., and Preiss, K., editors, *Proceedings of the 1988 IFIP/NSF Workshop on Geometric Modeling, Renselaerville, NY, USA*, pages 145–180. Elsevier Science, The Netherlands.

Ruchaud, W. and Plemenos, D. (2002). Multiformes: a declarative modeller as a 3D scene sketching tool. In *Proceedings ICCVG'2002, International Conference on Computer Vision and Graphics, September 25–29, Zakopane, Poland*. Elsevier Science, The Netherlands.

Shah, J. J. and Mantyla, M. (1995). *Parametric and Feature-based CAD/CAM; Concepts, Techniques and Applications*. John Wiley & Sons.

Shapiro, V. and Vossler, D. L. (1995). What is a parametric family of solids? In Hoffmann, C. M. and Rossignac, J. R., editors, *Proceedings of the Third ACM/IEEE Symposium on Solid Modeling and Applications, May 17–19, Salt Lake City, Utah, USA*, pages 43–54. ACM Press, New York, NY, USA.

Shea, K., Cagan, J., and Fenves, S. (1997). A shape annealing approach to optimal truss design with dynamic grouping of members. *Journal of Mechanical Design*, 119(3):388–394.

Silva, J. P. and Sakalla, K. A. (1996). Grasp - a new search algorithm for satisfiability. In *Proceedings ICCAD 1996, IEEE/ACM International Conference on Computer-Aided Design, November 10–14, San Jose, California, USA*, pages 220–227. IEEE Press, USA.

Sitharam, M. (2006). Wellformed systems of point incidences for resolving collections of rigid bodies. *International Journal of Computational Geometry and Applications*, 16(5):591–615.

Sitharam, M., Oung, J.-J., Zhou, Y., and Abree, A. (2006). Geometric constraints within feature hierarchies. *Computer-Aided Design*, 38(1):22–38.

Stiny, G. and Gips, J. (1972). Shape grammars and the generative specification of painting and sculpture. In Petrocelli, O. R., editor, *The Best Computer Papers of 1971*, pages 125–135. Auerbach, Philadelphia.

Sutherland, I. E. (2003). Sketchpad: A man-machine graphical interface. Technical Report 574, University of Cambridge, UK. Original PhD thesis from 1963 with a preface by Alan Blackwell and Kerry Roddenphone, available from http://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-574.pdf.

Tapia, M. (1999). A visual implementation of a shape grammar system. *Environment and Planning B: Planning and Design*, 26(1):59–73.

Tseitin, G. S. (1968). On the complexity of derivation in propositional calculus. In Slisenko, editor, *Structures in Constructive Mathematics and Mathematical Logic*, pages 115–125. Consultants Bureau, New York, USA.

Ullmann, J. (1976). An algorithm for subgraph isomorphism. *Journal of the ACM*, 23(1):31–42.

van den Berg, E., van der Meiden, H. A., and Bronsvoort, W. F. (2003). Specification of freeform features. In Elber, G. and Shapiro, V., editors, *Proceedings of Solid Modelling '03, Eighth ACM Symposium on Solid Modeling and Applications, June 16–20, Seattle, Washington, USA*, pages 56–64. ACM Press, New York, NY, USA.

van der Meiden, H. A. and Bronsvoort, W. F. (2005a). A constructive approach to calculate parameter ranges for systems of geometric constraints. In Kobbelt, L. and Shapiro, V., editors, *Proceedings SPM 2005, ACM Symposium on Solid and Physical Modeling, June 13–15, Cambridge, MA, USA*, pages 135–142. ACM Press, New York, NY, USA.

van der Meiden, H. A. and Bronsvoort, W. F. (2005b). An efficient method to determine the intended solution for a system of geometric constraints. *International Journal of Computational Geometry and Applications*, 15(3):279–298.

van der Meiden, H. A. and Bronsvoort, W. F. (2006a). A constructive approach to calculate parameter ranges for systems of geometric constraints. *Computer-Aided Design*, 38(4):275–283. (Special issue on 2005 ACM Symposium on Solid and Physical Modeling).

van der Meiden, H. A. and Bronsvoort, W. F. (2006b). Solving topological constraints for declarative families of objects. In Kobbelt, L. and Wang, W., editors, *Proceedings ACM Symposium on Solid and Physical Modeling, June 6–8, Cardiff, Wales, UK*, pages 63–71. ACM Press, New York, NY, USA.

van der Meiden, H. A. and Bronsvoort, W. F. (2007a). Solving topological constraints for declarative families of objects. *Computer-Aided Design*, 39(8):652–662. (Special issue on 2006 ACM Symposium on Solid and Physical Modeling).

van der Meiden, H. A. and Bronsvoort, W. F. (2007b). Tracking topological changes in feature models. In Lévy, B. and Manocha, D., editors, *Proceedings ACM Symposium on Solid and Physical Modelling, June 4–6, Beijing, China*, pages 341–346. ACM Press, New York, NY, USA.

van der Meiden, H. A. and Bronsvoort, W. F. (2008). Solving systems of geometric constraints using non-rigid clusters. In Chen, F. and Jüttler, B., editors, *Advances in Geometric Modelling and Processing, Proceedings GMP Conference, April 23–25, Hangzhou, China*, volume NCS4975 of *Lecture Notes in Computer Science*, pages 423–436. Springer, Berlin, Germany.

Vergeest, J. S. M., Wang, C., Song, Y., and Horvath, I. (2002). Dynamic shape typing. In *CDROM Proceedings DETC'02, ASME Design Engineering Technical Conferences, September 29 - October 2, Montreal, Canada*. ASME.

Weisstein, E. W. (2008a). Degenerate. From MathWorld–A Wolfram Web Resource.
http://mathworld.wolfram.com/Degenerate.html.

Weisstein, E. W. (2008b). Viviani's curve. From MathWorld–A Wolfram Web Resource.
http://mathworld.wolfram.com/VivianisCurve.html.

Wu, J., Zhang, T., Zhang, X., and Zhou, J. (2001). A face based mechanism for naming, recording and retrieving topological entities. *Computer-Aided Design*, 33(10):687–698.

Wu, W.-T. (1986). Basic principles of mechanical theorem proving in elementary geometries. *Journal of Automated Reasoning*, 2(3):221–252.

Zhang, L., Madigan, C. F., Moskewicz, M. W., and Malic, S. (2001). Efficient conflict driven learning in a boolean satisfiability solver. In *Proceedings ICCAD 2001, IEEE/ACM International Conference on Computer-Aided Design, November 4–8, San Jose, California, USA*, pages 279–285. IEEE Press, USA.

Summary

Semantics of Families of Objects

PhD. Thesis

H.A. van der Meiden

Design disciplines nowadays heavily depend on Computer-Aided Design (CAD) systems. Most current CAD systems are so-called feature modelling systems. In these systems, a model is built from features, which are parametrised shapes that add or remove material from the model, e.g. protrusions, holes and slots. The exact shape and relative position of features is determined by the parameters of the features.

Traditionally, a CAD model is interpreted as representing a single (physical) object. However, a parameterised model, e.g. a feature model, can also be interpreted as representing a family of objects. The members of the family are all objects that can be obtained by varying the values of the parameters of the model.

A CAD model of a family of objects is useful in several situations. First, the model can be used for manufacturing series of similar products, e.g. tools of different sizes, and even customised products. Secondly, the model can be reused as a part of a larger CAD model, with appropriate parameter values to fit it. Thus, modelling families of objects can yield increased design productivity and considerable cost reduction.

However, there are several major obstacles when using current CAD systems for modelling families of objects.

When instantiating a member of a family, by specifying parameter values, it is necessary to verify that the model satisfies particular design requirements, i.e. that the model is a valid member of the family. Verifying large models by hand is costly, and with current systems, this cannot be done automatically. Important functional requirements for features, in particular those related to the topology of a model, cannot be specified, let alone verified automatically.

Previous research on Semantic Feature Modelling has resulted in methods to specify and verify such functional requirements (semantics) for fea-

tures. This research builds on those methods, and extends the results to families of objects.

Another problem is that in current systems, the realization of a model, i.e. its geometric representation, is generated procedurally from a modelling history. In particular, the features add or remove material from the model in a fixed order, typically the order in which features were added to the model. Such a procedure, however, may conflict with the functional requirements specified in the model, in particular, topological constraints. Because of this, some realisations, even though they satisfy all given requirements, may not be found by the procedure. As a result, some members of a modelled family may be missed.

Therefore, we present a new, declarative model for families of objects, where shape and function of features, and the model as a whole, are specified by geometric and topological constraints, and realisations are determined by first solving the geometric and then the topological constraints. By solving the constraints, we can find all possible realisations, and in this way, a family of objects is completely specified by the constraints in the model, and not dependent on procedural details of how the geometry is generated.

The new model requires that systems of geometric constraints can be solved in 3D, whereas current CAD systems usually only solve constraints in 2D. Also, the solver must be able to find all solutions, which solvers used in current CAD systems cannot, and it should be able to select solutions efficiently using various selection criteria. Finally, the solver must efficiently handle incremental changes to the constraint system, so that it can be used for computing parameter ranges (see below). A completely new geometric constraint solving algorithm has been developed that satisfies these requirements.

Solving topological constraints is a new area in which very little work has been done previously. Topological constraints are here imposed on features or on topological entities that are defined by a feature in the model, e.g. a face of a feature may be constrained to be on the boundary of the model. However, each feature entity can be split into several topological entities in the realization of the model. Therefore, topological constraints on feature entities are mapped to topological constraints on model entities, which in turn are mapped to a system of Boolean constraints. The Boolean constraint system is solved by a Boolean SAT solver, and from the solutions that are found, the realisations are constructed.

A third problem is the complex relation between the parameters of a model and the members of the corresponding family. Only certain combinations of parameter values result in objects that satisfy all constraints in the model. Finding such a combination of parameter values can be difficult for an end-user of the model, i.e. someone who wants to instantiate a particular family member from the model. Also, the designer of a family

should be able to verify that the family behaves as intended, i.e. he should be able to examine how parameter values affect the geometric and topological properties of the corresponding members of the family. Current systems do not provide sufficient support for finding members of, and analysing the behaviour of, families of objects.

We present a method to determine the range of values that can be assigned to a parameter, such that the model satisfies all geometric and topological constraints, i.e. the parameter range corresponding to valid members of the family. This can help end-users with instantiating members of the family. The method first determines so-called critical values; the values for which geometric subproblems degenerate or for which topological entities degenerate. Then, for each interval between subsequent critical values, it is determined whether the interval belongs to the parameter range. The critical values can also be used to determine at which parameter values changes occur in the topology of a model. The latter can help designers with analysing the behaviour of the model.

The presented model for families of objects, and the methods for determining realisations of such models, i.e. the geometric and topological solvers, have been implemented in a prototype feature modelling system that has been developed at Delft University of Technology. Methods for determining parameter ranges are currently being implemented separately for simplified models.

The declarative approach to modelling families of objects is significantly different from the currently used history-based approach, and therefore cannot be directly implemented in most existing CAD systems. However, the geometric and topological solvers, and the methods for parameter range computation, are independent of this model, and can be applied to a broad range of applications that involve geometry and/or topology.

Samenvatting

Summary in Dutch

Semantiek van families van objecten

Proefschrift

H.A. van der Meiden

Ontwerpdisciplines maken tegenwoordig intensief gebruik van Computer-Aided Design (CAD) systemen. De meeste huidige CAD systemen zijn zgn. feature modelling systemen. In dergelijke systemen worden modellen opgebouwd uit features, d.w.z. parametrische vormen die materiaal aan het model toevoegen of uit het model verwijderen, zoals gaten, ribben en uitsparingen. De exacte vorm en relatieve positie van features worden bepaald door waarden voor de parameters op te geven.

Traditioneel wordt een CAD model geïnterpreteerd als een representatie van een enkel (fysiek) object. Een parametrisch model kan echter ook worden geïnterpreteerd als een representatie van een familie van objecten. De leden van de familie zijn alle objecten die kunnen worden verkregen door de waarden van de parameters van het model te variëren.

Een model voor een familie van objecten kan voor verschillende doeleinden worden gebruikt. Ten eerste kan het model worden gebruikt om een serie van gelijksoortige producten te fabriceren, en zelfs op maat gemaakte producten. Daarnaast kan een dergelijk model worden hergebruikt als deel van een groter model, door passende parameterwaarden te kiezen. Dus, het modelleren van families van objecten kan de productiviteit van ontwerpers verhogen en daardoor kostenbesparingen opleveren.

Er zijn echter nog aanzienlijke obstakels bij het gebruik van de huidige CAD systemen voor het modelleren van families van objecten.

Wanneer een lid van een familie wordt geïnstantieerd, door parameterwaarden te specificeren, moet gecontroleerd worden of het model aan de ontwerpeisen voldoet, d.w.z. of het model een geldig lid van de familie voorstelt. Grote modellen met de hand controleren is kostbaar, en met de huidige modelleersystemen kan dit niet automatisch. Belangrijke functionele eisen voor features, met name die met topologische eigenschappen te

maken hebben, kunnen niet worden gespecificeerd, laat staan automatisch gecontroleerd.

Voorgaand onderzoek op het gebied van Semantic Feature Modelling heeft geresulteerd in methoden voor het specificeren en controleren van dergelijke functionele eisen (semantiek) voor features. Dit onderzoek bouwt voort op de resultaten van dat onderzoek, toegepast op families van objecten.

Een ander probleem is dat in huidige systemen de realisatie van een model, d.w.z. een geometrische representatie, op procedurele wijze wordt bepaald op basis van een 'modelling history'. Features voegen materiaal toe, of verwijderen materiaal, in een vaste volgorde, meestal de volgorde waarin de features aan het model zijn toegevoegd. Een dergelijke procedure kan conflicteren met de functionele eisen die in het model gespecificeerd zijn, met name topologische eisen. Daardoor is het mogelijk dat er realisaties van het model zijn die wel aan alle eisen voldoen, maar die niet door de procedure worden bepaald. Het gevolg is dat er leden van de familie kunnen ontbreken.

Om deze reden stellen wij een declaratief model voor families van objecten voor, waarbij vorm en functie van features, en het model als geheel, worden bepaald door geometrische en topologische constraints. Realisaties worden gevonden door eerst de geometrische en dan de topologische constraints op te lossen. Met deze aanpak worden alle mogelijke realisaties gevonden, zodat een familie van objecten volledig is gespecificeerd door de constraints in het model, en niet afhankelijk is van procedurele details van de manier waarop de geometrie wordt gegenereerd.

Voor het nieuwe model is een solver vereist die stelsels geometrische constraints in 3D kan oplossen, terwijl huidige CAD systemen meestal alleen constraints in 2D oplossen. Ook moet de solver alle oplossingen kunnen vinden, iets wat huidige CAD systemen niet kunnen, en moet deze efficiënt oplossingen kunnen selecteren aan de hand van verschillende selectiecriteria. Tot slot moet de solver ook efficiënt omgaan met incrementele veranderingen in het stelsel, zodat deze kan worden gebruikt voor het berekenen van het bereik van parameters (zie verderop). Hiervoor is een compleet nieuw algoritme voor het oplossen van geometrische constraints ontwikkeld.

Het oplossen van topologische constraints is een nieuw gebied waarop nog weinig onderzoek is verricht. Topologische constraints worden hier opgelegd aan features of topologische elementen van features, b.v. dat een zijvlak van een feature op de begrenzing van het model moet liggen. In de realisatie van het model kan elk feature-element zijn opgesplitst in verschillende model-elementen, zodat constraints op feature-elementen worden afgebeeld op een stelsel constraints op model-elementen. Deze laatste constraints worden op hun beurt afgebeeld op een stelsel Booleaanse constraints. Dit stelsel wordt opgelost door een SAT solver, en van de gevonden oplossingen worden de realisaties van het model geconstrueerd.

Een derde probleem is de complexe relatie tussen de parameters van een model en de leden van de overeenkomstige familie. Alleen bepaalde combinaties van parameterwaarden resulteren in objecten die aan alle constraints in het model voldoen. Het vinden van een dergelijke combinatie van parameterwaarden kan moeilijk zijn voor de eindgebruiker van het model, d.w.z. degene die een bepaald lid van de familie wil instantiëren. Bovendien, de ontwerper van een familie moet kunnen controleren of het model zich gedraagt zoals bedoeld, d.w.z., hij moet kunnen onderzoeken hoe parameter waarden de geometrische en topologische eigenschappen van de overeenkomstige leden beïnvloeden. De huidige systemen bieden weinig ondersteuning voor het vinden van leden van, en het analyseren van het gedrag van families van objecten.

Wij presenteren een methode om het bereik van een parameter te bepalen waarvoor het model aan alle constraints voldoet, oftewel, het parameterbereik dat overeenkomt met geldige leden van de familie. Dit kan eindgebruikers helpen bij het instantiëren van leden van de familie. De methode bepaalt eerst de zogenaamde kritieke waarden, d.w.z. de waarden waarvoor geometrische deelproblemen degenereren of waarvoor topologische elementen degenereren. Vervolgens wordt voor elk interval tussen opeenvolgende kritieke waarden bepaald of het wel of niet bij het parameterbereik behoort. De kritieke waarden kunnen ook worden gebruikt om te bepalen voor welke waarden de topologie van het model verandert. Dit laatste kan ontwerpers helpen bij het analyseren van het gedrag van het model.

Het voorgestelde model voor families van objecten, en de methoden om realisaties van dergelijke modellen te vinden, zijn geïmplementeerd in een prototype feature modelling systeem dat is ontwikkeld aan de Technische Universiteit Delft. Methoden voor het bepalen van parameterbereik en kritieke waarden worden momenteel los geïmplementeerd voor vereenvoudigde modellen.

De declaratieve aanpak voor het modelleren van families van objecten is significant anders dan de gebruikelijke history-gebaseerde benadering, en kan daarom ook niet direct in de huidige CAD systemen worden geïmplementeerd. De geometrische en topologische solvers, en de methodes voor het bepalen van het bereik en de kritieke waarden van parameters, kunnen echter los van dat model worden ingezet voor veel andere toepassingen die met geometrie en topologie te maken hebben.

Curriculum Vitae

Hilderick Anne (Rick) van der Meiden was born June 19th, 1977 in Zevenaar, the Netherlands.

He went to Comenius College in Capelle aan de IJssel from 1989 to 1995 and received a VWO diploma.

In 1995 he started his studies in Computer Science at Delft University of Technology. He did his Master's project at the Computer Graphics and CAD/CAM group of the Faculty of Electrical Engineering, Mathematics and Computer Science. The title of his Master's thesis was "Specification of freeform feature classes", and he received his MSc degree in 2003.

Between September 2003 and September 2007, he has been working as a PhD student in the same group at the same university. His research, of which the results are presented in this thesis, was sponsored by NWO.

In the remainder of 2007 and in 2008, he was employed as a post-doc, working on a SURF sponsored project about intelligent feedback for linear algebra learning tools.