Validity Maintenance in Semantic Feature Modeling

Printed by Universal Press, Science Publishers
Veenendaal, The Netherlands

CIP–DATA KONINKLIJKE BIBLIOTHEEK, DEN HAAG

Bidarra de Almeida, António Rafael E.

Validity Maintenance in Semantic Feature Modeling / António Rafael E. Bidarra de Almeida. - [S.l. : s.n.]. Ill.
Thesis Technische Universiteit Delft. – With ref. – With summary in Dutch.
ISBN 90–9012599–X
NUGI 855
Subject heading: validity maintenance / feature semantics / feature modeling

Validity Maintenance in Semantic Feature Modeling

PROEFSCHRIFT

ter verkrijging van de graad van doctor
aan de Technische Universiteit Delft,
op gezag van de Rector Magnificus prof.ir. K.F. Wakker,
in het openbaar te verdedigen ten overstaan van een commissie,
door het College voor Promoties aangewezen,
op maandag 17 mei om 10.30 uur
door

António Rafael Emiliano BIDARRA DE ALMEIDA

engenheiro electrotécnico, Universidade de Coimbra
geboren te Lissabon, Portugal

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

Toegevoegd promotor:
Dr. W.F. Bronsvoort

Samenstelling promotiecommissie:
Rector Magnificus, voorzitter
Prof.dr.ir. F.W. Jansen, Technische Universiteit Delft, promotor
Dr. W.F. Bronsvoort, Technische Universiteit Delft, toegevoegd promotor
Prof.dr. H. Koppelaar, Technische Universiteit Delft
Prof.dr. I. Horváth, Technische Universiteit Delft
Prof.dr.ir. H.J.J. Kals, Universiteit Twente en Technische Universiteit Delft
Prof.dr. J.P. Teixeira, Universidade Nova de Lisboa
Prof.dr. C.M. Hoffmann, Purdue University

This research was financially supported by the Portuguese Foundation for Scientific and Technological Research (FCT), under Program Praxis XXI.

The production of this thesis was sponsored by Spatial Technology Inc.

Preface

The pre-history of this thesis dates back to the period 1990-1994, when I started my research in geometric and feature modeling, at the University of Coimbra. I keep many pleasant memories of those times, and at least two should be mentioned here. The first one regards the outstanding ambiance during that period, for which I wish to thank each and every member of the group. A special *muito obrigado* goes to Joaquim Madeira, Jorge Bernardino and Abel Gomes, whose friendship of then hasn't but grown steadily, despite time and distance.

The second grateful memory concerns the opportunities we were offered to listen and talk to foreign scientists, either at in-house seminars or at conferences abroad. In one such seminar, I got acquainted with the research work on feature modeling at Delft University of Technology. While I positively was in tune with his ideas, I could not imagine then that the invited speaker, Erik Jansen, would later become my promotor. It was probably about this time that he started appreciating Portuguese music... Thanks for coming !

Shortly after that, at a conference in Montreal, "Delft showed up" again in my course, via my future supervisor, Wim Bronsvoort. Once more, reciprocal tuning went high, eventually resulting in me starting this PhD project at the Computer Graphics and CAD/CAM group, in Delft. What I learned from Wim goes far beyond the borders of our research area (and, incidentally, of these pages, too...), so I cordially thank him for this valuable legacy.

This project was launched with the crucial cooperation of Jorge Pamies Teixeira, for whose diligent support, in particular during the periods I spent in Lisbon, I wish to thank him.

The terrific working environment in our group is an exemplary *collective masterpiece*, whose strokes convey much of the rich variety of characters in-house. I wish to thank each of these *artists* for their contribution, in one way or another, to my work.

Many keen ideas and comments came out of sparring discussions with the other PhD students involved in feature modeling: Winfried van Holland, the *Brooklyn man*, who has the rare gift of the-right-joke-at-the-right-moment; *grand* Maurice Dohmen, my former roommate, who actively "put up with" many creative discussions; my neighbor Klaas Jan de Kraker, whose versatility showed up, among other things, in the *flippo* "research project"; and my current roommate, Alex Noort, who is always ready to get his hands on new, exciting problems. Furthermore, Maurice and Klaas Jan did a great job implementing the first version of the SPIFF prototype.

Graduate students Marco Zwet and Abdel Idri bravely caught the mood at the AIO Lab. Moreover, Abdel provided a valuable source of feedback during his implementation of the Feature Library Manager.

The other PhD colleagues of our group were great, inventive mates during the term of my project, also in all sorts of after work get-togethers: Theo van Walsum, Wim de Leeuw and Erik Reinhard (God only knows why they still address me in English...); *maxime fabulosum* polyglot *I.* Ari Sadarjoen and multi-faceted sportsman Freek Reinders; and, lately, Paul de Bruin and Michal Koutek.

Much of this work wouldn't have been possible without the dedicated support of our technical staff: Piter Jonker, Kees Seebregts, Peter Kailuhu, Aadjan van der Helm and, currently, *motor freak* Ruud de Jong (to whom I also owe my knowledge of dutch idioms...) and Bart Vastenhouw. Moreover, secretaries Toos Brussee and Coby Bouwer wisely combined efficiency and cheerfulness.

Last, but definitely not least, I wish to thank my family for their invaluable support, and all my housemates at Lepelenburg for their interest and encouragement.

Rafael Bidarra
Utrecht, March 1999

Contents

To my parents

1

Introduction

Feature modeling is increasingly being used for modeling products. One of its main advantages over conventional geometric modeling techniques is the ability to associate functional and engineering information to shape information in a product model. This can be, for example, the function of some part of the product for the end-user, or information about the way some part of the product is manufactured.

The basic entity in a feature model is the *feature*, defined as *a representation of shape aspects of a product that are mappable to a generic shape and are functionally significant for some product life-cycle phase*. Typical examples of features are holes, slots and protrusions.

An essential element of a feature is thus that it has a well-defined meaning, or *semantics*, in a particular context or life-cycle activity. Consequently, to build and maintain a feature model, feature-based modeling systems require considerably more advanced facilities than conventional geometric modeling systems, which manage shape information only.

Two important aspects of the above definition are not well covered by most current feature-based modeling systems. First, feature semantics is poorly defined, inevitably limiting the capability of capturing design intent in the model. Second, feature semantics is poorly maintained, permitting previous explicit design intent to be overruled. One of the main reasons for this is that such systems are still too tied to methods and

techniques of conventional geometric modeling, e.g. they strongly rely on a history-based approach of the modeling process.

This thesis addresses various problems concerning the specification and preservation of feature semantics in a feature model, generally called *validity maintenance*.

This chapter first briefly introduces validity maintenance issues in feature modeling, outlining the main problems that motivated this research (Section 1.1). Next, it presents the main research goals of the work described in this thesis (Section 1.2). The chapter closes with an outline of the thesis (Section 1.3).

1.1 Validity maintenance

Current feature modeling systems provide the user with "engineering rich" dialogs aimed at the creation and manipulation of feature instances. In some systems, "features" occur solely at the user interface level, whereas in the product model only the resulting geometry is stored. Such systems are in essence only geometric modelers. Most other feature modeling systems, although they store information about features in the product model, fail to adequately maintain the meaning of features throughout the modeling process. For example, a modeling operation on one feature may affect the semantics of other features, without the user even being notified by the system, let alone assisted in overcoming the situation.

This is illustrated in the example of Figure 1.1. Assume that the two longer blind holes in the part were positioned relative to the block right-hand face, whereas the rounded pocket was positioned relative to the step side face, as indicated in Figure 1.1.a. If the width of the step is now increased, the rounded pocket overlaps with the two blind holes, "suppressing" their circular bottom faces from the model boundary, see Figure 1.1.b. Consequently, the two blind holes now have the shape imprint of through holes. Stated differently, the semantics of the blind holes has been changed. If the shape now produced was indeed desired, it might have been more appropriate not to use blind holes, but through holes instead, attached to the bottom of the rounded pocket and the bottom of the base block.

Assessing the extent to which feature semantics is kept in a feature model is an important issue in feature validity maintenance.

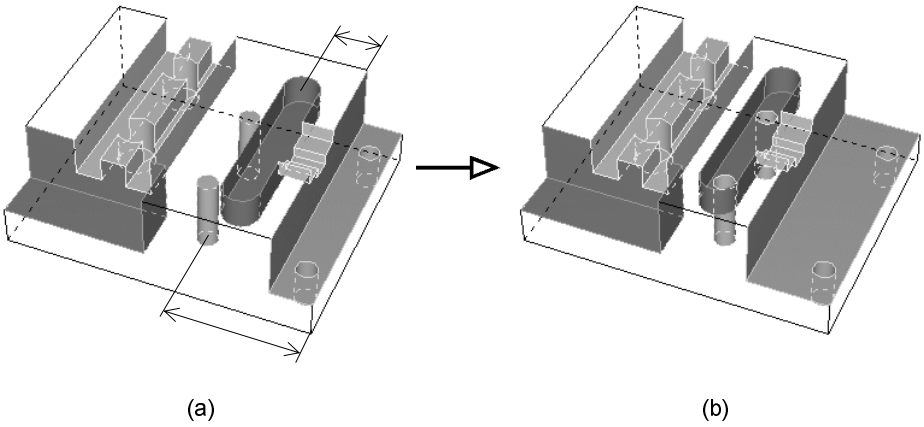


Figure 1.1 – Changing feature semantics with a modeling operation

Some recent research work has focused on the validation of features, both on various validity specification issues (Brunetti et al. 1996, Hoffmann and Joan-Arinyo 1998) and on validity maintenance (Dohmen et al. 1996, Mandorli et al. 1997). One of the main conclusions of this research is that a declarative scheme is preferable over the conventional procedural modeling approaches. In a declarative approach, the specification of each feature class includes the validity criteria that determine the semantics of all its feature instances. The feature modeler, in turn, is responsible for the maintenance of all features in the product model, in conformity with those criteria.

Research prototype systems that do have some form of validity maintenance, see for example (Mandorli et al. 1995, Vieira 1995, Dohmen 1997), are limited to the detection of a number of predefined invalid situations, for which the only solution offered by the modeling system is the rejection of the concerning modeling operation. This rigid scheme considerably hinders the modeling process, yet permits many unanticipated inconsistencies in the model.

Mostly, validity violations are due to modeling operations that cause overlapping features to affect each other's semantics, so-called *feature interactions*. It is therefore important to manage feature interaction phenomena in the context of validity maintenance, so that all relevant inter-

action situations can be detected, reported and handled in an appropriate way (Regli and Pratt 1996).

In conclusion, raising the level of assistance provided to the user in maintaining and recovering model validity is essential to bring feature technology to maturity. This thesis' contribution is in that direction.

1.2 Research goals

The main goal of the research presented in this thesis is the development of a global solution to the validity maintenance problems mentioned above. In this solution, the following partial goals should be achieved:

1. Feature libraries should include declarative specifications of feature classes, each containing a complete description of the specific semantics required for its feature instances.
2. The feature model should be fully specified as a set of interrelated feature and constraint instances. Furthermore, its structure and evaluated geometry should be unambiguously determined without invoking any model history considerations.
3. The computational cost of geometric model re-evaluation, after a modeling operation, should be kept independent of the number of features in the model.
4. Each modeling operation should be monitored, in order to assess the conformity of each feature in the model with its validity criteria. In particular, feature interactions should be handled. Additionally, every validity violation should not only be detected, but also be documented, reported to the user, possibly with context-sensitive system hints, and corrected.

Implementation and evaluation of this new approach to feature modeling –from here on designated *semantic feature modeling*– should be carried out within the prototype feature modeling system SPIFF (Bronsvoort et al. 1997).

1.3 Thesis overview

This thesis is organized as follows:

Chapter 2 surveys current feature modeling approaches, identifies their main shortcomings, and presents an overview of the semantic feature modeling approach.

Chapter 3 deals with the feature class specification scheme of this approach, with emphasis on specification of feature validity criteria.

Chapter 4 describes the semantic feature model, and its basic maintenance mechanisms. Special attention is here paid to the Cellular Model, the geometric representation of the product.

Chapter 5 discusses the fundamental notions of feature interactions, and gives a classification of interaction phenomena.

Chapter 6 presents mechanisms and algorithms for the detection of feature interactions in the semantic feature model.

Chapter 7 describes the validity maintenance scheme of the semantic feature modeling approach. This is illustrated with an extended modeling session.

Chapter 8 points out some directions for future research, and presents concluding remarks about the semantic feature modeling approach.

Parts of the research described in this thesis have been previously published or submitted elsewhere (Bidarra and Bronsvoort 1996, Bidarra et al. 1997, Bidarra et al. 1998a, Bidarra et al. 1998b, Bidarra and Bronsvoort 1999a, Bidarra and Bronsvoort 1999b, Bidarra and Bronsvoort 1999c).

2

Semantic feature modeling

> *"...there is appearing in the trade press, and apparently also in the minds of some software system vendors, a view that a feature is simply a macro in a solid modeler that enables designers to easily create and parameterize such forms as bosses, holes and the like. This view of features as quite synonymous with parametric design is quite far removed from the research view, and any suggestion that research on features is essentially complete since commercial systems are appearing with parametric capabilities for 'form features' is simply incorrect. These new systems are a small step in the right direction, but there is much to be learned from research before true feature-based systems can become a reality."*
>
> *(Dixon et al. 1990)*

Current feature modeling systems suffer from a number of shortcomings with regard to the modeling process. In particular, they lack a complete specification of feature semantics, and thus fail to maintain the meaning of each feature during modeling. Also, modeling operations sometimes are hampered by the model history, and occasionally even have ill-defined semantics.

This chapter identifies the main problems of current feature modeling approaches, with special emphasis on history-based systems (Section 2.1). An alternative way of modeling is then presented, which either avoids or solves those problems (Section 2.2). This new approach is called *semantic feature modeling*. Finally, a short overview is given of the archi-

tecture of the SPIFF system, the prototype feature modeling system in which the ideas proposed in this thesis have been implemented (Section 2.3).

2.1 Current approaches to feature modeling

Almost all current feature modeling systems are parametric, history-based modeling systems, using a boundary representation as main geometric model. The boundary representation can be used for several applications, e.g. process planning for manufacturing. Examples of such systems are the commercial systems Pro/Engineer (Parametric 1996), MicroStation Modeler (Peters 1997), I-DEAS Master Series (SDRC 1998) and Autodesk Mechanical Desktop (Autodesk 1998), and the academic systems of Shah et al. (1990) and Chen and Hoffmann (1995).

History-based modeling systems are procedural systems that, together with the evaluated boundary representation, keep track of information about each modeling operation performed, e.g. the type of feature created, its parameter values, and its model references for positioning. The stored sequence of modeling operations, called the *model history*, completely determines the resulting boundary representation. Creation of a feature produces in the evaluated boundary model the shape imprint characteristic of its feature type. Each new feature is positioned relative to boundary entities of the evaluated model, obtained from previously created features.

Feature instances can be modified by specifying new values for their parameters, or be deleted from the model. This is done by modifying, or deleting, the respective feature creation operation in the model history, after which a new boundary model is evaluated by sequentially re-executing the operations in the modified history. With this scheme, variants of a feature model can easily be created. An example of this is given in Figure 2.1. The model has a base block, a through slot and, attached to the latter, a pocket, see Figure 2.1.b. If the depth of the through slot is decreased, the model history in Figure 2.1.a is re-executed, yielding the model in Figure 2.1.c.

Current feature modeling systems have, at least, six major shortcomings that will now be identified and illustrated with typical examples. The first three have a common cause: a strong dependency on the chronological order of feature creation. The fourth shortcoming is due to constraint solving limitations. The fifth shortcoming is related to the historical

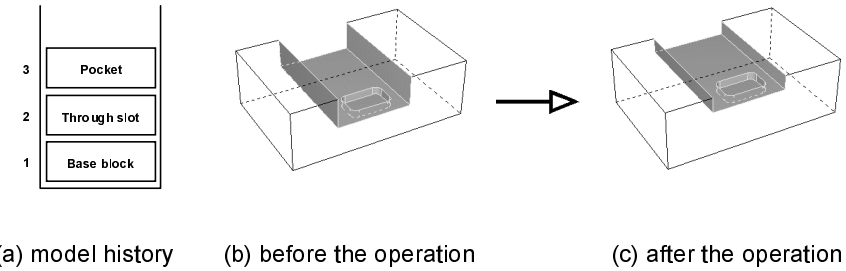


Figure 2.1 – Example of re-executing the model history

evolution of the entities in the evaluated boundary model. The sixth problem is mainly due to the use of a manifold boundary representation.

The first shortcoming is that re-executing the whole model history after modifying or deleting a feature has a computational cost that is proportional to the number of features in the model. Several methods have been devised to improve this, e.g. storing the intermediate evaluated model between each history step. Then, only the history steps after the modified, or deleted, operation need to be re-executed. However, storing intermediate models between all history steps requires a considerable amount of storage space, proportional to the square of the model history size. An alternative improvement is to store only the deltas between history steps, and to rollback to the state from which the model needs to be re-evaluated. This requires less storage space, but more computation time again. In any case, the sequence of history steps re-executed almost always includes more features than those actually modified by the operation in question.

The second shortcoming is that history-based re-evaluation of the boundary model does not always guarantee that the evaluated model matches the specified parameters of features that overlap. This is illustrated in the model of Figure 2.2.b, which consists of a base block, a blind hole and a protrusion. Because the blind hole and the protrusion do not overlap, the history of this model could be either that in Figure 2.2.a or that in Figure 2.2.c. However, if the blind hole depth is increased, so that it now overlaps with the protrusion, different models will result for the two histories: in case (a), re-execution of the history produces a blind hole with

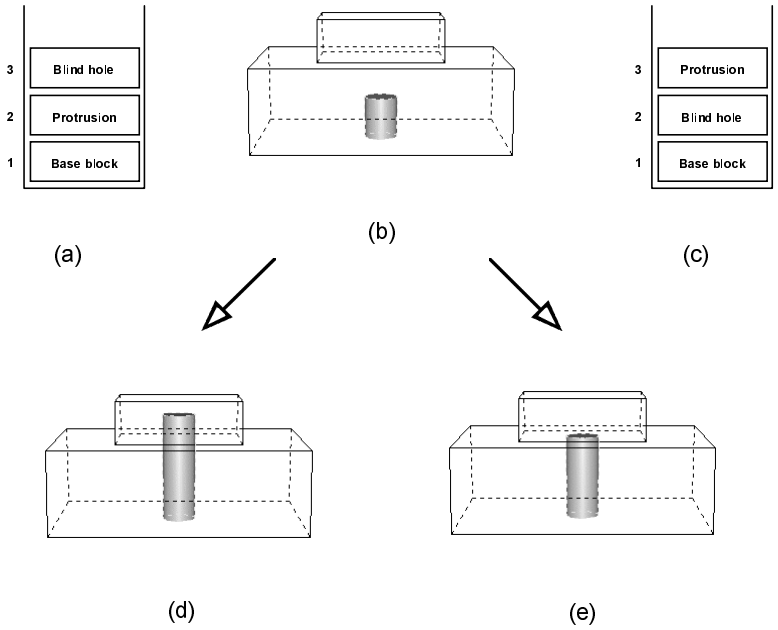


Figure 2.2 – Set operations problem in history-based model re-evaluation

the expected depth (Figure 2.2.d), whereas, in case (c), the blind hole will be "truncated" by the protrusion, its depth becoming equal to the block height (Figure 2.2.e). The problem is caused here by the static precedence order upon which model re-evaluation is based: the *chronological feature creation order*. The resulting models are different because the evaluation process uses two non-associative set operations according to the nature of a feature being processed: union for additive features, and difference for subtractive features. The order in which these are executed determines the result: performing the union of the protrusion as last operation prevents the blind hole to exhibit its nominal depth in the model of Figure 2.2.e.

The third shortcoming of history-based re-evaluation of the model is that it cannot always process feature modification operations such as, for example, feature re-attachment or re-positioning relative to other model entities. This is illustrated in the example of Figure 2.3. The model con-

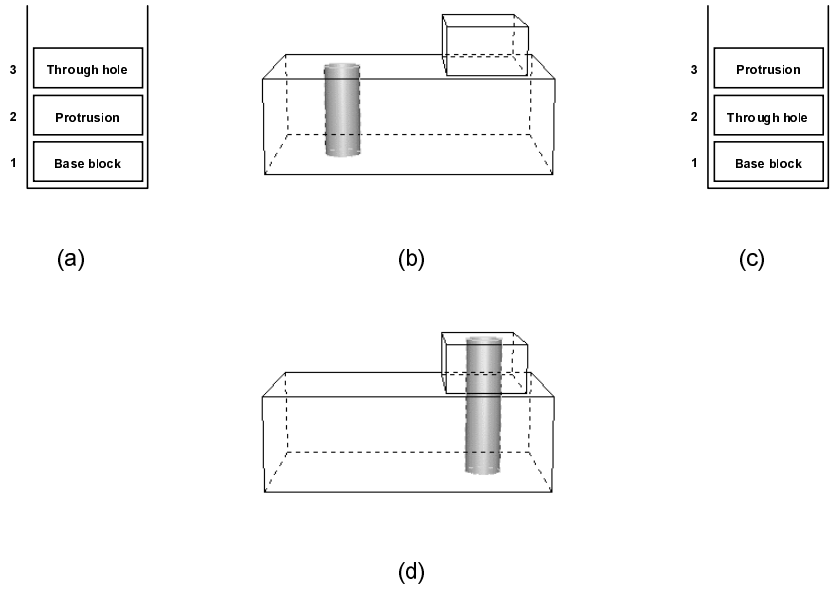


Figure 2.3 – Entity reference problem in history-based model re-evaluation

sists of a block, a through hole and a protrusion, see Figure 2.3.b. The history of this model could be either that in Figure 2.3.a or that in Figure 2.3.c. In the first case, re-attachment of the through hole to the top of the protrusion and the bottom of the block, see Figure 2.3.d, can be achieved by modifying the corresponding attach reference of the through hole in the history, and re-executing that history step. However, if this reference modification would be made in the model history of Figure 2.3.c, re-evaluation of the model would not be possible, because the through hole creation cannot be re-executed with a reference to a face (the top of the protrusion) that will be created in the model at a later stage of its history.

Evaluation of the boundary model by stepwise re-executing a sequence of operations allows each of them to refer *only* to those boundary entities left there by the previous operation. Therefore, modification of the references in a modeling operation, e.g. when re-attaching a feature to other model entities, is not always possible, because the entities concerned may be *tied to a posterior stage* of the model history.

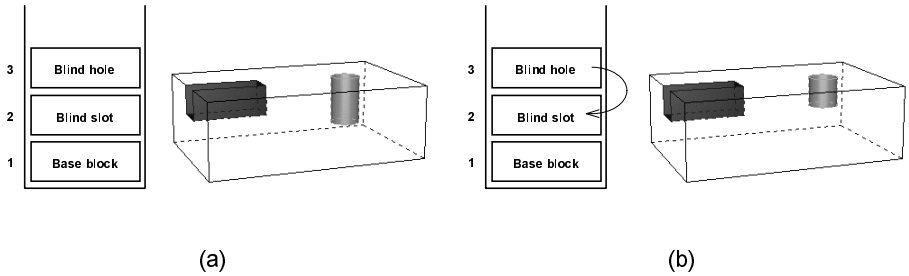


Figure 2.4 – Dimensioning of a model with unidirectional constraints

The fourth shortcoming of current feature modeling systems has to do with the type of constraints that can be used. Constraints can be created in a model, and are thereafter taken into account whenever the history is re-executed. For example, suppose that, after creating the blind hole in Figure 2.4.a, the designer wants to keep its depth equal to that of the blind slot. An algebraic constraint specifying this equality can be created, and the blind hole creation operation will be modified in the history to include a reference to this constraint. When the model history is re-executed, the depth of the blind hole is computed from that of the slot, yielding the desired result, see Figure 2.4.b. However, such constraints are, in most systems, unidirectional. In the example of Figure 2.4, only the blind hole depth is dependent on the slot depth, and not the other way round. This implies that if the blind hole depth is modified in a subsequent modeling operation, the depth of the slot is not adapted accordingly, and is thus no longer equal to the depth of the blind hole. This inability to cope with bidirectional constraints makes the dimensioning of the model undesirably rigid.

The fifth shortcoming of history-based modeling is that the semantics of modeling operations is not always well defined. The main cause of this is the so-called *persistent naming problem*. Each modeling operation uses references to topologic entities in the boundary representation of the current model, which is the combined result of all previous modeling operations. For example, a new feature can be attached to a face or an edge in the boundary representation. A consequence of this is that each operation in the history requires a specific set of topologic entities in the model,

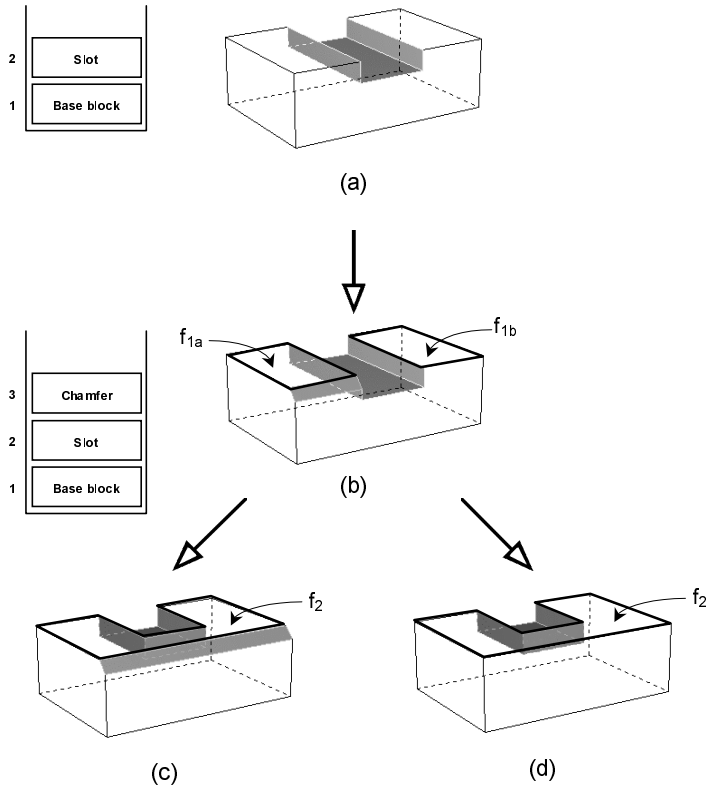


Figure 2.5 – Semantic problem related to entity naming: splitting and merging of faces

also when the operation is re-executed. However, a general property of boundary representations is that topologic entities may be split, merged or deleted as a result of modeling operations. Persistent naming is the process of identifying and tracking topologic entities when a geometric model is modified (Kripac 1995). Although some schemes for persistent naming have been implemented (Kripac 1995, Capoyleas et al. 1996, Lequette 1997), there are some fundamental problems related to this issue, of which two typical examples will now be given. See also (Raghothama and Shapiro 1998) for a formal approach to these problems.

The first example has been taken from Chen and Hoffmann (1995), see Figure 2.5 on the previous page. The model consists of a block to which subsequently a through slot (Figure 2.5.a) and a chamfer (Figure 2.5.b) have been added. The next modeling operation is to change the through slot into a blind slot, causing the two faces f_{1a} and f_{1b} to be merged into one face, f_2. When the model history is re-executed, depending on how the persistent naming scheme works, the chamfer will either be extended along the whole edge (see Figure 2.5.c) or, alternatively, be completely deleted (see Figure 2.5.d). Either result might be the one expected by the user, but he has no control on the choice.

The second example is based on an example given by Lequette (1997). The model consists of a block to which a protrusion has been added, so that their coplanar top faces are merged into one face, f_1, see Figure 2.6.a. Subsequently a through slot, which intersects both the block and the protrusion, has been attached to face f_1, causing it to be split into two faces, f_{2a} and f_{2b}, see Figure 2.6.b. The next modeling step is to slide the protrusion downwards. When the model history is re-executed, depending on how the persistent naming scheme works, the slot will either be changed into a step on the block, Figure 2.6.c, or, alternatively, intrude into the block, Figure 2.6.d.

In both examples, the model resulting from a sequence of modeling operations is in fact determined by the underlying persistent naming scheme. Although the result is *deterministic*, i.e. one will always end up with the same result after the same sequence of modeling operations, it is *ambiguous*, in the sense that it is definitely not always as expected by the user of the modeling system. Stated differently, the semantics of some of the operations is not well defined.

The sixth, and in a way most serious, shortcoming of history-based modeling systems is that they do not maintain feature semantics. Each feature type specifies its own feature creation scheme, possibly including some validation procedure (for example, regarding particular geometric requirements on the attach faces). This procedure is invoked whenever such a feature is created, and is meant to ensure that the operation produces its expected shape imprint. However, such validation procedures are very limited, because they can only analyze a subset of the boundary model, namely the entities involved in the creation operation. All other boundary entities are outside the scope of the operation and cannot, thus, be accessed in this analysis. As a consequence, features previously created in the model can easily be made invalid, i.e. in mismatch with their original validation requirements, without the system being able to detect

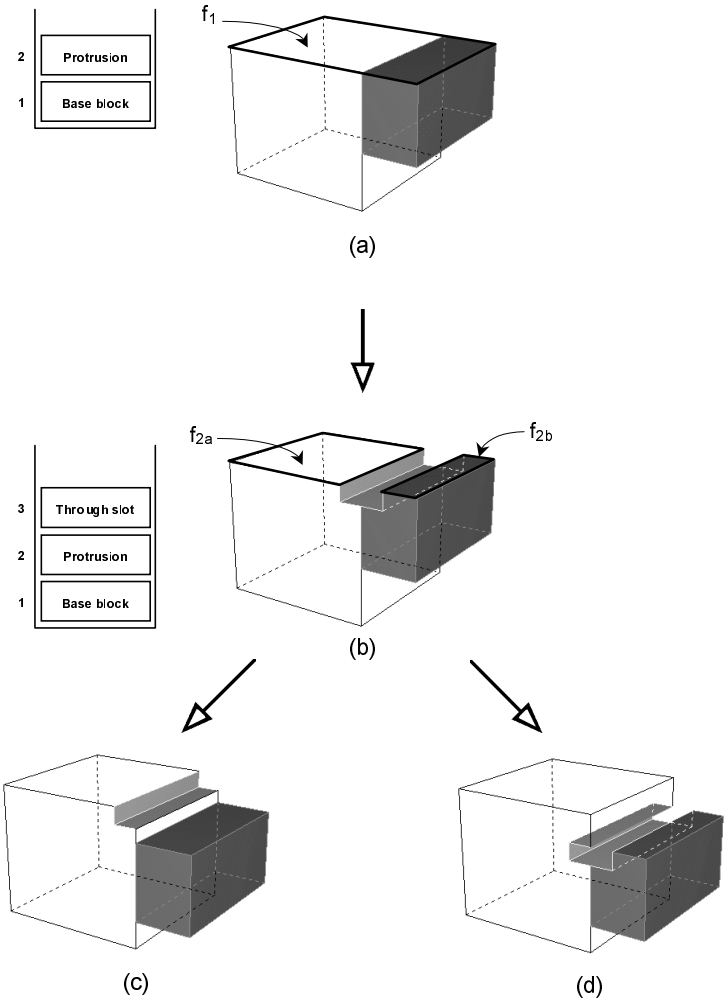


Figure 2.6 – Semantic problem related to entity naming: merging and splitting of faces

this. Systematically analyzing the whole boundary model after each operation is not the solution either, because in its entities there are no (or insufficient) traces of the preceding features.

A first example of problems with changing the semantics of features has already been given in Chapter 1 (Figure 1.1), where two blind holes were turned into through holes. Another example is that of Figure 2.6, in which none of the two results contains a real through slot, with two sides and a fully open top.

These problems are due to the inability of a manifold boundary representation to capture all feature information, e.g. closure faces of subtractive features. This, in turn, excludes the possibility to analyze the topology of the boundary of those features, which is essential to detect and prevent modifications in feature semantics as those illustrated in Figures 1.1 and 2.6.

In fact, history-based modeling systems offer more a geometric modeling approach, to create a boundary representation, than a genuine feature modeling approach. One of the basic ideas of feature modeling is, after all, that functional information can be associated to shape information. This association becomes, however, useless when the shape imprint of a feature, once added to the model with a specific intent, is significantly modified due to a subsequent modeling operation. In other words, arbitrarily modifying the semantics of a feature should be disallowed if one wants to make feature modeling really more powerful than geometric modeling.

So summarized, history-based feature modeling suffers from dimensioning and modeling limitations due to its strong dependency on the chronological order of feature creation and to the use of unidirectional constraints, occasionally suffers from ill-defined semantics of modeling operations, and does not adequately maintain the semantics of features.

A recent attempt to develop an alternative way of modeling, avoiding some of the pitfalls of history-based modeling, is made by Klein (1997a). He proposes to integrate geometric information with other kinds of information, and for this purpose develops a *declarative* geometric modeling approach from a knowledge representation perspective. The possibility to include knowledge in product models makes the approach promising for feature modeling. In this approach, called G-Rep, a solid can be composed of several basic volumes, each basic volume having its own *density*, or ranking. The evaluation of the solid is performed by composition of the basic volumes, based on their densities. The sign of the density at some point in 3D space determines whether it represents material (positive density) or "non-material" (negative density) in the modeled object. Points that lie in more than one basic volume have the density of the

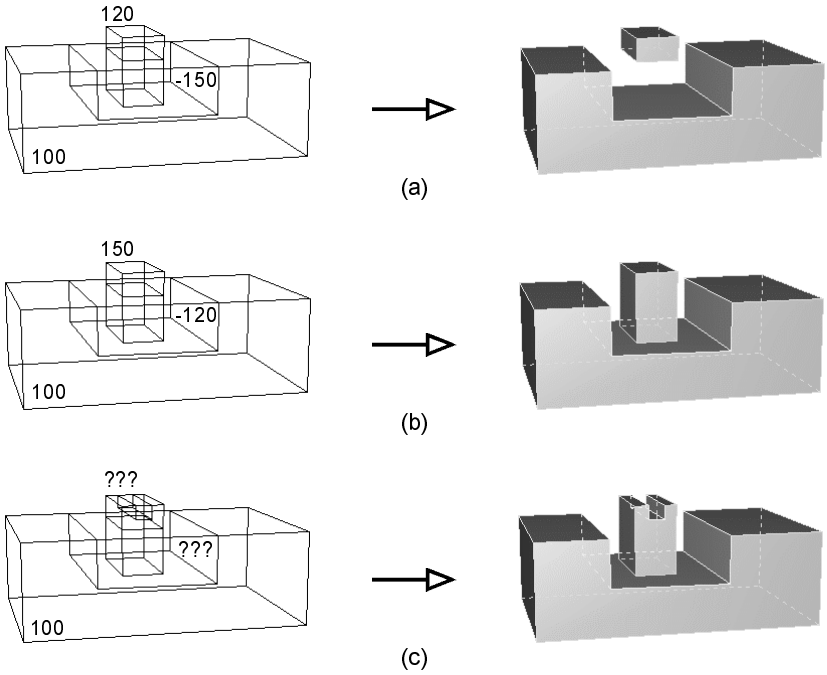


Figure 2.7 – Fixed density problem in G-Rep

basic volume with the highest absolute density value. In this way, evaluation of the geometric model is made independent of the order in which the volumes were added to the model.

A major problem with this approach is, however, that it is not clear how in practice densities have to be assigned to basic volumes. In case each feature class would specify in advance a fixed density for the basic volume of its instances, some model configurations are not feasible, as illustrated in Figure 2.7. When the density of the protrusion basic volume has an absolute value lower than that of the slot basic volume, the model represented is that in Figure 2.7.a, which is not what might be desired. Only if the densities are chosen according to, for example, those in Figure 2.7.b, is it possible to represent the protrusion attached to the slot bottom. No feature class density specification, however, is able to represent the model in Figure 2.7.c, because, in absolute values, one of the two slots

should have a density higher than the protrusion, whereas the other should have a density lower than the protrusion. In order to achieve more flexibility, it is suggested that densities might be configured dynamically, e.g. as a result of modeling operations that alter dependencies between features (Klein 1997b). Such an essential mechanism, however, has so far not been described.

Although certainly interesting, Klein's approach deviates considerably from current approaches to feature modeling. Furthermore, it has not yet been implemented.

We propose a new feature modeling approach that is closer to current practice: *semantic feature modeling* (Bidarra and Bronsvoort 1999b). This approach will be outlined in the next section, and elaborated in the subsequent chapters of this thesis, emphasizing how the problems pointed out so far for current feature modeling approaches are overcome.

2.2 What is semantic feature modeling ?

The semantic feature modeling approach is, just like Klein's approach, declarative. This means that, in contrast to most history-based approaches, feature specification and model maintenance are clearly separated. All properties of features, including their geometric parameters and validity conditions, are declared by means of constraints in a feature class specification. The main advantage of declarative modeling is the freedom in the type and order of constraints that can be specified, and therefore in the way a model can be edited and maintained.

In the semantic feature modeling approach, it is essential that each feature has a well-defined meaning, or *semantics*. This is specified in *feature classes*, which are structured descriptions of all properties of a given feature type, defining a template for all its instances. Such properties include the validity conditions that all feature instances of that type should satisfy. These conditions, as well as the feature shape and its parameters, are specified using a variety of constraint types.

Although most other systems have a rudimentary form of validity conditions too, this approach allows the specification of more powerful ones, which take into account, for example, requirements of a technological and functional character, often dependent on the specific application area. An example of such a validity condition is that the top and bottom face of a through hole should remain open, or, stated differently, that these faces should not be on the boundary of the resulting object. Such

feature validity conditions are in fact indispensable to maintain the semantics of features during the modeling process: without them, features can never be more than high-level geometric modeling primitives.

In our approach, users can define their own feature classes, e.g. by inheriting from an existing feature class and adding some constraints to its definition. Feature classes are stored in feature libraries, from which new features can be instantiated during a modeling session. Feature class specification is elaborated in Chapter 3.

Another characteristic of semantic feature modeling is that the whole modeling process is uniformly carried out in terms of features and their entities (e.g. faces and parameters), and of constraints among these. All modeling actions performed by the user are, thus, effectively *feature-based*, and the same applies to all output, both graphical and textual, generated by the modeling system. An advantage of this is that a feature and, in particular, its faces and their names are persistent. These remain valid and, thus, also all references to them, as long as that feature instance remains in the model. This is in contrast to most history-based modeling approaches, in which references to entities of the evaluated boundary model are kept in the model history, with the drawbacks identified in the previous section; unlike boundary faces in such systems (see examples described on page 14), feature faces are never split, merged or deleted, even though their geometric representation may be.

Probably the most important characteristic of the semantic feature modeling approach is that the semantics of all features is effectively maintained throughout model evolution, for all modeling operations. Some essential aspects of feature semantics in this approach, e.g. the through hole clearance described above, cannot be maintained without an evaluated geometric model, able to consistently represent the whole boundary of subtractive, possibly overlapping, features. In other words, a non-manifold geometric model, containing the relevant feature information, is indispensable to perform effective validity maintenance.

The two characteristics of semantic feature modeling just mentioned lead to a two-level structure in the semantic feature model, clearly distinguishing *modeling entities* from *entities in the evaluated geometric model*. The former, i.e. the entities on which all modeling operations are performed, are kept in the first level of the model –the so-called *Feature Dependency Graph*–, which contains all feature and constraint instances, interrelated by *dependency relations*. The second level contains the evaluated geometric representation of the product in the so-called *Cellular Model*. Its entities are kept internal, being only required to "reflect" the geometry that results from the modeling operations performed on the

first level. The semantic feature model, and mechanisms for maintaining the consistency between both levels, are elaborated in Chapter 4.

It turns out that most changes in the meaning of features are due to modeling operations that cause overlapping features to affect each other's semantics, so-called *feature interactions*, as illustrated in the examples of Figures 1.1 and 2.6. Managing feature interaction phenomena is therefore an essential issue for validity maintenance in the semantic feature modeling approach. Feature interactions are defined and classified in Chapter 5, and mechanisms for their detection in a semantic feature model are presented in Chapter 6.

Maintaining the feature model throughout the modeling process requires not only managing all its constraints, but also monitoring each modeling operation in order to assess the conformity of each feature in the model with its validity criteria. This can guarantee that all aspects of the design intent once captured in the model are permanently maintained. An advantage of maintaining feature model validity in this way is that it becomes possible to provide the user with much better assistance whenever a modeling operation leads to some constraint violation in the model. In particular, explanations on what is causing a constraint violation, and generation of context-sensitive corrective hints, can significantly improve the modeling process. Feature model validity maintenance is elaborated in Chapter 7.

2.3 Prototype system architecture

The semantic feature modeling approach has been implemented in the SPIFF[1] system, a prototype multiple-view feature modeling system developed at Delft University of Technology (Bronsvoort et al. 1997).

SPIFF consists of two main functional subsystems: the *Feature Library Manager* and the *Feature Modeler*. The Feature Library Manager provides interactive facilities for specification of feature classes and for their organization in application-specific feature libraries. These class specifications can be loaded into the Feature Modeler at runtime. The architecture of the Feature Library Manager is described in Section 3.6.

The Feature Modeler provides modeling facilities for creation and manipulation of feature models, according to the architecture depicted in Figure 2.8. Several system modules have been described elsewhere (de

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

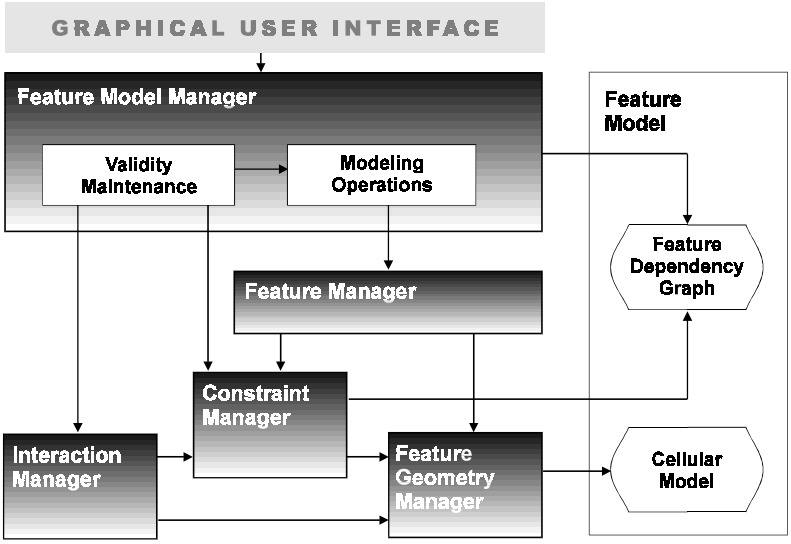


Figure 2.8 – Architecture of the SPIFF modeling system

Kraker et al. 1995, Dohmen et al. 1996, Bidarra et al. 1997), and will be only briefly summarized here.

The *Feature Model Manager* receives commands from the user via a graphical user interface, and translates them into elementary tasks, which are then dispatched to the other Managers. It is responsible for the control of all modeling operations, and for maintaining model validity (see Chapter 7). Furthermore, the Feature Model Manager maintains the *Feature Dependency Graph*, a high-level representation of the structure of the product (see Sections 4.2 and 4.4).

The *Feature Manager* supervises the model processing tasks of each modeling operation, which are actually performed by the *Constraint Manager* and the *Feature Geometry Manager*. The Constraint Manager is responsible for all constraint solving tasks, maintaining all constraints in the Feature Dependency Graph. The Feature Geometry Manager maintains a geometric model of the product in the so-called *Cellular Model*, and takes care of updating it as required by each modeling operation (see Sections 4.3 and 4.5).

The *Interaction Manager* is responsible for the analysis of the Cellular Model, in order to detect any disallowed feature interactions possibly resulting from a modeling operation (see Section 6.1).

3

Specification of feature semantics

"Complete support of user-defined features in a design-by-features system requires that feature classes created by the user become full-privileged members of the feature collection of the system. That is, they can be created, deleted and manipulated; they can have relationships to other features (and these relationships themselves can be defined too); they can be validated by validation constraints or rules (and new validation constraints and rules can be defined); their geometry can be anything that can be described in the underlying geometric modeling system. Clearly, to create a feature definition mechanism that covers all these facilities completely is a challenging task of software engineering (...)."

(Shah and Mäntylä 1995)

This chapter presents the declarative scheme of *feature class specification* developed within the semantic feature modeling approach. This scheme provides a unified description of the shape and validity issues of a feature class, as well as a flexible configuration of the feature class interface (Bidarra et al. 1998a). In this way, feature instances of these classes have powerful and well-defined semantics.

First, other research work dealing with specification of user-defined features is surveyed, and its shortcomings are identified (Section 3.1).

Then, an overview of the proposed scheme is given (Section 3.2). This is elaborated in subsequent sections, distinguishing the feature shape specification issues (Section 3.3), the validity specification aspects (Section 3.4), and the feature class interface presented to the user (Section 3.5). Next, the main functional aspects of the Feature Library Manager are described (Section 3.6), and its use is illustrated with an application example (Section 3.7).

3.1 Introduction

Current feature-based modeling systems provide basic procedures to create a part model using a feature vocabulary. However, this functionality is often hampered by a number of shortcomings:

- when a feature library provides only a fixed set of feature types for use in a model, the creation of complex shapes, e.g. associated with some desired functionality and/or technological process, may become a rather difficult task. Even if this can be achieved by composing several feature instances, the resulting composed shape can only be edited, queried and downstream processed in terms of the elementary instances, because there is no explicit interface defined for the "compound instance" (e.g. its dimension parameters). In addition, properties and validity conditions that were conveniently and meaningfully embedded in each of these elementary features, are of little use for the "compound" shape generated, if not completely undesirable (think, for example, of a cylindrical blind hole class, requiring the bottom face of the shape to be fully present on the part model boundary: one would not be allowed to compose two such instances in order to obtain a stepped-hole-like shape);
- some systems provide a mechanism to record a sequence of modeling steps, possibly in a parameterized way. In this procedural scheme, such macros may later be replayed, in order to create a given "compound feature" in the model. This suffers from the same drawback just described: it is hard to consider a library of such macros as a real feature library, because it does not offer appropriate validity specification mechanisms;
- on the other hand, it is common experience that using predefined feature libraries with a very large set of feature classes is not the solution either, because an exhaustive enumeration of all

possible feature classes is not only unmanageable, but even unfeasible. Furthermore, such sets would vary significantly with the application domain, e.g. the functional requirements of the designer or the technological production processes available.

In order to overcome these drawbacks, declarative schemes are receiving increasing attention. In this section, we survey results of research that deal with mechanisms for specification of user-defined features (UDFs).

Before the first efforts to develop workable UDFs, Dixon et al. (1990) already commented that "if a powerful and convenient capability for user-defined features can be provided, then the library of design-with features can be smaller and the need for combinatorial power is also reduced". They consider this capability to be a very time-consuming, sophisticated and probably not manageable task for the common user of a CAD system. Therefore they suggest that, in the future, CAD system vendors might be required to deliver hard-coded, customized feature libraries to individual customers, according to their specific requirements.

Shah et al. (1994) presented a declarative approach to the description of feature classes, as an alternative to their previous procedural proposal, within the ASU Features Testbed (Shah et al. 1990). A number of primitive geometric constraints is established on feature geometric entities (e.g. faces and edges), in order to define the volumetric shape of the UDF, and is combined in a directed graph. After such a constraint graph template has been stored in the feature library with the feature specification, instantiation of a feature is greatly simplified. From their description, it appears that the explicit geometric representation prevails over the parametric description of the feature, which might turn the definition of complex shapes error-prone and far from accessible for non-specialists. Their work concentrates on the shape definition aspects of a feature class, using geometric and algebraic constraints; in particular, the specification of feature validity issues is not dealt with.

Salomons et al. (1994) focused on interactive definition of new features during incremental modeling of a part, and on their representation by conceptual graphs. They propose combining profile sketching with geometric constraint graph editing, and manual feature identification on the solid model, in order to assist the user in the definition and insertion of new features into the model. A surface representation is used for both features and the part model. Feature models are stored in a hybrid scheme, using a database (for modeler independent data) and modeler files (for geometry-related information), which favors their intended feature recognition applications. Some feature validity issues on such UDFs

have recently been approached, see (Salomons et al. 1998), focusing mainly on dimensional and geometric feature constraints.

Another proposal is that of Hoffmann and Joan-Arinyo (1998), who deal with conceptual definition of UDF prototypes for a feature library. A UDF has a set of constraints and a set of attributes, aimed at specifying the overall shape and validity criteria, respectively. Remarkable in this scheme are the proposed separate treatment of feature attachments, and the incorporation of topological attributes for validation, analogous to the *semantic constraints* first proposed by Bidarra and Teixeira (1994), and elaborated by de Kraker et al. (1995). The procedural definition of each geometric component of the UDF, based on sketching planes, sketching profiles and datum planes, is conceptually very general and powerful, although somewhat laborious for complex shapes, due to the low-level details it is based on.

In short, so far the requirements of complete, flexible and user-friendly specification of UDFs, quoted at the beginning of this chapter, have only been partially met by current proposals. The main difficulties and limitations can be summarized as follows:

- new UDFs are defined and used in a model, but are not always stored as new classes in a feature library for later use;
- the mechanisms to define new feature classes focus on shape aspects, mostly leaving out validity issues;
- the specification of all relevant information needed in a feature class requires non-intuitive and complicated procedures, not easily applicable by common, non-programmer users.

The declarative scheme for the specification of feature classes presented in the remainder of this chapter, not only overcomes these drawbacks, but in fact fulfills all the requirements quoted at the beginning. It has been implemented in the Feature Library Manager of the SPIFF system.

3.2 User-defined feature classes

In Chapter 1, *features* were defined as "representations of shape aspects of a product, that are mappable to a generic shape and are functionally significant for some product life-cycle phase". In other words, each feature has a well-defined meaning, expressed through its geometric, topologic, parametric and functional properties. A *feature class* is a structured

description of all these properties, defining a template for all instances of a given feature type. Such properties include the *validity conditions* that all feature instances of that type should satisfy. Feature classes are sometimes also referred to as *generic feature definitions*.

Feature classes are grouped and stored in *feature libraries*, each of which is suited for a particular application. Ideally, feature libraries should be configurable by domain experts, mostly people without special programming skills, but with knowledge of the requirements, the technology and the criteria of a given application domain. For this, they need a feature library configuration system, providing feature class specification and management facilities. On the other hand, designers, during their modeling activity, are *users* of the feature classes available in feature libraries. For this, they use a CAD system, which performs modeling operations (e.g. create feature instances of selected classes) and takes care of the validity maintenance of the feature model. In this chapter, we concentrate on the specification of feature classes and, thus, on a feature library configuration system such as mentioned above. Issues on validity maintenance of feature models will be dealt with in Chapter 7.

We propose the use of a variety of constraint types on a given shape, to specify a feature class. In the following sections, the use of each of these constraint types will be discussed in detail; here only a brief description is given:

Attach constraints Specify how a feature instance is attached to the model, by coupling some of its faces to faces of other features already present in the model.

Geometric constraints Specify geometric relations, such as parallelism and distance, between feature elements (e.g. faces and datums).

Dimension constraints Specify the set of values allowed for a feature parameter.

Algebraic constraints Specify expressions for feature parameters.

Semantic constraints[1] Specify which topological variants of a feature instance are allowed, by stating the extent to which its feature faces should be on the model boundary.

[1] These constraints might be named *topologic constraints* as well. In the remainder of this thesis, however, we will use the designation *semantic constraint*, for the sake of consistency with previous research work (cf. pages 34 and 73).

Interaction constraints Specify whether a given feature interaction type should be disallowed for a feature instance.

With the scheme proposed here, all feature classes in a library use the same vocabulary, and exhibit a similar structure, see Figure 3.1. This scheme makes no distinction between standard feature classes and user-defined feature classes, in the sense that every feature class can be edited, refined and customized according to specific requirements. This turns out to be the same as saying, with Shah and Mäntylä (1995), that UDFs "become full-privileged members of the feature collection of the system".

The main characteristics of the semantic feature class specification scheme are:

- a powerful specification of feature semantics, comprising validity issues at geometric, topologic and functional levels;
- an inheritance mechanism among feature classes, which makes it trivial to refine and specialize feature classes;
- a clear and simple feature class interface, encapsulating implementation details of the class by means of so-called interface parameters; and
- a translation mechanism, which automatically maps a class description input by a user to a form suitable for use within the modeling system.

The specification schemes for the shape, the validity criteria and the interface of a feature class are elaborated in the following sections. The emphasis here is on the declarative feature class specification scheme, from the viewpoint of the user of the Feature Library Manager. A formal DVM model for this scheme can be found in (Idri 1998), together with syntax details of feature class specifications.

3.3 Feature shape specification

All constraint types introduced above operate on attributes of a *feature shape*. Therefore, the specification of a parameterized shape is the first necessary step in the creation of a feature class. The elementary component for this is the so-called *basic shape*.

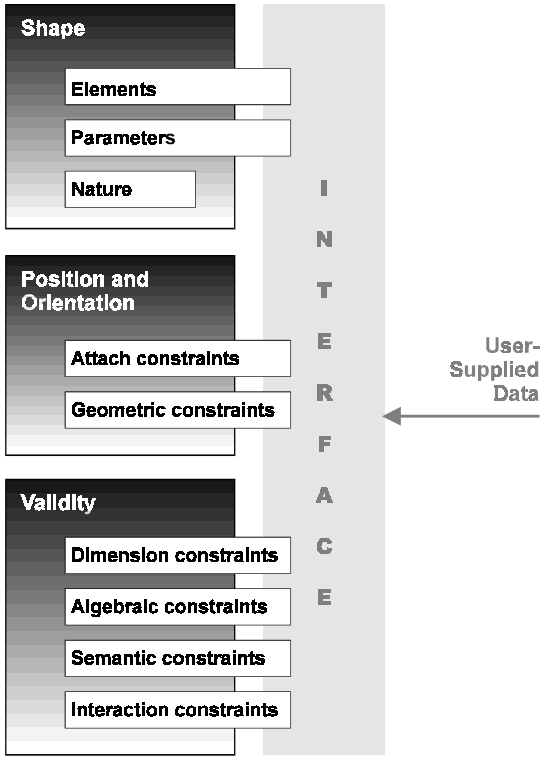


Figure 3.1 – Feature class structure

A basic shape encapsulates a set of geometric constraints that relate its parameters to the corresponding shape faces, see (Dohmen et al. 1996) for details. All basic shape classes have a similar structure (see top of Figure 3.1), with a set of attributes (e.g. parameters, faces, edges, and datum's) and some basic functionality (e.g. initialization and query methods). Figure 3.2 shows three examples of basic shape classes currently available in SPIFF: a rectangular block, a cylinder, and a trapezoidal block. The elements of a feature shape, e.g. the faces, are labeled with generic names. A cylinder shape, for example, has a *top*, a *bottom* and a *side* face. These names are used in all modeling operations. In the sequel,

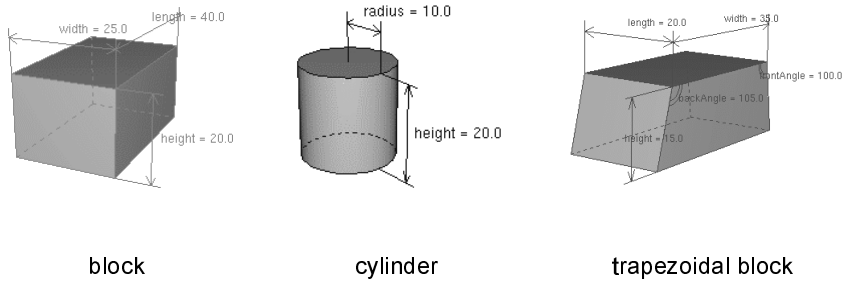


Figure 3.2 – Parameterized basic shapes

for clarity, we will refer to operations on faces only, although operations on other shape elements are also possible.

A feature class associates also to each feature shape the notion of *feature nature*, indicating whether its feature instances represent material added to or removed from the model (respectively *additive* and *subtractive* natures).

There are two mechanisms to use basic shapes for feature shape specification, (i) shape inheritance and (ii) shape composition.

Shape inheritance

Shape inheritance is used if the desired feature shape matches exactly one of the available basic shapes. In this case, the feature class is made to inherit directly from that basic shape class, thus acquiring all its attributes and functionality.

The inheritance mechanism allows one to rename any of the parent shape attributes. In this way, for example, the *height* parameter of a block basic shape could be renamed to *depth*, if the block is used in a rectangular slot or pocket feature class.

Shape composition

If the desired feature shape is not directly available from any of the basic shapes, shape composition is used. For this, several basic shape instances can be specified and geometrically related, possibly overlapping, in order

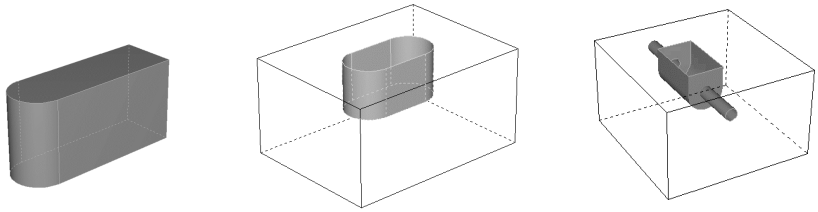


Figure 3.3 – Shape composition examples

to "build" the desired feature shape. With this constructive scheme, a large domain of shapes is achieved, not limited to that of linearly swept profile-based shapes, as in many feature modeling systems. Figure 3.3 shows examples of features whose shapes were obtained by combining instances of block and cylinder basic shape classes only.

In order to achieve this increased shape complexity, basic shapes are encapsulated inside the feature shape, as depicted in Figure 3.4.

The shape composition process consists of four steps:

1. **Selection of a number of basic shapes**. For example, for a *stepped blind hole* class, two cylinder shapes may be chosen, say *cylinder1* and *cylinder2* (see Figure 3.5).
2. **Relative positioning and orientation** of these shapes, applying geometric constraints among their faces. For the *stepped blind hole* example, this can be achieved with two geometric constraints, one aligning the two cylinders (coaxialCylinders), the other setting them contiguously (coplanarCylynders), initialized as follows:

```
coaxialCylinders(cylinder1.axis, cylinder2.axis)
coplanarCylinders(cylinder1.bottom, cylinder2.top)
```

3. **Specification of the compound shape parameters**, as a function of the elementary parameters of the basic shapes. This is achieved by means of algebraic constraints, and should produce a set of parameters that sufficiently determines the dimensions of

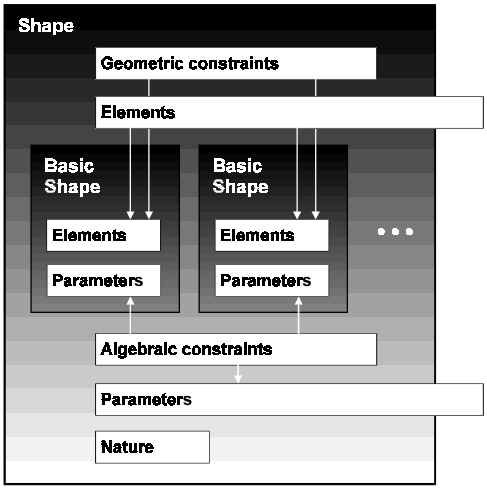


Figure 3.4 – Internal structure of feature shape composition

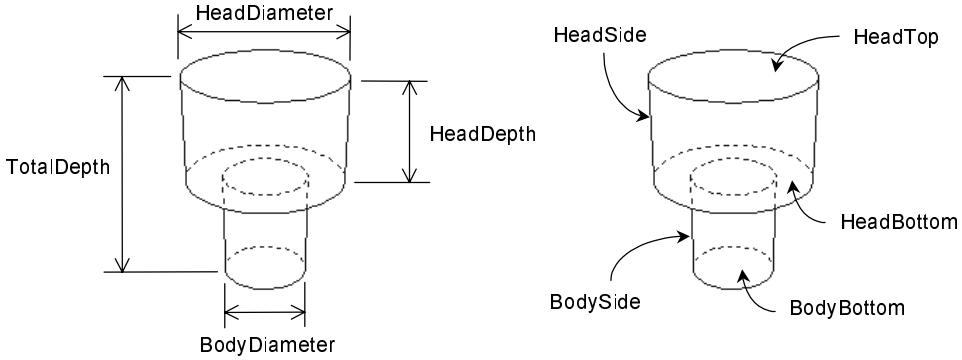
all basic shapes. In the case of the *stepped blind hole* class, such a set could be (see Figure 3.5.a):

```
TotalDepth = cylinder1.height + cylinder2.height
HeadDepth = cylinder1.height
HeadDiameter = 2 * cylinder1.radius
BodyDiameter = 2 * cylinder2.radius
```

4. **Specification of the compound shape faces**, defined in terms of the faces of the basic shapes. These new faces should provide full coverage of the boundary of the compound shape. Again for the *stepped blind hole* example, this could be (see Figure 3.5.b):

```
HeadTop = {cylinder1.top}
HeadSide = {cylinder1.side}
HeadBottom = {cylinder1.bottom}
BodySide = {cylinder2.side}
BodyBottom = {cylinder2.bottom}
```

As depicted in Figure 3.4, both geometric and algebraic constraints, as well as the list of basic shapes, are only internally used in the shape composition process, in order to produce the desired feature shape. Com-



(a) Compound shape parameters (b) Compound shape faces

Figure 3.5 – Compound shape for a *stepped blind hole* class

pound shape parameters and compound shape faces, instead, integrate the feature class interface, and are therefore meant to be referenced later, during modeling, by the user of the feature class. Therefore, the resulting shape has the same interface as in Figure 3.1, presenting a set of shape elements, a set of parameters, and a nature attribute. These characteristics will be further explored in Section 3.5.

3.4 Feature validity specification

Once the specification of the feature class geometry has been completed, it is essential to describe under which conditions the feature instances of that class can be considered valid: the so-called *validity conditions*. Specification of these validity criteria in a feature class is indispensable to perform validity maintenance later, during the modeling process, as discussed in Chapter 7.

Criteria for specifying validity of features can take into account requirements of a technological and functional character, often dependent on the specific application area. It is advantageous if each of those criteria can be specified independently and in a flexible way. In particular, the configuration of a feature library is considerably simplified, and the specialization of new feature classes is made easier.

Validity conditions can be classified into three categories: geometric, topologic and functional. These are now elaborated.

Geometric validity specification

One way of constraining the geometry of a feature class is by specifying the set of values allowed for a shape parameter. Examples of such specifications are an enumerated set of values and a range defined by upper and/or lower bounds. We use *dimension constraints* applied on shape parameters. For instance, the radius parameter of a given through hole class could be limited to values between 1 and 10.

Feature shapes can also be geometrically constrained by means of explicit relations among their parameters. These relations can be simple equalities between two parameters (e.g. between *width* and *length* of a square section passage feature) or, in general, algebraic expressions involving two or more parameters and constants. For this, we use *algebraic constraints*, similar to those used for shape composition in the previous section.

Topologic validity specification

As described in Section 3.3, the specification of a feature shape yields a set of shape faces providing full coverage of the boundary of a volumetric feature. However, for most features, not all these faces are meant to effectively contribute to the boundary of the modeled part. Some faces, instead, have a closure role, delimiting the feature volume without contributing to the model boundary. The specification of such properties is called topologic validity specification.

To specify topologic validity in a feature class, we use *semantic constraints* on each shape face (Bidarra and Teixeira 1994). Semantic constraints are of two types: *onBoundary*, which means the shape face should be present on the model boundary, and *notOnBoundary*, which means the shape face should not be present on the model boundary. Furthermore, both types of constraints are parameterized, stating whether the presence or absence on the model boundary is *completely* or only *partly* required. An example of this is a blind hole class for which the top face has a *notOnBoundary (completely)* constraint, the side face has an *onBoundary(partly)* constraint, and the bottom face has an *onBoundary(completely)* constraint.

Functional validity specification

Geometric and topologic validity specifications alone, as described above, are unable to fully describe several other functional aspects that are inherent to a feature class as well. These are better described in terms of the feature volume or feature boundary as a whole, and therefore require a higher-level specification, not directly based on shape parameters or faces. An example of this is the requirement that every feature instance of a class should somehow contribute to the shape of the part model. Another example, from a technological perspective, is the requirement for clearance of tool entrance faces of subtractive features.

Functional requirements can be violated by feature interactions caused during incremental editing of the model. *Feature interactions* are modifications of the shape aspects represented by a feature that affect its functional meaning. Feature interactions are dealt with in Chapter 5, where a classification of interaction types is presented, and in Chapter 6, where their detection is described.

We propose the specification of *interaction constraints* in a feature class in order to indicate that a particular interaction type is not allowed for its instances (Bidarra et al. 1997). Examples of these are the requirement that a subtractive feature instance should not become a closed void inside the model (no *closure interaction*), and the requirement that a feature instance should somehow contribute to the shape of the part model (no *absorption interaction*).

3.5 Feature class interface specification

An advantage of the scheme presented so far is that, during the specification of a feature class, control can be provided on which feature components are made public, i.e. accessible to the user of the modeling system when manipulating its feature instances, and which should be kept private inside the feature class. The latter have either an internal role, e.g. in supporting the shape specification as described in Section 3.3, or a fully specified character, in the sense that all parameters they require to be initialized have already been specified in the class. An example of the latter is a dimension constraint stating that the parameter *width* of a slot class should have a value greater than 3 mm: such a constraint is always initialized on parameter *width* and with value *3*, requiring no user input at feature instantiation stage.

On the other hand, several feature constraints and parameters may require external data to be provided at feature instantiation stage –the so-called *user-supplied data*–, as depicted in Figure 3.1. Those feature members constitute the *feature class interface*. The specification of the feature class interface determines how feature instances will be presented to the user of the modeling system and, thus, how the user will be able to interact with them. In the SPIFF system, all interface parameters have a name, which will be used later to designate them at the graphical user interface of the modeling system, when the feature is instantiated or modified.

Essential in the feature class interface is the positioning and orientation scheme, which is specified by means of attach and geometric constraints, as depicted in Figure 3.1.

An attach constraint of a feature couples one of its faces to a user-supplied feature face, to be chosen among those of the features already present in the model. Attach constraints are a kind of coplanar geometric constraints that take into account the natures of the two features involved in order to determine the appropriate normal orientations (Dohmen et al. 1996).

Geometric constraints position and orient a feature relative to (faces of) other features already present in the model, by fixing its remaining degrees of freedom. For this, a geometric constraint couples one of the feature faces to a user-supplied feature face in the model, possibly with some additional numeric parameter(s). For instance, to position a through slot, a *distanceFaceFace* constraint might be used, which requires an external reference feature face and a distance value.

Because feature faces, instead of faces of the boundary representation, are used to attach and position a new feature to the model, ambiguities caused in history-based systems by the persistent naming problem (see Section 2.1) are avoided.

Some shape parameters may be determined implicitly from the feature attachments, e.g. the depth of a through hole or the length of a through slot. All other shape parameters need a user-supplied value at feature instantiation stage, and are therefore also included in the feature class interface.

Feature validation constraints may also take part in the interface of a feature class, when some of their parameters are left unspecified until the creation of a specific feature instance. Examples of this are user-supplied data aimed at (i) initializing the bounds of an interval of some

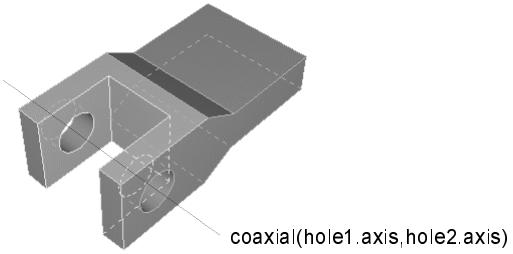


Figure 3.6 – Aligning features by switching off positioning geometric constraints

dimension constraint, (ii) determining whether an *onBoundary* semantic constraint should be configured as *partly* or *completely*, or (iii) setting the constant proportionality factor between two shape parameters in some algebraic constraint.

Finally, it is possible to declare some interface parameters as *switchable*. This means that, whenever desired during the modeling session, these can be deactivated, and therefore not taken into account in the validation process. While this possibility is not likely to be used for most feature validity constraints, it is particularly convenient for shape parameters and for positioning geometric constraints. In this way, they may be overruled by additional geometric constraints explicitly created later (so-called *model constraints*, see Chapter 4). As an example of this, consider the fixture part in Figure 3.6. Although the two through holes may have been positioned independently, each one using a different pair of reference model faces (e.g. the block front and top faces), it might be desirable to keep them aligned. This can be achieved by switching off the positioning geometric constraints of one of them, and replacing these by a coaxial constraint between the axes of the two through holes.

3.6 The Feature Library Manager

The Feature Library Manager of SPIFF comprises a Feature Class Manager and a Graphical User Interface (Idri 1998), as depicted in Figure 3.7. The Feature Class Manager in turn comprises a Parser and a Generator to process Class Models. Through the Graphical User Interface, the user may create a new feature class, or modify an existing one.

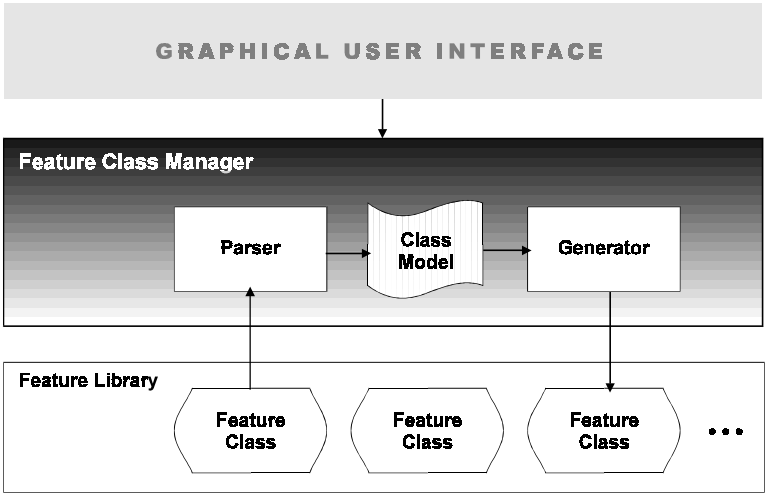


Figure 3.7 – Architecture of the Feature Library Manager of SPIFF

The system provides two methods to create a new feature class. In the first method, the feature class is specified from the outset: the Feature Class Manager incrementally builds the Class Model, as the user follows the steps outlined in Sections 3.3 to 3.5. In the second method, existing feature classes can be used as a basis for the new class. Two cases are then distinguished.

In the first case, the new feature class inherits from a single feature class. Therefore, the new class gets the same shape and validity conditions as its parent. Its specification can then be completed by defining additional validity conditions and a new interface (see Sections 3.4 and 3.5). An example of this is the creation of a square passage class based on a rectangular passage class, by just adding an algebraic constraint that specifies its length and width parameters to be equal.

In the second case, the new feature class is composed from multiple existing feature classes. Here, the positioning geometric constraints of the component classes are used to compose the shape of the new class, instead of explicitly building it with additional constraints, as when the class is defined from the outset. An advantage of using other feature classes to compose (the shape of) a new class is that both components

with additive and components with subtractive natures can be used. The other specification steps are similar to those used for building classes from the outset, as outlined in Sections 3.4 and 3.5.

When an existing feature class is selected to be modified, it is read by the Feature Class Manager and parsed in order to build its Class Model. After that, the class can be interactively modified by the user, through the Graphical User Interface.

After a feature class has been fully specified, it is stored in a Feature Library, by means of the Generator. This module translates the Class Model into a class in LOOKS, an object-oriented imperative programming language (Peeters 1993). SPIFF contains a LOOKS interpreter, into which feature libraries are loaded and, thus, made available in the modeling system.

Eventually, the Feature Library Manager validates a feature class just specified, in order to avoid over- and underconstrained specifications. This validation process roughly performs a short "modeling session": a prototype feature instance is created with the necessary user-supplied parameters, and attached to a basic block feature. The constraint solver of the modeling system then checks whether a well-constrained situation is achieved (Noort 1997). Possible conflicts with algebraic and geometric constraints are graphically reported. Such a scheme is useful to fix many inconsistent and incomplete geometric/algebraic constraint specifications. It is, however, far from exhaustive because, among other reasons, it tests only one feature prototype instance, and the concrete set of user-supplied parameter values determines the constrained model behavior. Also, in the current implementation, no checking is made on redundant or conflicting dimension and semantic constraints.

The Graphical User Interface of the Feature Library Manager will be illustrated in the next section.

3.7 Application example

This section describes the use of the Feature Library Manager for the specification of a *rounded blind slot* feature class; see Figure 3.8.

To describe the shape for the *rounded blind slot* class, two basic shapes are specified: block *b* and cylinder *c*. These are then positioned relative to each other according to the following scheme:

1. Align the two shapes vertically and give them the same height, by making their top and bottom faces coplanar, with two *coplanar*

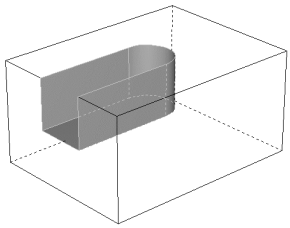


Figure 3.8 – Example of a rounded blind slot feature instance

geometric constraints, *alignTop* and *alignBot*, initialized as follows (see Figure 3.9.a):

```
alignTop(b.top, c.top)
alignBot(b.bottom, c.bottom)
```

2. Make the diameter of the cylinder equal to the width of the block, with an algebraic constraint (see Figure 3.9.b):

```
b.width = 2 * c.radius
```

3. Position the cylinder at the middle of the back face of the block, with two *distLineFace* geometric constraints, *cylinderDist1* and *cylinderDist2*, initialized as follows (see Figure 3.9.c):

```
cylinderDist1(c.axis, b.right, c.radius)
cylinderDist2(c.axis, b.back, 0)
```

The *rounded blind slot* dimension parameters are then defined in terms of the basic shapes' parameters, using algebraic constraints as follows (see Figure 3.9.d):

```
length = b.length + c.radius
width = b.width
depth = b.height
```

Similarly, the *rounded blind slot* boundary faces are defined in terms of the basic shapes' faces as follows:

```
top = {b.top, c.top}
bottom = {b.bottom, c.bottom}
```

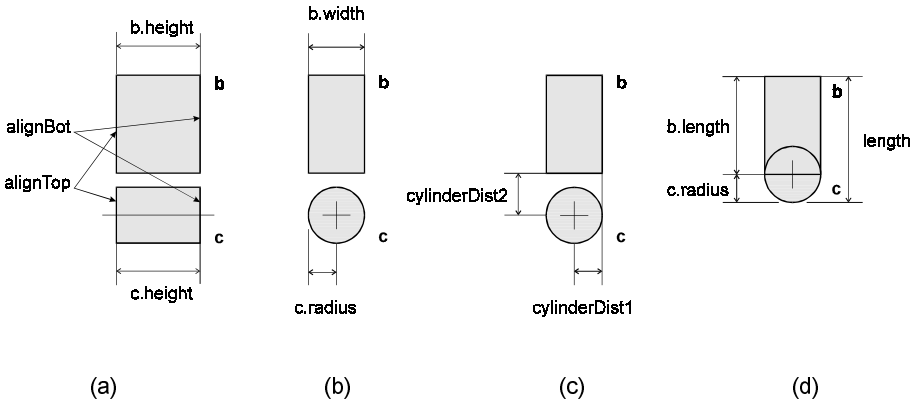


Figure 3.9 – Shape specification for the *rounded blind slot* class

```
front = {b.front}
back = {c.side}
left = {b.left}
right = {b.right}
```

The *rounded blind slot* needs two attach constraints, *attachTop* and *attachFront*, on its top and front faces, each of them requiring a user-supplied model reference face, *modelFace1* and *modelFace2*:

```
attachTop(top, modelFace1)
attachFront(front, modelFace2)
```

Finally, the position becomes fully specified with a *distFaceFace* geometric constraint setting the user-supplied distance *d*, between one of the *rounded blind slot* side faces and a user-supplied model reference face, *modelFace3*:

```
position(left, modelFace3, d)
```

Several validity conditions can now be specified for the *rounded blind slot* feature class. For example:

1. To preserve the desired feature shape properties, the block should be prevented from degenerating (that would be the case if

the length of the *rounded blind slot* were smaller than the cylinder radius); similarly, the *rounded blind slot* width and depth should be restricted to positive values. This is achieved by setting lower bounds for these parameters with dimension constraints, initialized as follows:

```
dimensionLength(length, c.radius)
dimensionWidth(width, 0)
dimensionDepth(depth, 0)
```

2. Faces top and front of the *rounded blind slot* should not be on the model boundary, whereas each of the remaining faces should have, at least, some contribution to the model boundary. For this, the following semantic constraints declarations are made:

```
semanticTop(top, notOnBoundary, completely)
semanticFront(front, notOnBoundary, completely)
semanticLeft(left, onBoundary, partly)
semanticRight(right, onBoundary, partly)
semanticBack(back, onBoundary, partly)
semanticBottom(bottom, onBoundary, partly)
```

3. Closure and absorption interactions could be disallowed for *rounded blind slot* instances, by including the respective interaction constraints in the class specification.

The final specification of the *rounded blind slot* feature class is shown in Figure 3.10.

3.8 Conclusions

Current feature class definition schemes, surveyed in Section 3.2, are very limited, mainly due to either excessive low-level input requirements, or incomplete specification of feature validity.

The declarative scheme presented in this chapter overcomes these drawbacks, providing a very flexible mechanism for the specification of feature classes. It supports a wide range of shapes, and allows full specification of validity conditions at geometric, topologic and functional levels, by means of a variety of constraint types. This approach has been implemented in the Feature Library Manager of the SPIFF system, which provides a graphical user interface, and thus requires no programming skills.

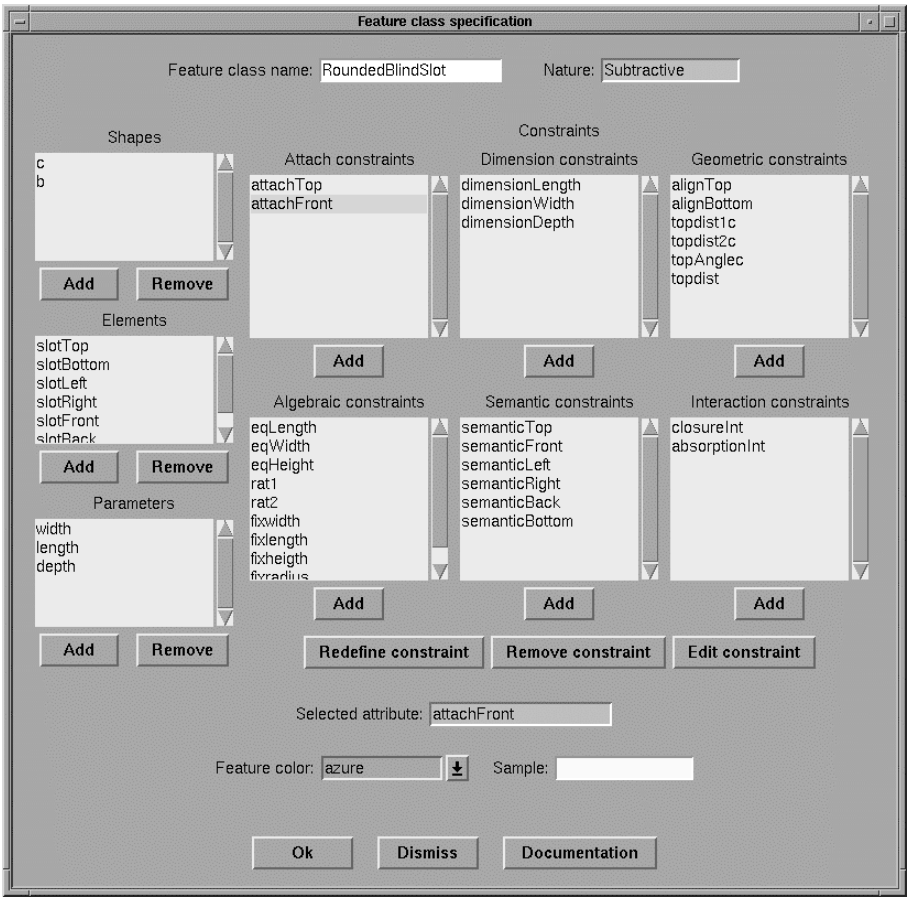


Figure 3.10 – Final specification for the *rounded blind slot* class

Feature classes created in this way have a simple interface, facilitating the creation and manipulation of their feature instances during the modeling process.

4

The semantic feature model

> *"If the designer wishes to create a hole, a slot, or a pocket, then (i) he should be able to design precisely in those terms, and (ii) the system should retain the information that a particular set of topological entities belongs to a particular part feature. (...) What is really needed is the ability to automatically link face sets upon initial creation of the feature. (...) The explicit presence of feature information in the modeler's database will greatly help in the generation of process plans (...)."*
>
> *(Pratt 1984)*

This chapter describes the *semantic feature model*, on which the semantic feature modeling approach, outlined in Chapter 2, is based. First, the important notion of *dependency* between model entities is introduced (Section 4.1). Next, the two levels integrated in the feature model –the *Feature Dependency Graph* and the *Cellular Model*– are elaborated (Sections 4.2 and 4.3). Finally, mechanisms for model maintenance are presented for both levels (Section 4.4), illustrating with typical examples how the feature model evolves throughout a modeling session.

4.1 The dependency relation

Many researchers in feature modeling have pointed out the convenience of keeping track of the model structure in terms of the relations among its features, in addition to that provided by the low-level, evaluated geometric model. A variety of structures has been proposed, expressing attachment, adjacency, connectivity or similar relationships among the features of a model. Some of them were based on a rigid, CSG-like, parent-child relationship, yielding a tree-structured model; many others adopted a general graph structure, in order to better capture feature relations in more complex models (Kyprianou 1980, Henderson 1984, Luby et al. 1986, Shah and Rogers 1988, van Emmerik and Jansen 1989, Bronsvoort and Jansen 1993).

More research effort has been dedicated to the nodes in such graphs (the features) rather than to the edges (the relations between them). So, for example, issues like *"which features are in the model?"* or *"how should they be represented?"* have received more attention than issues such as *"which relationship is there among those features?"* or *"how can this relationship be precisely defined and represented?"*.

The dependency relation defined in this section provides a sound basis for the latter. In particular, it is clearly defined, has rich semantics, and has direct application in feature model maintenance.

Dependencies among features

As described in Section 3.5, instantiation of a new feature requires the user to supply a set of parameter values, aimed at initializing all its constraints and parameters. Some of these values consist of references to elements of other features (e.g. faces), and are meant to specify how the new feature should be attached and positioned relative to the features already present in the model. In accordance with the requirements introduced in Section 2.2, such references are persistent, in the sense that they remain valid as long as the features referred to remain in the model.

Moreover, these references establish a clear dependency among the features in the model. Thus, for example, if a blind hole is attached to the bottom of a slot, see Figure 4.1.a, it will be displaced when the depth parameter of the slot is increased, see Figure 4.1.b. Also, the blind hole attachment has to be readjusted, or, alternatively, the blind hole itself removed, when the slot feature is removed from the model.

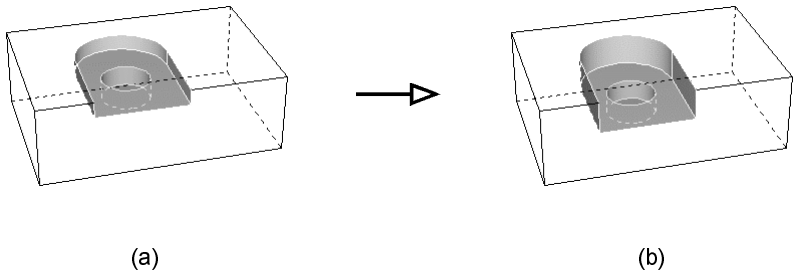


Figure 4.1 – Example of a dependency relation between features

A dependency between two features is unidirectional: one can always distinguish the feature that is determined from the feature that determines. In this sense, again referring to the example of Figure 4.1, removal of the blind hole from the model does not present any problem to the slot feature.

We can therefore state that feature f_1 *directly depends* on feature f_2 whenever f_1 is attached, positioned or, in some other way, constrained relative to f_2. Stated differently, f_1 directly depends on f_2 if some feature constraint of f_1 has a reference to an entity of f_2.

By extension, a feature is considered to depend on another feature if the above definition recursively applies between them: feature f_1 *depends* on feature f_2 whenever f_1 directly depends on some feature f_3 that *depends* on f_2. Finally, two features are said to be *independent* if and only if none of them depends on the other.

Dependencies between constraints and features

In feature modeling, it is very convenient to be able to define, besides the constraints in feature classes, constraints among the feature instances in the model, with the goal of further specifying design intent. Constraints for this can be of any type mentioned in Section 3.2, and are called *model constraints*.

An example of the use of model constraints has already been presented in Figure 3.6. Another example is given in Figure 4.2. The slot and the passage features, which were independently positioned relative to the block side faces (see Figure 4.2.a), are repositioned and aligned by

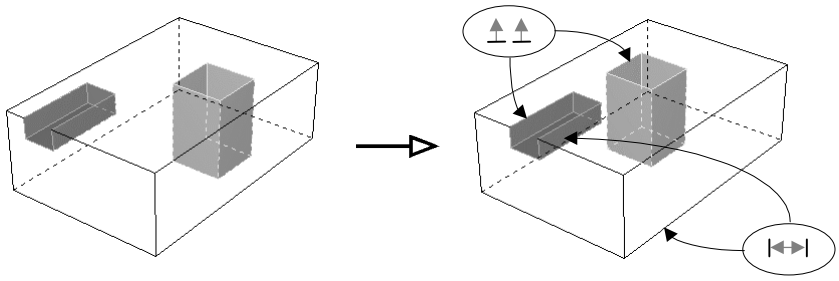


Figure 4.2 – Relative repositioning of features by means of model constraints

means of two geometric constraints: one requiring the left faces of the two features to be coplanar, the other setting the distance of the slot, and thus also of the passage, to the block right face (see Figure 4.2.b).

This example illustrates three important properties of model constraints:

1. Unlike the feature constraints used throughout Chapter 3, model constraints are model entities comparable to features: they are created, edited, maintained and removed from the model in a similar way.
2. They have a multidirectional character among the features they refer to: a modification in *any* of them is always propagated to *all* the others. In the example of Figure 4.2.b, regardless of which of the two features is positioned by the distance constraint (relative to the block right face), the coplanar constraint will always cause the other feature to "follow" it (compare this with the limitation pointed out on page 12).
3. They mostly overrule some feature constraints previously specified. In the example of Figure 4.2.b, the two model constraints prevail over the original positioning constraints of the two features relative to the block sides. If this would not be done, an overconstrained situation would arise.

A logical consequence of properties (1) and (2) is that model constraints, regarded as model entities, depend on the features they refer to. By analogy to feature-feature attachments, we say that they are *attached* to those features. Because of this dependency, it is impossible, for example, to remove either the slot or the passage from the model in Figure 4.2.b without, at the same time, removing, or at least modifying, the model constraints attached to it.

Similarly to what has been defined for features, we can now state that a model constraint *c depends* on a feature *f* whenever *c* is attached to *f*.

The notion of dependency plays a crucial role in the semantic feature model, described in the next sections. It is a dynamic relation among modeling entities, and can thus evolve as these entities are modified, in contrast to the static chronological feature creation order used in most history-based feature modeling systems.

4.2 The Feature Dependency Graph

The *Feature Dependency Graph* contains all *feature instances* in the product model, each of them with its own set of entities (e.g. shape, parameters and constraints), and all *model constraint instances*. These instances are interrelated by the dependency relations introduced in the previous section, yielding a directed acyclic graph structure, consisting of the set of all model entities (feature instances and model constraint instances), and the set of dependency relations among these entities. Each edge represents one dependency relation, and is oriented towards the dependent feature or model constraint. As an example, Figure 4.3 on the next page depicts the Feature Dependency Graph of the model in Figure 4.2.b.

The goal of the Feature Dependency Graph is to provide a high-level structure of the feature model. In fact, it contains *all* entities and information required for model manipulation, in a structured way. Interaction between the user of the modeling system and the model takes place in terms of the features and model constraints in the Feature Dependency Graph. Also, all modeling computations are primarily carried out at this level. For example, all geometric and algebraic constraint solving tasks act upon entities at this level. Each entity in the Feature Dependency Graph may be queried about its current parameter values and dependen-

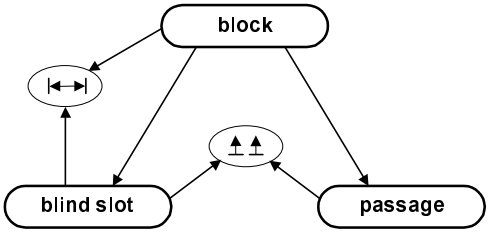


Figure 4.3 – Feature Dependency Graph of the model in Figure 4.2.b

cies. Furthermore, each feature node in the graph "knows" about its current global position, as well as its geometry.

The Feature Dependency Graph contains no evaluated model geometry, but instead all information necessary to generate and maintain this in the Cellular Model, as will be described in the following section.

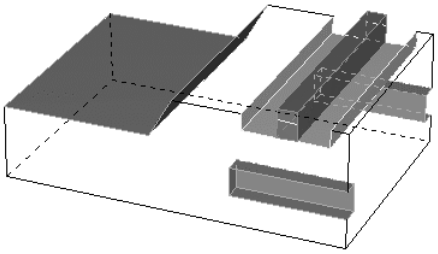
4.3 The Cellular Model

The *Cellular Model* is a non-manifold representation of the feature model geometry, integrating the contributions from all features in the Feature Dependency Graph. The Cellular Model is presented in detail in (Bidarra et al. 1998b), together with a survey on other research proposals for the geometric representation of feature models.

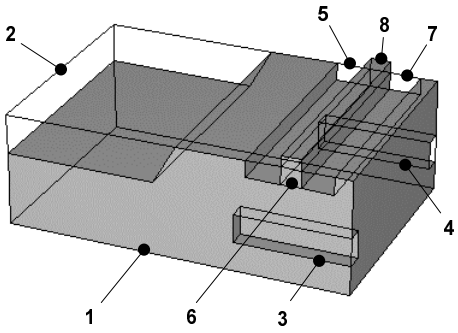
The geometry of each feature instance, designated the feature's *shape extent*, accounts for the bounded region of space comprised by its volumetric shape. Moreover, its boundary is decomposed into functionally meaningful subsets, the *shape faces*, each one labeled with its own generic name, as described in Section 3.3.

The Cellular Model represents a part's geometry as a connected set of volumetric quasi-disjoint *cells* of arbitrary shape, in such a way that each one either lies entirely *inside* a shape extent or entirely *outside* it. The cells represent the point sets of the shape extents of all features in the model. Each shape extent is, thus, represented in the Cellular Model by a connected subset of cells.

Furthermore, the cellular decomposition is interaction-driven, i.e. for any two overlapping shape extents, some of their cells lie in both shape extents (and are called *interaction cells*), whereas the remaining cells lie



(a) part



(b) Cellular Model

1 <block>
2 <block, step>
3 <block, blind slot 1>
4 <block, blind slot 2>
5 <block, through slot>
6 <block, through slot, rib>
7 <block, through slot>
8 <rib>

(c) cell owner lists

Figure 4.4 – Cell owner lists in the Cellular Model

in either of them. As a consequence of this, two cells can never volumetrically overlap. They may, however, be adjacent, in which case there is an interior face of the Cellular Model separating them. Such a face can be regarded as having two "sides", designated as partner *cell faces*. A face that lies on the boundary of the Cellular Model has only one cell face (one "side"), that of the only cell it bounds. In either case, a cell face always bounds one and only one cell. Each shape face is, thus, represented by a connected set of cell faces.

In order to be able to search and analyze features and their faces in the Cellular Model, each cell has an attribute –called *owner list*– indi-

cating which shape extents it belongs to, see Figure 4.4. Similarly, each cell face has also an owner list, indicating which shape faces it belongs to.

Just like for features, the *nature of a cell* expresses whether its volume represents "material" of the part or not. Its determination will be precisely described in the next section. For example, in the model of Figure 4.4, cells 1, 6 and 8 have additive nature (i.e. the nature of either the block or the rib), whereas all other cells have subtractive nature (i.e. that of a subtractive feature in their owner lists). Similarly, the *nature of a cell face* expresses whether it lies on the boundary of the part or not.

The Cellular Model, including its attribute mechanism to maintain and propagate the owner lists of cells and cell faces, was implemented using the Cellular Topology husk of the Acis Geometric Modeler (Spatial 1998).

4.4 Feature Dependency Graph maintenance

Feature model maintenance is the process of updating the feature model, according to the requirements of each modeling operation. It is performed at both levels of the feature model: first, the Feature Dependency Graph is modified and analyzed; next, the Cellular Model is updated accordingly and analyzed. The model modification process at the Feature Dependency Graph level is described in this section. Maintenance at the Cellular Model level is described in the next section. The analysis mechanisms will be discussed in Chapter 7, in the broader context of model validity maintenance.

Modeling operations can be grouped into two major categories: *feature operations* and *model constraint operations* (or simply *constraint operations*). Feature operations include the following:

Adding a new feature instance to the model This operation creates a new feature instance of the chosen feature class, and requests from the user a full set of initialization parameter values for the new feature. Together with this, all constraint members specified in its class are also instantiated, and initialized with the corresponding user-supplied parameter values (e.g. distance parameters and external feature faces for attach and positioning constraints, see Section 3.5).

Editing a feature instance in the model This operation permits modifying *any* feature interface parameter value provided earlier to that feature instance.

Removing a feature instance from the model This operation removes from the model the feature and all feature constraints instantiated at its creation stage.

Constraint operations are similar to feature operations: model constraints can be added, modified and removed. They are, however, most often specified and executed in "batch form" for user convenience: several new model constraints can be added to the model in one step, and existing model constraints modified or removed, while at the same time some feature constraints can be selected to be switched off, in order to avoid geometrically overconstrained situations (see Figure 4.2.b and property (3), on page 48, for an example).

In Chapter 7 the generic scheme of modeling operations will be analyzed in detail, distinguishing in them a number of steps (Figure 7.1, on page 102). In the current context, it is enough to refer to those steps responsible for the modification of the feature model.

The first step for all modeling operations (except for feature removal operations) is the internal geometric and algebraic constraint solving process. When this process is successfully performed, all feature instances in the Feature Dependency Graph have their parameters, position and (shape extent) geometry updated. The solving process also records which features have actually been geometrically modified by the modeling operation.

The Feature Dependency Graph is updated according to the specificity of each modeling operation:

Adding a new feature instance to the model The new feature instance is added to the Feature Dependency Graph, and all its dependencies stored, according to the user-supplied interface parameters values.

Editing a feature instance in the model The modified feature instance is kept in the Feature Dependency Graph. All its feature constraints are adjusted as required by the operation, possibly modifying some of the feature dependencies (as for example in a re-attach operation).

Removing a feature instance from the model The feature is simply removed from the Feature Dependency Graph. The re-

moval operation is, however, *only* allowed if it has no dependent entities (features or constraints) in the Feature Dependency Graph; otherwise, the user is given the possibility of modifying these in order to eliminate their dependencies (e.g. by changing their attachments, see Chapter 7).

Constraint operations All new model constraints are added to the Feature Dependency Graph, according to their *attach* dependencies. Similarly, modified and removed model constraints are also updated in, or removed from, the Feature Dependency Graph.

4.5 Cellular Model maintenance

Another step of a modeling operation consists of updating the Cellular Model, so that changes in the Feature Dependency Graph are also reflected in the geometric model. This step is essential in order to check semantic and interaction constraints, which are concerned with the concrete geometry and topology exhibited in the Cellular Model by the features involved in the operation (that process, called *feature interaction detection*, is elaborated in Chapter 6).

For each modeling operation, this step is carried out in two phases. In the first phase, the Cellular Model is incrementally re-evaluated; this is described in the remainder of this section. In the second phase, the Cellular Model is interpreted, according to the feature information stored in its cellular entities and the current dependencies among the features; this is discussed in the next section.

In accordance with the goals stated in Section 1.2, these two phases are aimed at satisfying the following essential requirements:

1. The process of re-evaluation of the Cellular Model, after each operation, should be limited in scope, in order to keep its computational cost independent of the number of features present in the model.
2. The evaluation and interpretation of the Cellular Model, corresponding to the structure of the Feature Dependency Graph, should be completely and unambiguously determined without invoking any model history considerations.

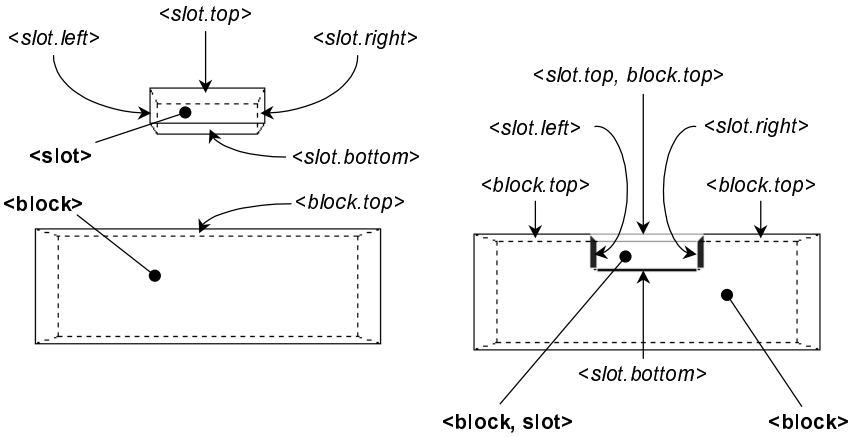
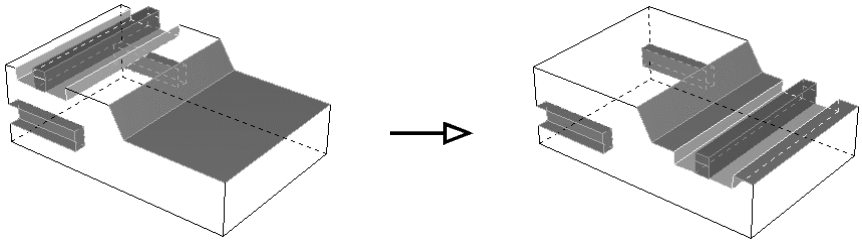


Figure 4.5 – Propagation of owner data in a cellular union operation

Cellular Model incremental evaluation

In contrast with history-based systems, which use two non-associative set operations (union and difference) to evaluate the geometric model (see example on page 10), in the semantic feature modeling approach *only one set operation* is used to evaluate the Cellular Model: it is computed by performing the *non-regular cellular union* of the shape extents of all features. Because it is a union operation, the order in which the shape extents are processed is irrelevant for the final Cellular Model obtained. By these non-regular cellular operations, between (the single cell representing) each shape extent and the other cells generated so far, the cellular decomposition described in Section 4.3 is computed. Essential in this process is the correct propagation of the owner lists of each cell and cell face when these are further decomposed, so that each entity "knows" precisely which shape extents, or shape faces, it belongs to.

A simple example of a non-regular cellular union operation is given in Figure 4.5, where a rectangular slot is inserted into a Cellular Model



(a) before the operation (b) after the operation

Figure 4.6 – Re-evaluation of the Cellular Model for a model modification

consisting of a single block. Before the cellular union, the owner lists of both cells are as shown in Figure 4.5.a (for the sake of legibility, only some face owner lists of both shapes are depicted). After the operation, the block cell is decomposed into two cells, of which one is shared with the slot, as depicted in Figure 4.5.b. The owner lists of the cell faces in Figure 4.5.a are also propagated, when these faces are split, as shown in Figure 4.5.b.

Re-evaluation of the Cellular Model after each modeling operation makes extensive use of the ability to process the cellular topology. A detailed description of Cellular Model processing algorithms can be found in (Bidarra et al. 1998b). According to the particular feature operation, these can be summarized as follows:

Adding a new feature instance to the model The shape extent of the new feature is added to the current Cellular Model. For this, the nonregular cellular union operation is used, which computes the cellular decomposition described above, and propagates the owner list attributes among the relevant cells and cell faces in the Cellular Model.

Removing a feature instance from the model This is carried out in three steps: (i) all references to that feature are removed from the owner lists of Cellular Model entities; (ii) cells with an empty owner list are removed from the Cellular Model; and (iii) adjacent cells and cell faces with the same owner list are merged.

1	block	1	block	1	block
2	step	2	through slot	2	step
3	through slot	3	blind slot 1	3	through slot
4	blind slot 1	4	blind slot 2	4	rib
5	blind slot 2	5	rib	5	blind slot 1
6	rib	6	step	6	blind slot 2

(a) before the operation (b) after the operation

Figure 4.7 – Feature precedence examples for the models of Figure 4.6

Editing a feature instance in the model In this case, *only* the edited feature, and all its dependent features that are also modified by the operation, need to be taken into account. They are removed from the Cellular Model and then re-added with their new parameters, using the *add* and *remove* operations just described.

A simple example of a feature modification operation is given in Figure 4.6, where the top attach of the through slot is modified, from the top of the block to the bottom of the step. The rib is also displaced, due to its attachment to the slot, whereas all other features maintain all their parameters and their position. The scope of modified features is easily obtained from the Constraint Manager (see Section 2.3), which keeps track of which features it modifies during the geometric and algebraic constraint solving process, as mentioned in the previous section. In this example, thus, *only* the slot and its dependent rib need to be updated in the Cellular Model. This is carried out by removing the (cells of the) slot and the rib from their original position (i.e. cells 5, 6, 7 and 8 in Figure 4.4), and adding their shape extents (with a cellular union operation) in the new position.

This example also illustrates that re-evaluation of the Cellular Model is independent of the chronological order of feature creation: the process is the same, regardless of whether the slot has been the first feature to be attached to the block or not (see Figure 4.7.a). In contrast with this, in the history-based approach, after the slot displacement operation, the whole model history (at least since the slot creation) is re-executed, including features whose imprint remains unaltered, e.g. blind slot 1 and

blind slot 2 (see model history at the left-hand of Figure 4.7.a). Even worse, a history-based modeling system would not be able to perform this operation if the model history were that at the right-hand of Figure 4.7.a, because the step is there more recent than the slot (see discussion of this drawback on page 11).

Computational cost of Cellular Model re-evaluation

An important issue is the efficiency of operations on the Cellular Model, because boundary evaluation is still a bottleneck in many modeling systems. The structure of the Cellular Model is certainly more complex than that of a manifold boundary representation, normally used in history-based feature modeling systems. In addition, attribute storing and propagation mechanisms demand some additional processing not required by set operations on a conventional manifold boundary representation. However, this is far outweighed by the performance improvement of incremental re-evaluation of the Cellular Model.

In history-based feature modeling, evolution of the model is, by definition, dependent on the re-execution of sequences of modeling operations from the model history. As discussed in Chapter 2, it is impossible to always avoid including in those sequences operations on unmodified features, although their re-execution is superfluous. The computational cost of a modeling operation, such as the modification or removal of a feature, is therefore proportional to the total number of features in the model if no intermediate evaluated models are stored, or to the number of features created after the modified or deleted feature, if intermediate models or deltas are stored.

Building the whole Cellular Model from scratch has also a computational cost that is proportional to the number of features in the model. Fortunately, this is only required when the Cellular Model needs to be built in one step, e.g. when starting a modeling session with a previously created model file.

Once this has been done, the computational cost of re-evaluating the Cellular Model after a modeling operation is kept limited, i.e. it is independent of the total number of features present in the model, because, as described in the previous subsection, only the imprint of the shape extents whose geometry has been affected by a modeling operation is updated. This scope is easily obtained from the geometric and algebraic constraint solving process, which keeps track of which features it actually modifies during the operation (see Section 4.4).

In conclusion, the computational cost of Cellular Model re-evaluation is dependent on the number of features whose geometry is affected by the operation. Usually, this number is very limited, so computational cost is minimized.

4.6 History-independent interpretation of the Cellular Model

Interpretation of the Cellular Model consists of determining whether the point set represented by each cell does or does not belong to (or represent "material" of) the product, i.e. the nature of that cell. This requires deciding which of the features in its owner list "prevails", either as additive or as subtractive (Bidarra and Bronsvoort 1999c). It is only at this point that the precedence among features needs to be taken into account.

Determination of cell natures

If, based on some precedence criteria, a *global ordering* can be defined on the set F of all features in the model (say assigning to them unique, increasing *precedence numbers*), then every cell owner list (a subset of F) can be sorted according to these precedence numbers. The nature of a cell becomes, then, the nature of the last feature in its owner list (i.e. the feature with the highest precedence number). It is obvious that such a global ordering is always possible, as the set of features in the model is discrete and finite, and thus numerable.

The main question is then *which* precedence criteria should be used in this sorting process. Before going into this, some remarks should be made on *what* is involved in the notion of cell nature, and *what* is thus relevant in its determination.

The example in Figure 4.7 shows that, at least in some cases, the same set of features can be sorted in different sequences, yet yielding the same model interpretation. Indeed, what makes such sequences have the same interpretation is the fact that they cause *the same nature* to be assigned to *the same cells* in the Cellular Model.

Considering that the nature of a cell, whose owner list has n elements, is exclusively determined by the n^{th} element (the last feature) in the owner list, we can derive the following properties:

1. The nature of a cell is independent of (the precedence numbers of) features that do not occur in its owner list.

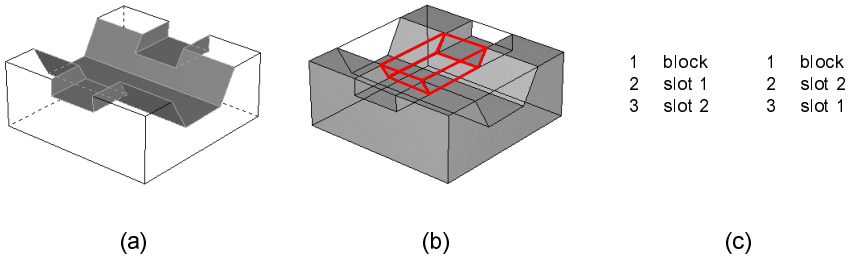


Figure 4.8 – Precedence permutation among independent features with the same nature

For example, referring to the model in Figure 4.6, the nature of the (cell of the) blind slot 1 is independent of whether the precedence numbers of, say, the step, the blind slot 2 and the through slot are higher or lower than its own precedence number.

2. The nature of a cell is preserved under permutations of the n elements of its owner list, provided that the nature of the n^{th} element is kept the same. In particular, the cell nature remains invariant under permutations of the first $n–1$ elements of its owner list.

   This is illustrated by the example model with two crossed slots, in Figure 4.8.a: the nature of the interaction cell shared by both slots (see Figure 4.8.b) is not affected by the relative precedence of these two subtractive features, and thus both precedence sequences in Figure 4.8.c yield the same interpretation.

From these two properties, we can conclude that, in general, different feature precedence sequences can result in the same nature for each cell. For the interpretation of the model, it is thus enough to have a procedure that is always able to generate *one* such sequence. We now discuss appropriate precedence criteria to achieve this goal.

Precedence criteria

The example in Figures 4.6 and 4.7 suggests that sorting the precedence sequence of features according to the *static* chronological feature creation

order, is not a good criterion for the interpretation of the Cellular Model. In fact, whichever the sequence of precedence numbers before the operation, changing the slot's attachment requires the step to precede the slot and the rib after the operation. Otherwise, the precedence number of the rib would be lower than that of the step, and the former would appear truncated by the latter. We can conclude from this example that the precedence sequence of features should be *dynamic*, i.e. subject to revision after each modeling operation.

This example, together with property 2 above, also shows that for the interpretation of the structure of the feature model at any moment, the chronological order in which its features were originally created is, in general, not determinative. Instead, *the actual dependencies* among them *at that stage* do provide the key for this precedence analysis.

For the model of Figure 4.6.a, for example, one can draw the following two precedence relations, based on an attachments' analysis: (i) the through slot feature precedes the rib feature (i.e., the latter is dependent on the former), and (ii) the base block feature precedes all other features (i.e., they are all dependent on it). Relative precedence among all other features is irrelevant when it comes to interpret this model; so, for example, both feature precedence sequences of Figure 4.7.a produce the same model interpretation of Figure 4.6.a. Figure 4.7.b, on the other hand, shows a possible sequence of precedence numbers for the modified model in Figure 4.6.b. Whatever the sequence of precedence numbers before the operation, it can be remarked that the step now precedes the through slot (i.e. has a lower precedence number), as required by the new attachment of the latter.

In short, the dynamic dependency relation of the Feature Dependency Graph permanently "reflects" the current structure of the feature model. Therefore, it makes up the first precedence criterion in our goal of generating a global precedence sequence:

> **Criterion I** Each edge in the Feature Dependency Graph represents a precedence relation between two features in the model: if feature f_2 depends on feature f_1, then f_1 *precedes* f_2.

By definition, the above criterion is able to define a precedence relation between dependent features only. On the other hand, for the modeling operation described in Figure 2.2, on page 10, a precedence problem was pointed out between two independent features, a blind hole and a protrusion: if the precedence numbers were kept as shown in Figure 4.9.a, i.e. following the sequence of the history in Figure 2.2.c, the top interaction

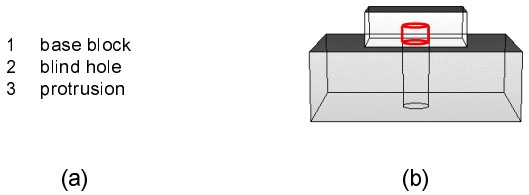


Figure 4.9 – The precedence sequence (a) for the model of Figure 2.2.e yields an incorrect nature for the cell highlighted in (b)

cell of the blind hole (highlighted in Figure 4.9.b) would be additive, i.e. have the nature of the protrusion. This nature is incorrect, because it is not in accordance with the semantics of the modeling operation performed: the nominal depth of the blind hole, which has been increased, does not match the actual depth it exhibits in the model.

What is characteristic of the situation described in Figures 2.2 and 4.9, is that the modeling operation in question causes an *overlap* between two *independent features of different natures*. To avoid incorrect interpretations of a model such as that shown in Figure 4.9, an explicit precedence relation should be established when, as a result of a modeling operation, two independent features with different natures come to overlap. The question arises then on *which* orientation should be assigned to this precedence relation, considering that none of the two features depends on the other. The above example suggests that, to preserve the semantics of a modeling operation, a feature f that is modified by the operation should "prevail" in the determination of the nature of its interaction cells. Stated differently, other overlapping independent features with different nature should precede f in the precedence sequence. After the operation in Figure 2.2, thus, the protrusion should precede the blind hole. Hence the following:

Criterion II To each *new* overlap between independent features f_1 and f_2 of different natures, caused by some modeling operation on f_2, corresponds a precedence relation f_1 precedes f_2.

With the two criteria above, based on the dependency relation and on possible overlap between independent features, a global sorting of all features in the model can be achieved. In the next subsection, we show how such precedence criteria are used to produce a correct interpretation

of the Cellular Model, which is unambiguously determined without invoking model history considerations.

Computation of feature precedence relations

The precedence relation, defined in the previous section, is an example of a so-called *partial ordering* relation, i.e. a relation which defines an ordering between *some* pairs of elements in a set S, but not among all of them. In general, partial ordering relations satisfy the following three properties for any distinct elements x, y and z of the set S:

1. Transitivity

   if ***x*** precedes ***y*** and ***y*** precedes ***z***,

   then ***x*** precedes ***z***

2. Asymmetry

   if ***x*** precedes ***y***,

   then ***y*** does not precede ***x***

3. Irreflexivity

   x does not precede ***x***

The dependency relation used in Criterion I is permanently maintained in the Feature Dependency Graph, and is therefore always explicitly available for use in the model interpretation process.

Criterion II states that an explicit precedence relation should also be established when a modeling operation causes an overlap between two independent features with different natures. To detect such occurrences and determine the orientation of the relation, the set of features involved in the modeling operation (i.e. those which are actually processed in the incremental re-evaluation of the Cellular Model, see Subsection "Cellular Model incremental evaluation" in the previous section) is analyzed according to the Algorithm 4.1. Basically, the algorithm checks whether any of these features, f_i, has acquired a new overlap with an independent feature f_j; if this is the case, and the features have different natures, then the relation "f_j precedes f_i" is recorded.

In detecting a new overlap, the algorithm uses the notion of *overlapping set* of a feature f, denoted *OS(f)*, i.e. the set of all features that overlap with feature f (see Section 6.1). Determination of the *OS(f)* is straightforward and requires no geometric computations: it is simply

```
InvolvedFeatures = {features involved in modeling operation}
for each fi in InvolvedFeatures
   NewOverlappings = OSafter(fi)\OSbefore(fi)
   for each fj in NewOverlappings
      if fi independent of fj and fi.nature ≠ fj.nature
      then record relation "fj precedes fi"
```

Algorithm 4.1 – Precedence detection algorithm for overlapping independent features

```
NewSequence = <>
OldSequence = <current precedence sequence>
while OldSequence is not empty do
   find in OldSequence the next feature f such that
           all precedents of f are already in NewSequence
   move f from OldSequence to NewSequence
   assign new dependency number to f
```

Algorithm 4.2 – Topological sorting algorithm for assigning feature precedence numbers

computed as the union of the owner lists of all cells of feature *f*. The overlapping set of each feature f_i involved in the operation is computed and stored before the Cellular Model is re-evaluated, and compared with the $OS(f_i)$ determined after the re-evaluation, in order to detect new overlaps.

Once the precedence relations have been established, using the two criteria described, the global sorting of features can be easily performed by a classical topological sorting algorithm, whose goal is precisely to generate a linear ordering of a partially ordered set of elements (Wirth 1976). The algorithm, see Algorithm 4.2, builds a new sorted sequence by iteratively selecting (and removing) from the old sorted sequence a feature whose precedents are all already sorted.

The number of tests in this selection is minimized if candidate features are sought in the order of the old sorted sequence, because most modeling operations have a fairly local effect, affecting only the precedence of a few other features, if any at all. For example, adding a new feature to the model typically maintains the whole precedence sequence, the new feature being just attached at its end.

Eventually, the features in the resulting sorted sequence have new precedence numbers assigned, and the nature of all cells becomes thus automatically determined.

Summarizing, precedence numbers are revised after every modeling operation. For this, precedence relations are updated in the model, and a new sorting is performed among all its features. These get then new precedence numbers assigned, reflecting the new model structure, as has been illustrated for the modeling operation of Figures 4.6 and 4.7.

Application examples

In this subsection, a number of examples is presented to illustrate both the incremental evaluation and the interpretation of the Cellular Model. In each example, a modeling operation is performed that involves changes in one (or more) feature(s). The Cellular Model corresponding to the final situation is also shown, together with the graph of precedence relations used in its interpretation. In this graph, the feature nodes that are actually modified by the operation are highlighted (in black). Moreover, additional precedence relations between independent features, established by the precedence detection Algorithm 4.1, are drawn with a dotted line, to distinguish them from the other precedence relations, derived from the dependencies in the Feature Dependency Graph.

Example I

The initial model consists of a base block, two ribs and a through hole, attached between the two ribs, see Figure 4.10.a on the next page. A protrusion is then inserted between the ribs, so that it overlaps with the through hole, see Figure 4.10.b. Considering that the through hole and the protrusion are overlapping independent features, the precedence detection Algorithm 4.1 prescribes that the through hole should precede the protrusion, as indicated by the dotted edge in Figure 4.10.c. Thus the protrusion receives the highest precedence number in the sorting algorithm and, consequently, the nature of the interaction cell highlighted in Figure 4.10.d is additive, i.e. that of the protrusion.

In a way, this is comparable to what history-based modeling systems correctly assume when a *new* feature is *added* to the model: it becomes the last feature in the model history and, thus, it is the last to leave its shape imprint on the model boundary. This strategy is, in fact, a particular case of Criterion I (a *new* feature is always made dependent on

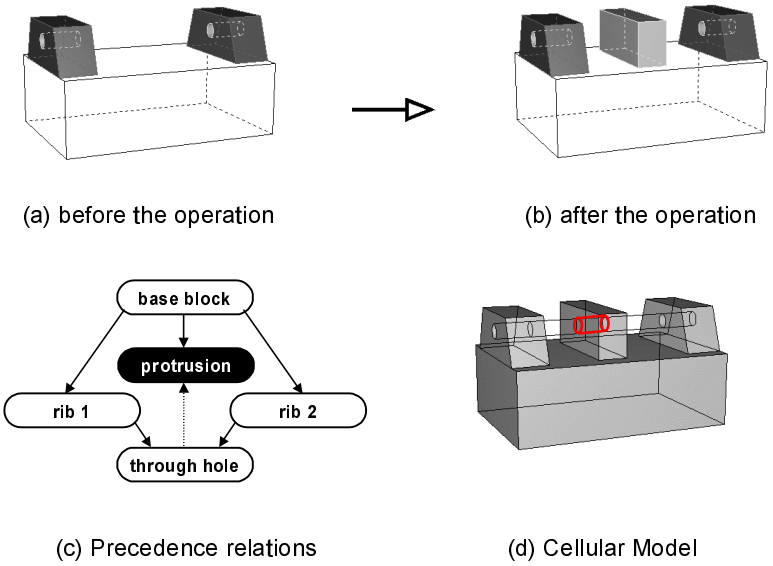


Figure 4.10 – Cellular Model interpretation after adding a new feature

existing features), and possibly also of Criterion II (the *new* feature overlaps with *existing* independent features, as in the example of Figure 4.10).

However, history-based boundary re-evaluation often fails when some *existing* feature is *modified* in the model, as discussed in Chapter 2. The approach presented above, instead, remains applicable for all modeling operations, as will be illustrated with the next two examples.

Example II

In this example, the model has two crossing slots of different depths attached to a base block, and a rib on the bottom face of the deeper slot, through slot 1, see Figure 4.11.a. The depth of the split through slot 2 is then increased, so that it overlaps with the rib, see Figure 4.11.b. Again, as these two features are independent, their overlap leads to a precedence relation being established between them, see Figure 4.11.c. As a consequence, the rib receives a precedence number lower than the

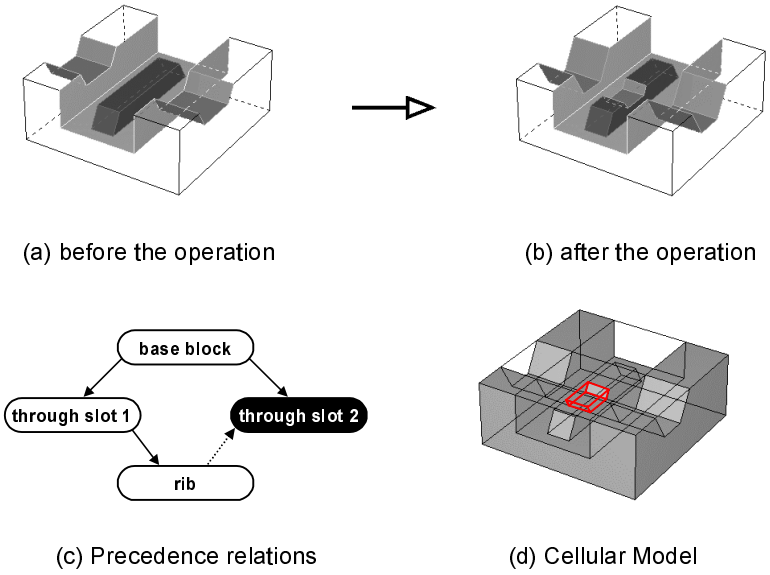


Figure 4.11 – Cellular Model interpretation after editing a subtractive feature

through slot 2, and their interaction cell is thus subtractive, as shown in Figure 4.11.d.

With history-based boundary re-evaluation, the resulting model of Figure 4.11.b, would not be achievable if the through slot 2 had been created before the rib.

Example III

The third and last example is based on the same model of example II, see Figure 4.12.a. However, the modeling operation now consists of decreasing the depth of the deeper slot, through slot 1, such that its dependent rib becomes in interaction with the other slot, through slot 2, see Figure 4.12.b. In this case, from the analysis of the precedence detection algorithm, the (indirectly) modified rib is preceded by the independent through slot 2, see Figure 4.12.c, resulting in an additive nature for their interaction cell, highlighted in Figure 4.12.d.

Again, the detection of the new overlap, and the precedence relation established, yields a model interpretation in which the nature of the

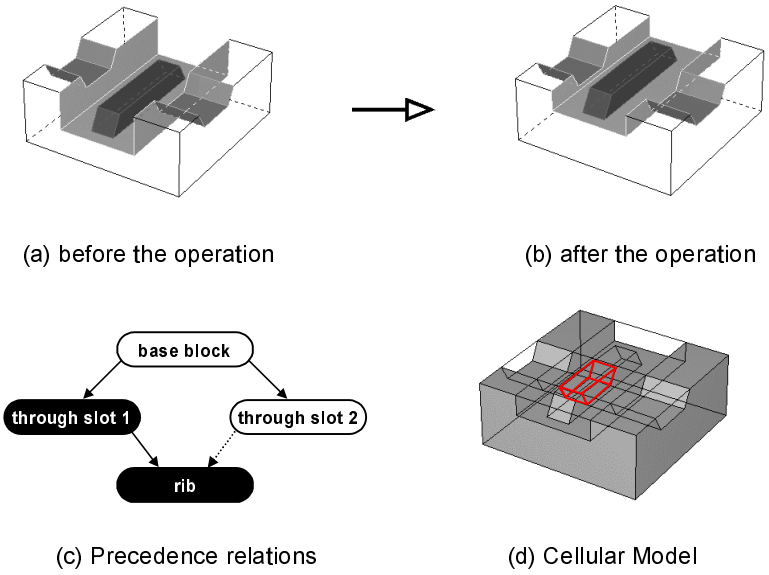


Figure 4.12 – Cellular Model interpretation after (indirectly) editing an additive feature

modified features prevails over that of the other overlapping features. To achieve the model of Figure 4.12.b using a history-based modeling system, the through slot 2 should be created before the rib (which is exactly the history sequence that would make the resulting model of example II unfeasible).

4.7 Conclusions

The semantic feature model described in this chapter provides a structured model representation that is very suitable for feature modeling.

First, it maintains a high-level representation of the product –the Feature Dependency Graph–, with only those entities that are relevant for its manipulation: features and model constraints. This facilitates user interaction, providing a natural dialog in terms of features, and hiding from the user many unnecessary details of the geometric model. Fur-

thermore, performing all constraint solving tasks at this level avoids the shortcomings of entity naming schemes pointed out in Chapter 2.

Second, it maintains the Cellular Model as evaluated geometric representation of the product. From the properties of the Cellular Model described in this chapter, we can conclude that it is significantly more suitable than a conventional boundary model for the geometric representation of feature models. In particular, (i) it is able to represent subtractive and overlapping features in a consistent way, (ii) its re-evaluation after each modeling operation has a lower computational cost, compared to that of boundary representations maintained in history-based modeling systems, and (iii) its evaluation and interpretation is independent of the chronological order of feature creation in the model. The latter solves several problems inherent to history-based modeling.

These two levels are integrated in the semantic feature model, which disposes of mechanisms for automatically maintaining the consistency between them. In addition, it supports a variety of queries at both levels, making it possible to perform effective model validity maintenance throughout the modeling process, as will be described in Chapter 7.

5

Feature interactions

"(...) interactions among features pose many complex problems to the developers of integrated CAD/CAM systems."

(Regli and Pratt 1996)

Many feature validity violations are caused by feature interactions, which arise from modeling operations such as the creation of a new feature or the modification of an existing feature. It is important to get insight in feature interaction phenomena, so that all relevant interaction situations in a particular context can be detected, classified and handled in an appropriate way.

This chapter defines what feature interactions are within the scope of the research described in this thesis. First, a survey is given of previous research contributions dealing with feature interactions (Section 5.1). Next, a definition of feature interaction is developed (Section 5.2). Finally, a classification of feature interactions is presented and illustrated, based on design and technological criteria (Section 5.3).

5.1 Previous research

The first explicit definition of an interaction relationship between two features was given by Pratt (1988). This was based on the notions of *effec-*

tive volume of interaction (EVI) and *actual volume of interaction (AVI)*, and a feature graph was proposed to represent such relations on a form feature model. Although limited in scope, partly due to the lack of a convenient geometric model, this scheme was indeed able to capture simple interactions between features.

Rossignac (1990) alerted for the difficulties and inconsistencies, mainly due to the occurrence of feature interactions, that might arise from a naive interpretation of usual editing commands on feature models. His analysis provides a good insight into some high-level, design-oriented validity issues of feature models, although some of the problems pointed out only occur with a CSG-based representation of features. He also proposes the use of SGCs –Selective Geometric Complexes (Rossignac and O'Connor 1990)– as a geometric representation scheme that would facilitate the detection of features in interaction.

An informal definition for feature interactions was presented by Shah (1991), relating them to "intersections between entities of two or more features such that either the shape or semantics of a feature are altered from the generic definition". A number of typical feature interactions were also illustrated. The conclusion from his analysis is that unconditionally preventing such occurrences would excessively constrain the designer activity, and it is suggested that the modeling system should detect and react to them according to pre-defined procedures.

da Silva et al. (1991) proposed a strategy for explicit representation of relations between cavity features, in order to perform manufacturability reasoning. They distinguish interacting and inter-feature relationships. The latter are equivalent to geometric constraints now commonly available in modeling systems; the former are actually no more than simple attach relations between feature constituents. A major limitation of their approach is that both the relationships and the feature constituents themselves have to be extracted from the geometric model, which is virtually impossible for intersecting features in general.

Karinthi and Nau (1992) addressed some interaction problems from a formal perspective, defining an algebra of features. Its purpose is to generate all possible alternative configurations of a feature model in terms of manufacturing features, in order to provide better input for a process planning system. They assume that most of the alternatives are due to feature interactions. However, the restricted algebra they implemented either does not yield all possible interpretations, or generates interpretations that are unsuitable for manufacturing purposes (Gupta and Nau 1995).

Bidarra and Teixeira (1993) proposed a functional classification of feature interactions, suggesting that these could be detected by analyzing the topology of the boundary of volumetric features. For this, a conceptual partition of the feature boundary was proposed, distinguishing in it functionally different subsets –the so-called *feature elements* (e.g. the *top*, *bottom* and *side* faces of a blind hole feature). The actual representation of feature elements in terms of topologic entities in the geometric model could then be constrained, according to whether those elements should contribute to the boundary of the part model or not, by means of so-called *semantic constraints*. Bidarra and Teixeira (1994) suggest that such constraints can be integrated in the validity criteria of each feature class, and maintained in the model throughout a modeling session. (See Section 3.2, on the use of these constraints in the present approach.)

A methodology for analytical detection of geometric interactions between features was proposed by Talwar and Manoochehri (1994). The algorithms presented take all the necessary input from a boundary representation and require an intensive, low-level query and traversal of this model. These algorithms provide a primary classification of feature interactions (contained, touching, intersecting, etc.), and are mainly intended for use in downstream, rule-based manufacturability analysis systems.

Suh and Ahluwalia (1995) approached feature interaction problems from the perspective of incremental feature generation. They propose to analyze each new shape, created in the model with a set operation, identifying the scope of its interaction and the actual feature produced. In addition, changes caused by this operation on previously existing features are detected, and these are redefined accordingly. This method is suitable for detection of a limited set of interaction types only, mainly because identified features are regarded as open face sets in the evaluated geometric model. Furthermore, cases having more than two interacting features are not dealt with.

Regli and Pratt (1996) distinguished 3 types of interactions independent of any application domain: (i) *interference interaction*, among features whose volumes intersect, (ii) *adjacency interactions*, mainly related to assembly connections, and (iii) *remote interactions*, mostly derived from geometric relationships between non-overlapping features. Although their analysis is somehow biased towards feature recognition issues, some interesting open problems are pointed out, e.g. the interaction between features of different views of a product.

It can be concluded that the perception of feature interactions achieved so far is either too abstract or too detailed: the former is hard to implement and put into effect, the latter fails to capture important aspects of feature semantics. Mostly, difficulties found with feature interaction phenomena arise from studying them mainly at the geometric representation level of the feature model. Therefore, it seems clear that effective feature interaction management requires high-level, functional information in the product model to be considered as well (Bidarra and Bronsvoort 1996).

5.2 Definition of feature interactions

From the literature survey in the previous section, it becomes clear that no general consensus exists as to what *feature interactions* are. This may be partly assigned to the lack of a common understanding of the concept *feature* itself. In particular, proposals to regard as features various non-shape aspects of a product (material, precision, function, etc.) lead almost inevitably to notions of interaction with only analogous, and often vague, contents.

However, recent literature also shows that even dealing with only *form features*, different phenomena are often regarded under the notion of feature interaction. For example, the adjacency and remote interactions pointed out by Regli and Pratt (1996) are, in our view, better described by means of geometric and algebraic constraints among feature elements and parameters, respectively. This would have the additional advantage that such relations could be maintained and analyzed by known constraint solving methods, see (Dohmen 1997) for an application example. Similarly, many life-cycle interactions identified by Regli and Pratt, can be advantageously captured by means of the inter-view link constraints also described by Dohmen (1997).

Within the scope of this research, the term *feature* always denotes *form feature*, defined in Chapter 1 as a "representation of shape aspects of a product that are mappable to a generic shape and are functionally significant for some product life-cycle phase". In this context, we reserve the term *feature interaction* for those cases where a spatial overlap occurs between feature shapes. In other words, *two features interact whenever they share some region in space*, designated *the interaction extent*. Interaction extents can be three-dimensional –*volumetric interaction*– or have a lower dimension –*boundary interaction*.

From the above definition of feature interaction, it follows that virtually every feature in a model interacts with some other feature. This is not surprising, because the combination of overlapping features is often the best method of producing a desired complex shape in the model; sometimes it is actually the only way, in particular when no suitable compound feature classes are directly available from the feature library.

On the other hand, it is also clear that the occurrence of interacting features can cause the shape imprint of features to undergo significant modifications. Such changes may have rather diverse characteristics and relevance. For example, within feature recognition approaches, as pointed out by Regli and Pratt (1996), the distortion of certain feature hints or traces (e.g. topologic entities) in the geometric model is a serious problem, because many recognition algorithms are strongly dependent on them. In design by features, and particularly in semantic feature modeling, the crucial aspect is the eventual *change in feature semantics* resulting from such feature interactions.

Feature semantics, as described in Chapter 3, is specified by means of a variety of constraints. Among them, semantic and interaction constraints are meant to specify to which extent interaction with other features is allowed. Such constraints can either be specified in a feature class, and thus hold for all its instances, or be associated to one particular feature instance as a model constraint, during a modeling session.

5.3 Types of feature interactions

In this section, a number of feature interaction types is identified. Their relevance is rather dependent on the application domain: some have a clear technological connotation, whereas others are more biased towards modeling or functional aspects. Feature interactions have a very wide range of effects on a feature model. Even using the restricted definition of feature interaction presented in the previous section, we argue that a complete taxonomy for feature interactions is unattainable. Two main reasons can be given for this: (i) overlap between features with arbitrarily complex shapes cannot be cast into an exhaustive classification; and, above all, (ii) the strong domain-oriented relevance of feature interactions mentioned above would make such a general attempt unacceptably restrictive. In this sense, this section is no more than an enumeration of *typical* feature interaction classes.

This classification does not mean that all interaction types are equally relevant and meant to be disallowed for all feature classes in all

feature libraries. As stressed in the previous section, disallowing an interaction type for a particular feature class (or instance) is done by specifying for it the appropriate semantic and/or interaction constraints. It is, thus, upon the user of the Feature Library Manager (or of the Feature Modeler) to establish which conditions best suit the requirements of his/her particular domain and perspective of a product. For example, a blind hole class might have different interaction requirements in a design feature library and in a manufacturing feature library.

In practice, the interaction classes described here are not even mutually exclusive, as will soon become clear. This means that a feature instance may undergo several of them simultaneously. This point will be discussed in the next chapter, on account of the detection of multiple interactions (Section 6.3).

For some interaction classes described here, a partition of the feature boundary (FB) of a feature will be considered, relating its points to the model boundary, according to:

$$FB = FB^{+} \cup FB^{-}$$

where

- FB^{+} represents the set of points of the feature boundary that are *on* the model boundary, and
- FB^{-} represents the set of points of the feature boundary that are *inside* or *outside* the model boundary (e.g. points of so-called closure faces of subtractive features).

The goal of the proposed classification is threefold: (i) to provide the user with the capability to allow/disallow selected interaction classes, by means of interaction constraints; (ii) to make possible the development of appropriate detection mechanisms (see Chapter 6); and (iii) to allow specific potential reactions of the modeler to each interaction detected (see Chapter 7).

Splitting interaction

Definition: interaction that splits the FB^{+} of a feature into two (or more) disconnected subsets.

Splitting interactions are always caused by an overlap with one (or more) subtractive feature(s). This results in the suppression from the model

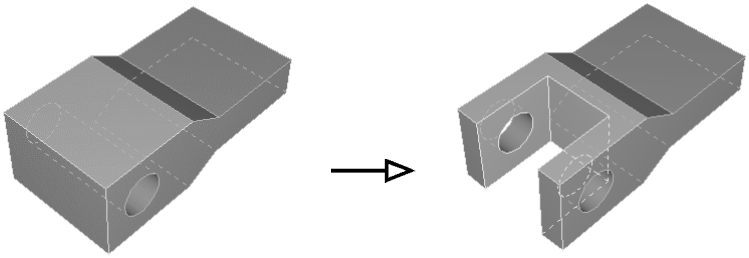


Figure 5.1 – Splitting interaction: insertion of the slot splits the through hole boundary into disconnected components

boundary of a subset of the FB^+ of a feature, in such a way that this becomes split into disconnected components, see Figure 5.1.

This kind of interaction may have both functional and technological consequences. On the one hand, the knowledge that a particular feature, e.g. a through hole, became split in the model, may give useful hints on feature precedence in a process planning analysis of the model. On the other hand, the functional purpose of some feature classes may also be affected by a splitting interaction, e.g. the function of rib or stiffener additive features, or the sliding functionality of a slot from an assembly point of view.

Disconnection interaction

Definition: interaction that causes the volume of an additive feature (or part of it) to become disconnected from the model.

Disconnection interactions may occur as a result of the manipulation of subtractive features, e.g. when their dimensions are made too large, causing the part model to become topologically disconnected, as shown in Figure 5.2.

Disconnection interactions are important to deal with because for some applications of the model (e.g. manufacturability analysis) a part should mostly consist of one connected component only (otherwise, assembly information would most likely be required as well).

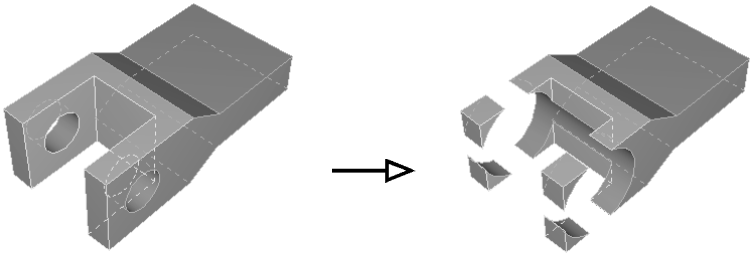


Figure 5.2 – Disconnection interaction: enlargement of the through hole diameter disconnects part of the block from the remaining of the model

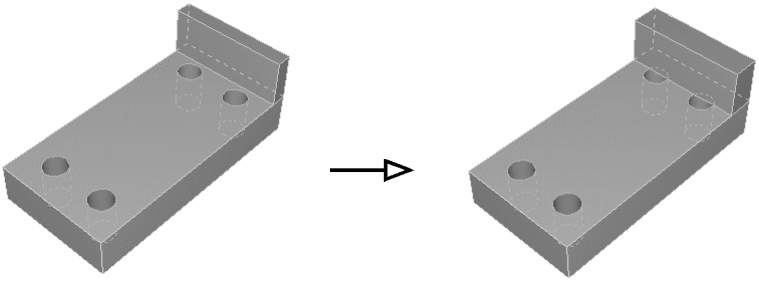
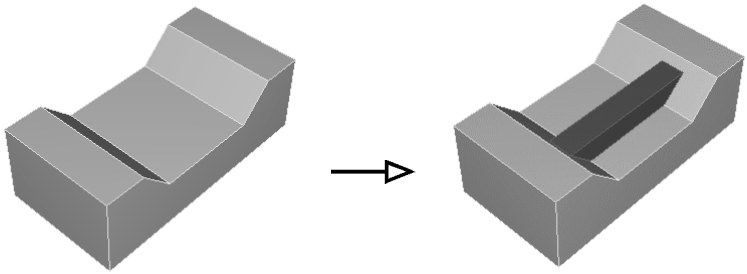


Figure 5.3 – Boundary clearance interaction: enlargement of the protrusion width obstructs entrance faces of the through holes

Boundary clearance interaction

Definition: interaction that causes (partial) obstruction of the FB^- of a subtractive feature.

Boundary clearance interactions have a technological connotation: they usually affect the machining process possible for a subtractive feature in several ways, e.g. approach tool direction, machine accessibility and tool path clearance. An example is given in Figure 5.3.



(a) (b)

Figure 5.4 – Volume clearance interaction: insertion of a rib obstructs the subtractive volume of the slot

Volume clearance interaction

Definition: interaction that causes partial obstruction of the volume of a subtractive feature.

Some subtractive feature classes, e.g. slot or step classes, may pose strong manufacturability requirements in terms of restricting additive volumetric interference into their own subtractive volume. The intrusion of an additive feature into the volume of a subtractive feature often constrains the machining process of the latter. For example, while the slot in Figure 5.4.a can be machined with one (or more) translational milling operation(s), a more complex machining process is required to produce the slot-rib combination in Figure 5.4.b.

Closure interaction

Definition: interaction that causes some subtractive feature volume(s) to become a closed void inside the model.

Closure interactions are an extreme case of clearance interactions, in the sense that they cause a complete inaccessibility of subtractive features.

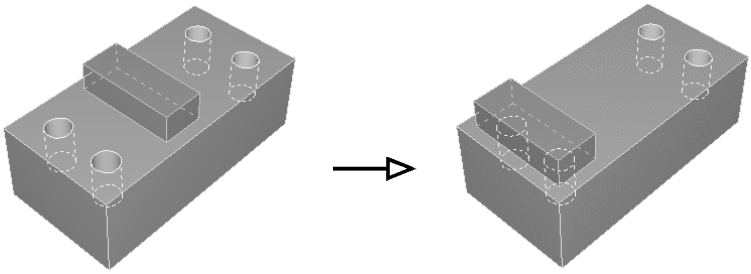


Figure 5.5 – Closure interaction: displacement of the protrusion causes the two blind holes to become closed voids inside the model

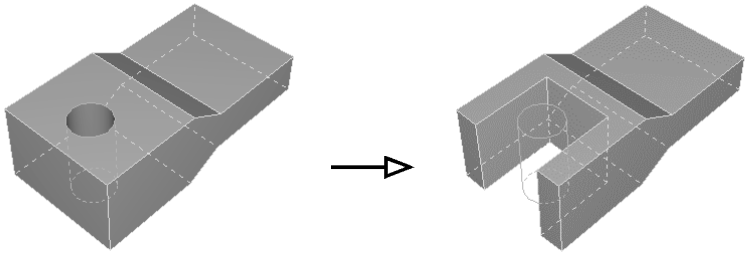


Figure 5.6 – Absorption interaction: insertion of a slot suppresses contribution of the through hole to the model shape

Therefore, their detection may be also important, from both the functional and the manufacturing viewpoint.

Each closed void produced by a closure interaction can be the volume of a single feature, as illustrated in Figure 5.5, or the combination of several overlapping volumes.

Absorption interaction

Definition: volumetric interaction that causes a feature to cease completely its contribution to the model shape.

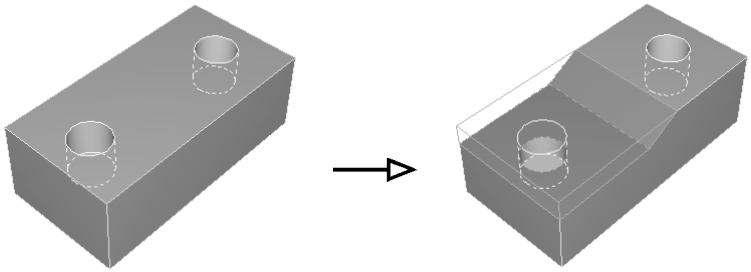


Figure 5.7 – Geometric interaction: insertion of a step changes the actual depth of the blind hole

Absorption interactions are almost always undesirable, because they turn useless a feature that was previously created. An exception to this, however, is the creation of temporary features, e.g. for fixture or gripping purposes, in a process planning model: after they have played their role, such features may be absorbed by others in the model.

Absorption interactions occur whenever the FB^+ of a feature becomes empty and its volume is completely enclosed within the volume of one (or more) feature(s), as depicted in Figure 5.6.

Geometric interaction

Definition: volumetric interaction that causes a mismatch between a nominal parameter value and the actual feature geometry.

Geometric interactions may affect the functionality intended for a particular feature, in the sense that some of its shape parameters exhibit a value different from that nominally specified for it. This mismatch between *actual* and *nominal* parameter values may be relevant in process planning (e.g., availability of required tools or precedence of machining features). An example of this interaction is a decrease in blind hole depth due to a step insertion, as depicted in Figure 5.7.

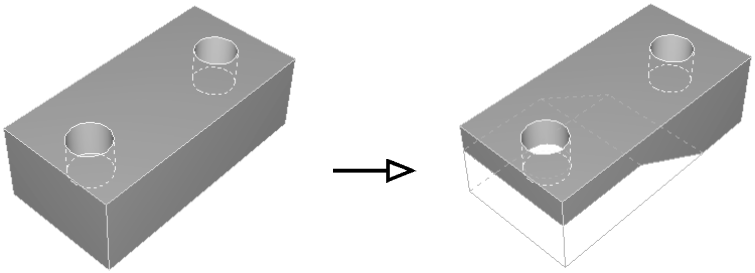


Figure 5.8 – Transmutation interaction: insertion of a step turns the blind hole into a through hole

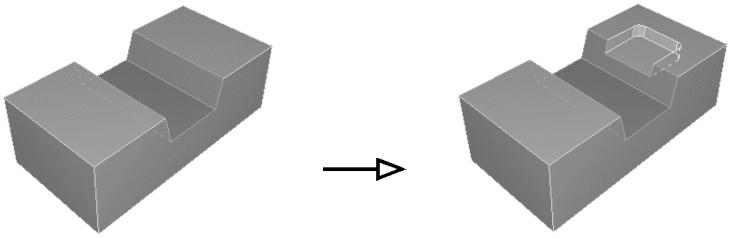


Figure 5.9 – Topologic interaction: insertion of a pocket suppresses part of a slot side face

Transmutation interaction

Definition: interaction that causes a feature instance to exhibit the shape imprint characteristic of another feature class.

The detection and control of transmutation interactions may be of particular importance from a process planning perspective of a product, considering that the transmutated feature may allow (or require) a machining process different from that of its original feature class. Moreover, their detection may avoid inadvertent overruling of previous designer intent. An example of this interaction is given in Figure 5.8, where a blind hole is turned into a through hole due to a step insertion.

Topologic interaction

Definition: interaction that causes the violation of a semantic constraint in a given feature.

Semantic constraints, introduced in Chapter 3, specify topologic requirements on elements of a feature shape. When, due to an interaction, one such constraint does not hold anymore, a topologic interaction takes place. As an example, the two side faces of a through slot may be required to lie completely on the model boundary. This means that no subset of these feature elements may be suppressed due to an interaction with another subtractive feature, in the way illustrated by the example in Figure 5.9.

5.4 Conclusions

The nature of feature interactions arising from the incremental manipulation of feature models has been addressed in this chapter. A definition of feature interactions was proposed, as a first necessary step in the development of any interaction management mechanism. This definition covers all interaction situations where two features overlap, and is directly mappable into detection algorithms, as will be shown in the next chapter. A classification of feature interactions was presented and illustrated, taking into account criteria from various life-cycle phases of a product.

6

Detection of feature interactions

"If there is no control over interactions, one could create nonsense features (...). If interactions are disallowed completely, it can be very inconvenient for users, because there will be a need to define many new generic features or users will not be able to create the geometry they want. (...) Sometimes it may be all right to allow interactions. (...) In all cases, it is necessary to detect the existence of interaction conditions, and determine what actions to take."

(Shah 1991)

After having presented the basic notions on feature interactions (Chapter 5), we can now turn to the more technical aspects of their detection in the semantic feature model. As anticipated in Section 4.4, the interaction detection mechanism is invoked at the end of each modeling operation, once the Cellular Model has been updated. In this chapter, both the global interaction detection mechanism (Section 6.1) and the detection algorithms for each interaction type (Section 6.2) are presented. Finally, possibilities for further extensions are discussed (Section 6.3).

6.1 The interaction detection mechanism

For each of the main feature operations –insertion, modification and removal of a feature–, the global interaction detection mechanism may be subdivided into three main phases (Bidarra et al. 1997):

1. determination of the interaction scope of the modeling operation;
2. detection of specific feature interactions arising from the operation; and
3. individual analysis of each interaction detected, in order to determine and report its scope and causes.

The *feature interaction scope* (FIS) of a feature operation is the set of all feature instances in the model that may potentially be affected by the operation.

For the determination of the FIS, two important notions with regard to a feature *f* are:

- the set of features that overlap with *f*, either volumetrically or with their boundaries; these features make up the *overlapping set* of *f*, denoted `OS(f)`, and they are identified by querying the Feature Geometry Manager, which keeps track of all feature shapes and their intersections in the Cellular Model (see Section 4.5);
- the set of features that depend on *f*; these features make up the *dependency set* of *f*, denoted `DS(f)`, and they are identified by querying the Constraint Manager, which recursively traces in the Feature Dependency Graph the dependency relations on *f* (see Section 4.1).

Depending on the modeling operation, the FIS will consist of different combinations of overlapping and dependency sets, as follows:

Adding a new feature instance to the model By definition, after adding feature *f*, there are no dependencies of other features on *f* yet, i.e. `DS(f)`=∅. The FIS of the operation is thus limited to

```
FIS ← {f} ∪ OS(f)
```

Editing a feature instance in the model In this case, the FIS has to be determined in two steps. Before the operation, it is initialized as

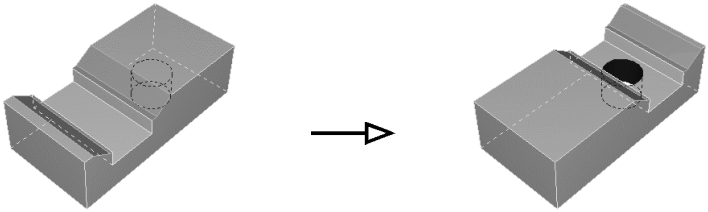


Figure 6.1 – Indirect interaction caused by a dependent feature

$$FIS \leftarrow \{f\} \cup DS(f) \cup OS(f) \cup \bigcup_{f_i \in DS(f)} OS(f_i)$$

in order to include those features whose overlap with feature *f* (or with its dependent features) will possibly cease after the operation.

Later on, i.e. after the Cellular Model has been re-evaluated, the FIS is updated, so that all features that only then overlap with feature *f* or with its dependent features are also taken into account

$$FIS \leftarrow FIS \cup OS(f) \cup \bigcup_{f_i \in DS(f)} OS(f_i)$$

With this scheme, interactions caused or suffered indirectly by any dependent feature are also detected. An example of this is given in Figure 6.1: displacement of the upper V-shaped slot implies the displacement of the attached rectangular slot, which in turn causes the transmutation of the blind hole.

Removing a feature instance from the model As pointed out in Section 4.4, this operation requires that the feature to be removed has no dependent features, i.e. $DS(f)=\varnothing$. The FIS is thus determined, before executing the removal of feature *f*, as

$$FIS \leftarrow OS(f)$$

The determination of the FIS has as purpose to avoid checking for feature interactions in vain later: features that are known in advance to be

left unaffected by the operation are not analyzed in the interaction detection procedure. This strategy pays because:

1. Mostly, feature operations have a localized scope, affecting only a small subset of all features in the model. This is particularly apparent in large models.
2. The information required to determine the FIS is explicitly stored in the feature model, either in the Feature Dependency Graph or in the owner lists of the Cellular Model, and its retrieval has, thus, a low computational cost. All that is needed is to query the Constraint Manager and the Feature Geometry Manager, respectively (see Section 2.3).
3. Many feature classes specify several interaction constraints for its instances. Checking all of them always, i.e. even when those instances would fall outside the FIS of an operation, has a higher computational cost than that of FIS determination.

In other words, for moderately complex, realistic feature models, situations for which the above strategy is not optimal occur very seldom, namely only: (i) when the FIS determined would include (almost) all features, because the scope pruning achieved would then be minimal, yet time consuming; and (ii) when most features in the model would have few (or no) interaction constraints, because no real detection computations would then be pruned out with the FIS.

Feature interactions taking place on any feature of FIS are detected by checking their interaction and semantic constraints. At this stage, the other Managers are queried, in order to obtain the specific data required by each detection algorithm described in the next section.

Each constraint violation is recorded by the Interaction Manager (see Section 2.3). Eventually, the set of constraint violations is analyzed, in order to identify their causes and report these to the user (see Chapter 7).

6.2 Interaction detection algorithms

In this section, detection algorithms are presented for the interaction classes described in the previous chapter. For simplicity, the algorithms are presented as logical predicates, although in fact each of them collects and returns additional information, in order to provide the user with a detailed explanation on the scope and causes of the interaction.

```
boundary ← s.boundary(ADDITIVE)
cf1 ← boundary.first
for each cell face cf2 in boundary
   if not boundary.accessible(cf1,cf2)
      return TRUE
return FALSE
```

Algorithm 6.1 – Splitting interaction detection algorithm

Each of these algorithms is aimed at checking the respective interaction constraint. Therefore, they operate on a feature shape, denoted by *s*. Only the detection algorithm for disconnection interactions operates on the whole model, provided that such interactions may take place without actually splitting any single feature shape, but rather disconnecting it from the remaining model volume.

The algorithms shown make use of methods provided by the Constraint Manager and the Feature Geometry Manager, in order to query their data. Most of these methods are described in detail in (Bidarra et al. 1998b); for completeness, a summary of them is presented in Table 1 on the next page.

Splitting interaction

Splitting interactions are described in terms of the nature of feature boundaries. They occur to a feature shape whenever the cellular decomposition of its boundary is such that the subset of its cell faces with additive nature is not connected.

To assess the connectivity of a set of entities, see Algorithm 6.1, the method checks whether from one of them, say `e`, all other entities in the set are *accessible* via the adjacency topologic relation of the Cellular Model. The predicate `accessible(e1,e2)`, defined on a set `s` of Cellular Model entities (cells or cell faces), returns `TRUE` if and only if for the two specified entities of the set, e_1 and e_2, the following holds:

1. $e_1 = e_2$; or
2. e_1`.adjacent(`e_2`)`; or
3. $\exists_{e_3 \in s}$: e_1`.adjacent(`e_3`)` $\wedge$ `s.accessible(`e_3`,`e_2`)`

Table 1 – Summary of methods used in the detection algorithms

CELLULAR MODEL, cm	
cm.cells(nature)	returns the list of cells with specified nature (`ADDITIVE` or `SUBTRACTIVE`) in the cellular model
FEATURE SHAPE, s	
s.nature	returns the nature specified for shape s
s.elements	returns the list of shape elements of shape s
s.cells	returns the list of all cells that lie in the shape extent of s
s.boundary(nature)	returns the list of cell faces with specified nature that lie in the extent of shape elements of s
s.overlappingSet(nature)	returns the list of shapes of specified nature that overlap with shape s (either volumetrically or with their boundaries - cell faces and edges)
s.constraints(type)	returns the list of constraints of specified type related to shape s
SHAPE FACE, f	
f.shape	returns the shape to which the shape face f belongs
f.cellFaces	returns the list of cell faces that lie in the extent of shape face f
CELL, c	
c.ownerlist	returns the list of shapes that own cell c
c.boundary	returns the list of cell faces that bound the volume of cell c
CELL FACE, cf	
cf.cell	returns the cell bounded by cell face cf
cf.partner	returns the partner cell face of cf that bounds an adjacent cell (if this exists)
cf.ownerlist	returns the list of shape elements that own cell face cf
cf.nature	returns `ADDITIVE` if the cell face cf lies on the model boundary, and `SUBTRACTIVE` otherwise
OWNER LIST, l	
l.last	returns the last element of the owner list l
l.after($element_1$, $element_2$)	returns `TRUE` if $element_1$ occurs after $element_2$ in the owner list l

The names of the split feature elements, as well as the feature(s) causing the interaction, are collected by the algorithm.

```
cells ← cm.cells(ADDITIVE)
c1 ← cells.first
for each cell c2 in cells
   if not cells.accessible(c1,c2)
      return TRUE
return FALSE
```

Algorithm 6.2 – Disconnection interaction detection algorithm

```
semanticConstraints ← s.constraints(SEMANTIC)
for each sc in semanticConstraints
   if sc.type = nob(COMPLETELY) and not sc.check
      return TRUE
return FALSE
```

Algorithm 6.3 – Boundary clearance interaction detection algorithm

Disconnection interaction

Disconnection interactions are analogous to splitting interactions, but they are better described in terms of additive shape volumes instead of additive boundary elements. They occur to an additive feature (or to the model as a whole) whenever its cellular decomposition is such that the subset of its additive cells is not connected, see Algorithm 6.2. The feature(s) causing the disconnection, and the feature possibly being disconnected, are also identified by the algorithm.

Boundary clearance interaction

A boundary clearance interaction occurs to a subtractive feature whenever a semantic constraint of type `notOnBoundary(completely)` on one of its shape elements is not satisfied, see Algorithm 6.3. For this, the `check` predicate of semantic constraints is used. Moreover, the feature shapes causing the semantic constraint violation are identified and returned.

```
for each cell c in s.cells
   list ← c.ownerlist
   for each shape si in list
      if list.after(si,s) and si.nature = ADDITIVE
         return TRUE
return FALSE
```

Algorithm 6.4 – Volume clearance interaction detection algorithm

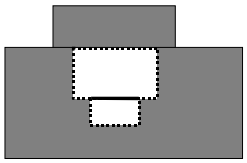
Volume clearance interaction

A volume clearance interaction occurs to a subtractive feature whenever a subset of its volume is later occupied by an additive feature. The detection of this interaction type, see Algorithm 6.4, relies on checking the owner list of all cells in the subtractive feature shape. Considering that each owner list is sorted according to the precedence criteria presented in Section 4.6, the detection algorithm succeeds when an additive feature is found in an owner list after the subtractive feature being analyzed. In case of interaction, the additive features involved are also identified.

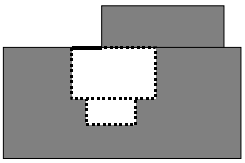
Closure interaction

This interaction class is characterized by the occurrence of a (group of interacting) subtractive feature(s) whose (compound) volume becomes a closed void inside the model.

In case of *single* closure, there is only one feature shape involved and, hence, a necessary and sufficient condition is that its whole shape boundary is completely present on the model boundary, i.e. it has no subtractive cell faces. In *multiple* closure, however, such cell faces can occur on the closed features' boundaries, but only separating their overlapping volumes, see Figure 6.2.a. Therefore, the detection Algorithm 6.5 exits as soon as it finds one subtractive cell face of these boundaries that is not separating two subtractive cells, but instead opens to the exterior, as depicted in Figure 6.2.b. In addition, the set of closed feature shapes is also returned, as well as the feature shape(s) causing the closure interaction.



(a) subtractive cell face between two closed pockets: void produced



(b) subtractive cell face opening to the exterior: no void produced

Figure 6.2 – Detection of multiple closure interactions

```
closedShapes ← s ∪ s.overlappingSet(SUBTRACTIVE)
for each shape si in closedShapes
   for each cell face cf in si.boundary(SUBTRACTIVE)
      if not exists cf.partner
         return FALSE
      else
   closedShapes.add(cf.partner.ownerlist.last.shape)
return TRUE
```

Algorithm 6.5 – Closure interaction detection algorithm

Absorption interaction

Absorption interactions are described in volumetric rather than in boundary terms. They occur to either an additive or a subtractive feature, whenever it ceases to contribute to the model shape. A sufficient and necessary condition is that all cells of the absorbed feature shape are contained in, i.e. owned by, one or more other interacting shapes, see Algorithm 6.6. This information is explicitly stored in the owner list of a cell, sorted according to the precedence criteria presented in Section 4.6. Additional data collected by the algorithm includes the interacting feature shape(s) causing the absorption.

```
for each cell c in s.cells
   if c.ownerlist.last = s
      return FALSE
return TRUE
```

Algorithm 6.6 – Absorption interaction detection algorithm

Geometric interaction

Geometric interactions on a subtractive feature are described by a combination of volumetric and boundary conditions on shape elements. Informally, they can be described as the removal of a "slice" of the feature shape adjacent to one of its shape elements. The detection algorithm (see Algorithm 6.7) therefore, analyzes, for each shape element, the boundary of all shape cells in its neighborhood, and exits as soon as it finds one such cell bounded by additive cell faces, as depicted in Figure 6.3.b. It collects and returns also the feature parameter(s) involved, as well as the feature(s) causing the interaction.

The "amount" of geometric interaction, i.e. the computation of the actual parameter value shown, requires additional queries: determination (i) of the parameter related with the shape element (queried from the Constraint Manager), (ii) of the respective direction (also queried from the Constraint Manager), and (iii) of the dimension of the remaining shape volume in that direction (queried from the Feature Geometry Manager).

Transmutation interaction

Transmutation interactions are analogous to geometric interactions, in that they also act on a shape element. For a shape element e, its *semantic nature* is defined by the following predicate:

$$\texttt{e.semanticNature} = \begin{cases} \texttt{ADDITIVE}, & \text{if e has a semantic constraint of type } \textbf{onBoundary} \\ \texttt{SUBTRACTIVE}, & \text{if e has a semantic constraint of type } \textbf{notOnBoundary} \\ \texttt{NIL}, & \text{otherwise} \end{cases}$$

(a) grey cell has only subtractive boundary cell faces: geometric interaction

(b) grey cell has additive boundary cell faces: no geometric interaction

Figure 6.3 – Analyzed cells adjacent to the blind hole top

```
for each shape element e in s.elements
   geom_int ← TRUE
   for each cell face cf in e.cellFaces
      for each cell face cf_i in cf.cell.boundary
         if cf_i.nature = ADDITIVE
            geom_int ← FALSE
            exit
      if not geom_int
         exit
   if geom_int
      return TRUE
return FALSE
```

Algorithm 6.7 – Geometric interaction detection algorithm

With a transmutation, none of the cell faces of a shape element has its semantic nature, see Algorithm 6.8. Shape elements on which no semantic constraints are specified, meaning that their presence/absence on the model boundary is irrelevant for feature semantics, cannot be thus subject to this interaction class. To determine the new class of the transmutated feature, a dedicated module is used that performs incremental identification of features in the cellular model, see (de Kraker 1997). The shape element with the unsatisfied semantic constraint, and the identified feature class of the transmutated feature are also returned by the algorithm.

```
for each shape element e in s.elements
   n ← e.semanticNature
   if n ≠ NIL
      transm_int ← TRUE
      for each cell face cf in e.cellFaces
         if cf.nature = n
            transm_int ← FALSE
            exit
      if transm_int
         return TRUE
return FALSE
```

Algorithm 6.8 – Transmutation interaction detection algorithm

```
semanticConstraints ← s.constraints(SEMANTIC)
for each sc in semanticConstraints
   if not sc.check
      return TRUE
return FALSE
```

Algorithm 6.9 – Topologic interaction detection algorithm

Topologic interaction

Topologic interactions on a given feature are detected by simply checking the semantic constraints that operate on its feature elements, see Algorithm 6.9. The feature element and the type of semantic constraint violated, as well as the feature causing the violation, are also identified and returned.

6.3 Discussion

An interesting issue in interaction detection is that of multiple interactions arising from one modeling operation. The Interaction Manager detects and collects information about interactions taking place in all features in the FIS. These can then be reported and handled according to

the scheme described in Chapter 7, in order to overcome the invalid situation.

However, as remarked in Chapter 5, the interaction classes identified there are not mutually exclusive, i.e. it might occur that the same feature instance undergoes several interactions simultaneously. For example, an absorption interaction can be regarded as an extreme case of geometric (or even transmutation) interaction, and a boundary clearance interaction as an extreme case of semantic interaction. It is possible that a feature specifies interaction constraints in such pairs.

In the current implementation, the Interaction Manager checks, for each feature, all its interaction constraints in the order given in the previous section: the most "severe" first. For each feature, the interaction detection process is stopped as soon as one interaction constraint is violated. This means that other possible interactions on the same feature are ignored, and only the one detected will be reported. This approach has been intentionally chosen, considering that *one* (disallowed) interaction occurring with a feature is sufficient to have the user notified and asked to correct the modeling operation, in order to recover model validity.

The fact that interaction classes may overlap, and the possibility to further extend the Interaction Manager with new interaction classes, seem also to encourage this scheme, as it avoids dealing with unpredictable combinations of different interactions on the same feature.

7

Feature model validity maintenance

Embedding validity criteria in each feature class, as described in Chapter 3, can significantly enhance the modeling process, as it guarantees that the semantics of each feature instance created in the model effectively matches the specific requirements of its feature class. As has been stated before, one of the basic ideas of feature modeling is that functional information can be associated to shape information in a feature model. However, the usefulness of this association is voided if, for example, the modeling system would allow a modeling operation to significantly modify the shape imprint of a feature, once added to the model with a specific intent. In other words, arbitrarily modifying the semantics of a feature should be disallowed if one wants to make feature modeling really more powerful than geometric modeling.

The goal of this chapter is to present the validity maintenance scheme of the semantic feature modeling approach, and to show that consistently maintaining model validity effectively raises the level of assistance provided by the feature modeling system. First, the main principles of validity maintenance are discussed (Section 7.1). Next, the two phases of validity maintenance are further elaborated: validity

checking (Section 7.2) and validity recovery (Section 7.3). Finally, an example modeling session is described, illustrating which high-level user assistance is provided under this approach (Section 7.4).

7.1 Validity maintenance

Feature model validity maintenance is the process of monitoring each modeling operation in order to ensure that all features conform to the semantics specified in their respective classes. Maintaining feature model validity throughout the modeling process requires not only managing all its constraints, but also assessing the conformity of each feature in the model with its validity criteria. This guarantees that all aspects of the design intent once captured in the model are permanently maintained.

The two basic principles of validity maintenance can be summarized as follows:

1. A modeling operation, to be considered as *valid*, should yield a feature model that conforms to all constraints.

   This ensures that every feature in the model conforms to the designer intent explicitly specified up to that moment.

2. After an invalid modeling operation, the user should be assisted in overcoming the constraint violations in order to recover model validity again.

   This can reduce the frequency of backtracking by enlarging the choice of possible reactions towards validity recovery. In particular, explanations on what is causing a constraint violation, and context-sensitive corrective hints, can significantly improve the modeling process.

In the SPIFF modeling system, validity maintenance tasks are performed by the Feature Model Manager described in Section 2.3. These can be classified into two types of tasks:

validity checking, performed at key stages of each modeling operation; and

validity recovery, performed when a validity checking task detected a violation of some validity criterion.

These are now separately discussed in the next two sections.

7.2 Validity checking

As mentioned above, the first basic principle of model validity maintenance is that a valid modeling operation should entirely preserve the designer intent specified so far with each feature, as well as with all model constraints. In other words, after a valid modeling operation, the feature model conforms to all its constraints.

Modeling operations were introduced in Section 4.4, and classified into two major categories:

feature operations, which include adding a new feature instance to the model, and editing or removing an existing feature instance;

constraint operations, which include adding new model constraint instances, and editing or removing existing model constraints.

A generic scheme of a modeling operation is presented in Figure 7.1 on the next page, showing its main internal steps. Also shown in the diagram are the various points at which the operation can turn out to be invalid. Whenever this occurs, the operation branches into the *reaction loop*, instead of following the normal flow, and we say the model has entered an *invalid* state. We now concentrate on the description of the main steps in the diagram, and on the circumstances under which specific invalid situations may arise in each of these steps. An important goal here is to enter the reaction loop, if required, with sufficient knowledge of the current status of the model, so that it can be appropriately handled, reported to the user and, ultimately, overcome. The reaction loop itself will be dealt with in the next section.

Dependency analysis

This step is only required by the removal of a feature from the model. The removal of a feature f is only allowed if f has no dependent entities (features or model constraints) in the Feature Dependency Graph (otherwise, such dependent entities would be left referring to a non-existing graph node). In case there are entities dependent on f (see Section 4.1), they are collected and the operation enters the reaction loop.

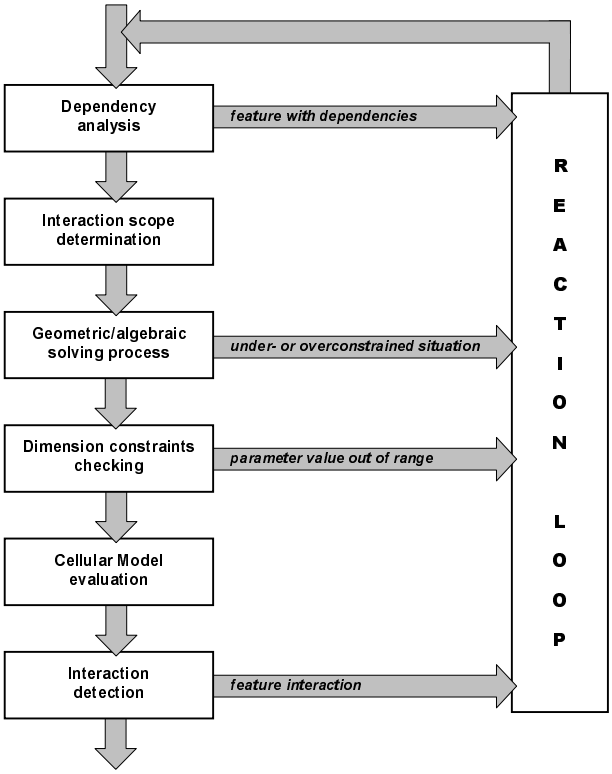


Figure 7.1 – Generic scheme of a modeling operation

Interaction scope determination

The determination of the feature interaction scope (FIS) is performed at this stage, as described in Section 6.1. Its purpose is to optimize the interaction detection phase (the last step in Figure 7.1), by avoiding checking features that are known in advance to be left unaffected by the operation.

Geometric and algebraic solving process

This step is required by all modeling operations, except feature removal. Its goal is to determine or update the dimensions, position and orientation of all features in the model. This task is performed by the Constraint Manager, which deploys two dedicated constraint solvers: a geometric constraint solver based on extended 3D degrees of freedom analysis (Kramer 1992), and a SkyBlue algebraic constraint solver (Sanella 1992). The iterative cooperation of these solvers, under the control of the Constraint Manager, is described by Dohmen (1997).

At this stage, modeling operations are considered invalid if this solving process detects:

1. an *overconstrained situation*, i.e. some feature(s) has (have) conflicting geometric and/or algebraic constraints; or
2. an *underconstrained situation*, i.e. the features and/or model constraints specified, with the parameter values provided by the user, are not sufficient to uniquely determine and fix the degrees of freedom of all features in the model (Noort et al. 1998).

In both cases, the operation enters the reaction loop.

Dimension constraints checking

When the solving process is successfully concluded, all feature shape dimensions have their values assigned, and checking of all dimension constraints takes place. The modeling operation is considered invalid if some feature dimension parameter is out of the range specified by the respective constraint.

Cellular Model re-evaluation

When this step is reached, each feature in the Feature Dependency Graph has all its parameters successfully updated. In particular, all feature shape extents have their dimensions, position and orientation fully determined. The Cellular Model may thus be updated, so that the effects of the operation are also reflected in the evaluated geometric model. This process has already been described in Section 4.5.

Interaction detection

Once the Cellular Model has been updated, detection of disallowed feature interactions takes place. At this stage, a modeling operation is considered invalid if any semantic or interaction constraint is violated, for some feature in the FIS, previously determined. The interaction detection process has been described in detail in Chapter 6.

7.3 Validity recovery

When a modeling operation is invalid, for any reason pointed out in the previous section, a valid model should be achieved again. This is straightforward if the modeling operation is cancelled: all that is needed is to backtrack to the valid model state just before executing it, by "reversing" the invalid operation. According to their type, invalid modeling operations are reversed as follows:

Adding a new feature instance to the model The added feature is removed from the model, using the feature removal operation.

Removing a feature instance from the model The removed feature is added back to the model, using the feature adding operation with the original parameter values.

Editing a feature instance in the model The original parameter values of the edited feature are restored, using another feature editing operation, in all regards similar to the first operation.

Constraint operations Each of them is reversed similarly to the feature operations (i.e. added constraints are removed, edited constraints are restored, etc.).

Reversing a modeling operation can be done very efficiently under our approach. The parameter values possibly required for undoing each modeling operation are kept in a log, the so-called *operations stack*. Every modeling operation is registered in this stack, and marked according to whether it led the model to a valid state or not. Undoing is therefore always possible, at any moment in a modeling session, by popping operations from the stack and executing their reverse operation *until* a "valid state" marker is found. This is illustrated in Figure 7.2: assuming the insertion of the stiffener is invalid, because of turning the through hole

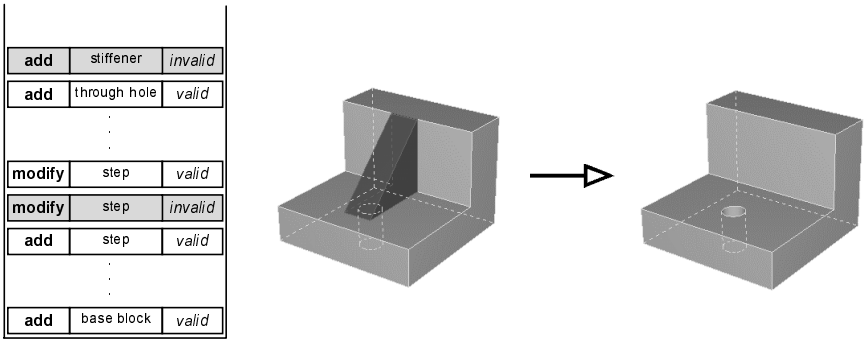


Figure 7.2 – Undo mechanism using the operations stack

into a blind hole, that operation (the last on the operations stack) is popped from the stack and undone to restore the original situation.

However, to always have to recover from an invalid operation by undoing it is too rigid. It is often much more effective to constructively assist the user in overcoming the constraint violations, after an *invalid* modeling operation, in order to recover model validity again. In most cases, if the user receives appropriate feedback on the causes of an invalid situation, it is likely that corrective actions other than undoing, which restore model validity as well, might preferably be chosen.

We call this process *validity recovery*, and it emphasizes the importance of a user dialog in terms of features and their semantics. Validity recovery includes reporting to the user constraint violations, documenting their scope and causes, and, whenever possible, providing context-sensitive corrective hints.

To achieve this, a corrective mechanism was devised –the *reaction loop*, represented in Figure 7.1–, which is activated whenever an operation turns out to be invalid. The user can then specify several modeling operations in a batch (typically editing features and/or model constraints), and execute them, in order to overcome the invalid model situation. Execution of these *reaction operations* follows the same scheme of Figure 7.1, which means that their outcome is analyzed, checking for validity at each step, just as for "direct" modeling operations. The reaction loop is only exited when, as a result of the specified reactions, all constraints are satisfied again. At any stage when the model is invalid, the

user may give up attempting to fix it by specifying more reactions, and backtrack to the last valid stage (i.e. right before the operation that entered the reaction loop). Again, undo is here possible because all reaction operations executed are also pushed onto the operations stack, and can thus be reversed.

The specification of reaction operations is assisted by automatically generated hints, which document each constraint violation detected and support the validity recovery process. Documentation of constraint violations varies with the operation step at which the reaction loop is entered, and with the type of constraint involved. Referring to the scheme of Figure 7.1, we have:

Dependency analysis The user is presented a list of all entities that depend on the feature *f* to be removed, in order to decide how to handle each of them. For example, the user might choose to remove with *f* some of its dependent entities, but to modify others, by making them dependent on another feature.

Geometric and algebraic solving process For both over- and underconstrained situations, the reaction loop notifies the user of where the conflict was found, highlighting the features involved in a viewing camera. The user can then make the appropriate corrections (typically, modifying some of the features or constraints involved).

Dimension constraints checking The user is notified about the particular feature and parameter where the conflict was found, as well as about the admissible range for that parameter.

Interaction detection For each interaction detected, the user is notified of its causes (mostly the features creating the interaction), and of its effects (e.g. a feature face or parameter affected). According to the particular interaction type, specific reaction choices may also be given. Examples of these are:

- **transmutation interaction**: replace the transmutated feature by another feature instance of the identified feature class (for example, after adding the stiffener to the model in Figure 7.2, the user might replace the through hole feature instance by a blind hole feature instance);
- **geometric interaction**: re-attach the feature affected, by replacing its attach reference face with a parallel face of the

feature causing the interaction (an example of this is given in the next section);

- **absorption interaction**: remove from the model the absorbed feature;
- **splitting interaction**: replace the split feature by two (or more) instances of the appropriate feature class(es).

In all cases above, the scope of the reaction choices made available to the user is restricted to those features and model constraints that are somehow involved in the invalid situation (i.e. features that overlap or have a dependency relation with the affected feature). This further helps the user in concentrating validity recovery efforts on effective and meaningful reactions.

7.4 Example modeling session

The usefulness of the validity checking and recovery mechanisms is illustrated in this section with examples taken from a modeling session with the SPIFF system.

The user starts the modeling session with opening a new model (see Figure 7.3.a, on the next page). He then defines a base block, and creates on it two blind slots (see Figure 7.3.b). The subsequent modeling steps are now described. For each step, the invalid situation reported occurs because the underlying feature classes do specify the validity criteria violated at that stage.

Step 1 (Figure 7.4) The user creates a step in the model, but the step width is such that it overlaps with the blind slots, turning them into through slots. The system detects these transmutation interactions and documents the occurrence, reporting that the step is absorbing the slots' back faces. As a reaction to the invalid situation, the user decides to decrease the length of the two blind slots, and also to reduce the step depth, obtaining the model in Figure 7.5.

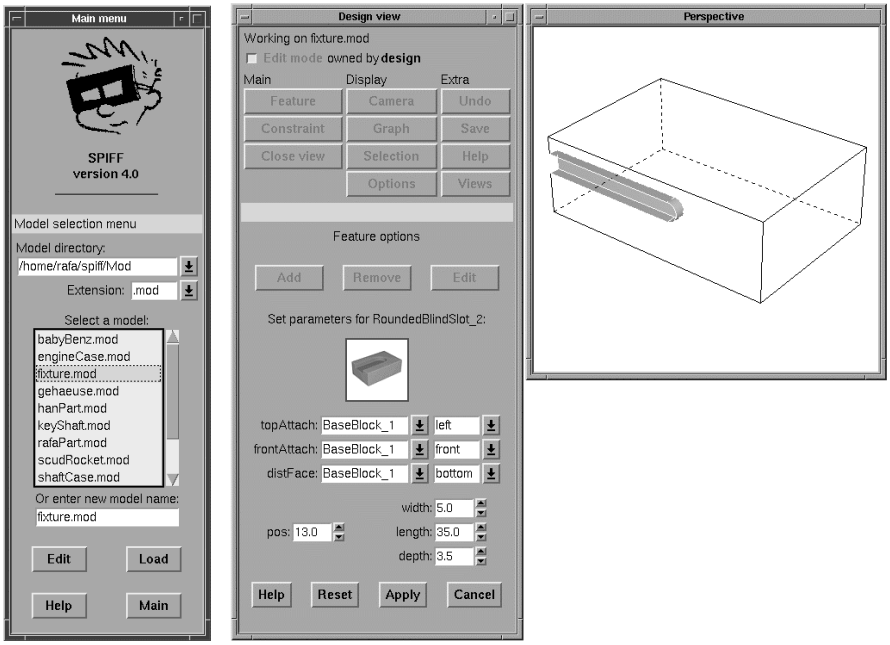


Figure 7.3 – Starting a modeling session in SPIFF

Step 2 (Figure 7.5) The user creates a through slot, attached to the top face of the block. Because of the slot depth value chosen, a topologic interaction takes place, again affecting the two blind slots. After this is reported by the system, the user overcomes the interaction by displacing the two blind slots downward, obtaining the model in Figure 7.6.

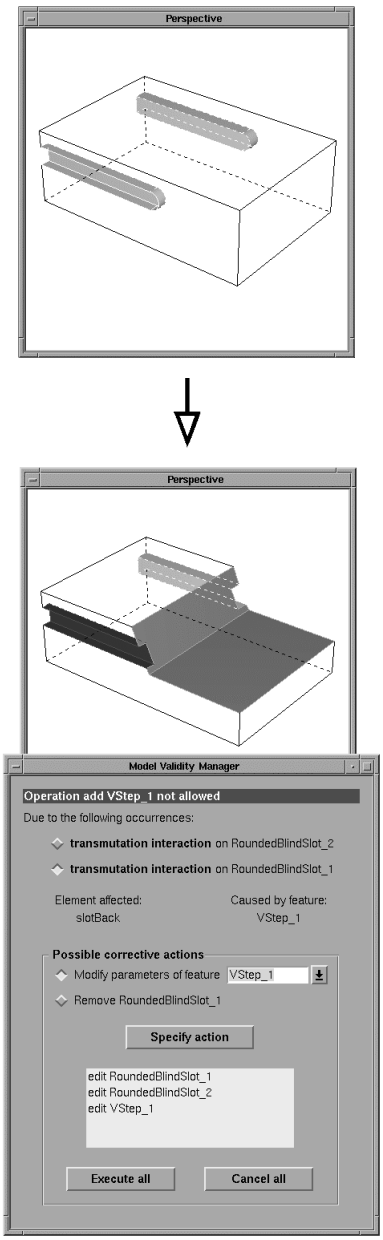


Figure 7.4 – Step 1: reporting a transmutation interaction

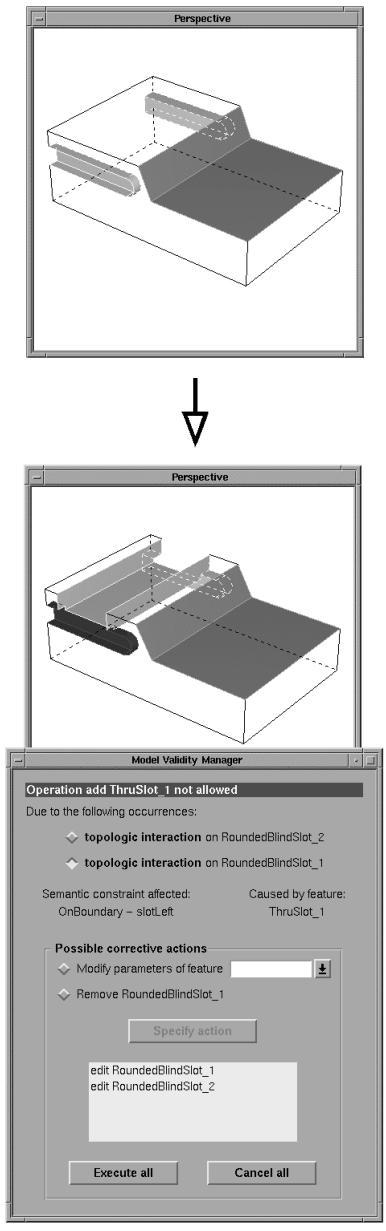


Figure 7.5 – Step 2: reporting a topologic interaction

Step 3 (Figure 7.6) Next, the user attaches a rib feature to the bottom of the through slot. The rib feature class, however, prescribes a minimum width value, not obeyed by this instance, thus the system reports a dimension constraint violation. The user corrects this by adjusting the rib width to the minimum value allowed, as shown in the model of Figure 7.7.

Step 4 (Figure 7.7) Subsequently, the user attempts an alternative design for the part, re-attaching the through slot from the top of the block to the bottom of the step. Consequently, the rib feature, which is dependent on the through slot, is also displaced with it. However, the upper region of the rib intrudes into the subtractive volume of the step. This is disallowed by the validity criteria of the step (by means of a volumetric clearance interaction constraint), thus the operation is notified as invalid, and the situation is reported to the user.

To recover from this interaction, the system suggests modifying the rib and/or the through slot. In this case, the user opts for increasing the slot depth.

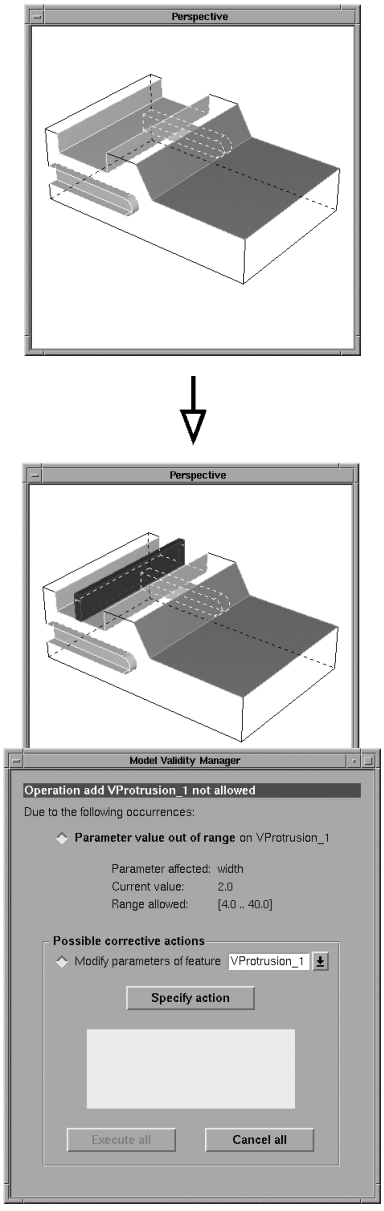


Figure 7.6 – Step 3: reporting a dimension constraint violation

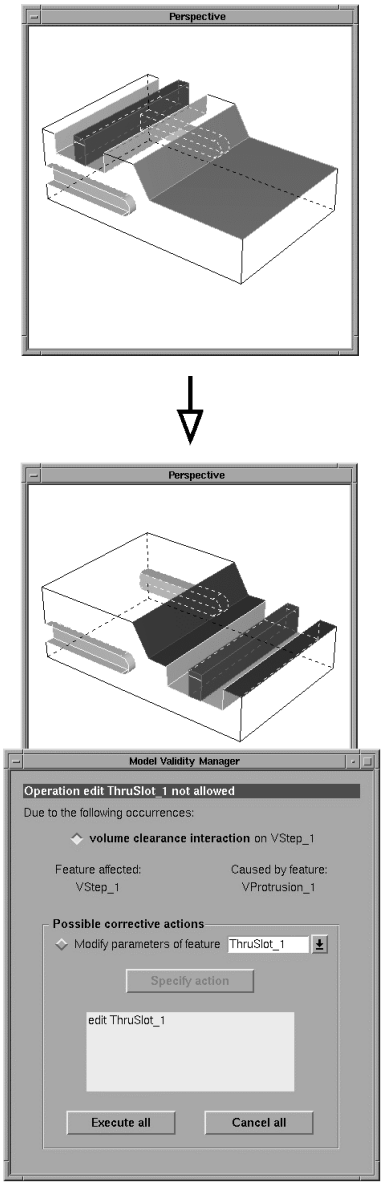


Figure 7.7 – Step 4: reporting a volume clearance interaction

Step 5 (Figure 7.8) By mistake, the user supplies too high a value for the slot depth, causing the model to become disconnected. Although the previous clearance interaction on the step is indeed overcome, now a new invalid situation –the model disconnection– occurs and is reported. As a reaction to this, the user may readjust the slot depth, specify a larger height for the block, or decrease the step depth (or a combination of these reactions). In this case, he chooses for decreasing the slot depth.

As remarked in the previous section, features that are irrelevant to overcome the invalid situation, for example the two blind slots, are not editable at this stage of the reaction loop.

Step 6 (Figure 7.9) At this stage, the user chooses for a variant of the part without the step feature, and issues its removal from the model. Because the through slot is dependent on the step, and thus indirectly also the rib, the system requires these dependencies to be eliminated prior to removing the step. Removal of the dependent features from the model and modification of their attaches are among the possible reactions suggested by the system. The user chooses to re-attach the through slot to the top face of the block, by which its dependent rib is also automatically displaced, as shown in Figure 7.10.

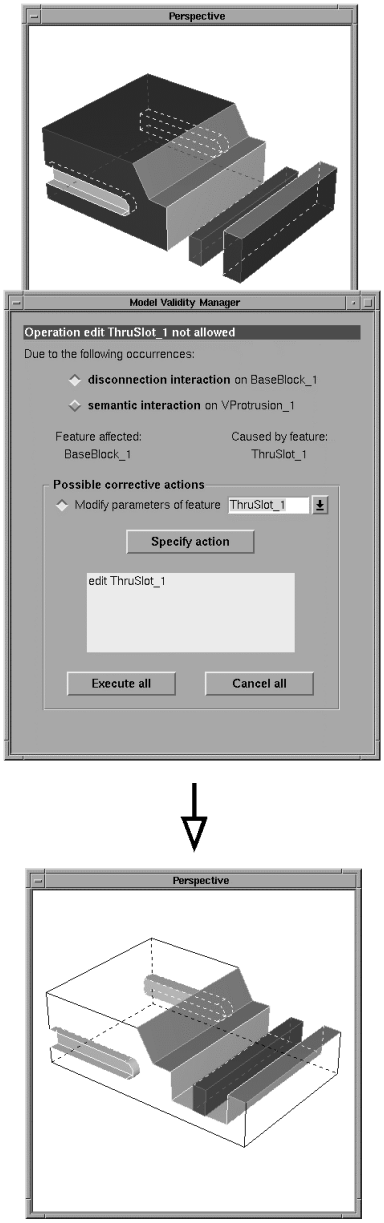


Figure 7.8 – Step 5: reporting a disconnection interaction

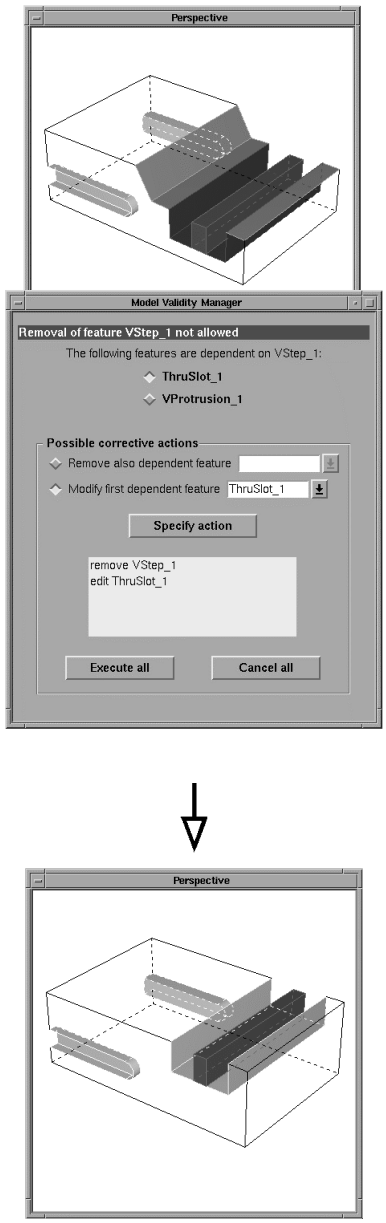


Figure 7.9 – Step 6: reporting dependencies before a feature removal

Step 7 (Figure 7.10) The user proceeds with the design by attaching a through hole between the top and bottom faces of the block. By mistake, however, the two model faces chosen for positioning the through hole are parallel (the front and back faces of the block), and thus insufficient to determine its position. The underconstrained situation is reported to the user, who is asked to specify appropriate reference faces for positioning the through hole, after which a valid model is achieved again.

Step 8 (Figure 7.11) Finally, the user creates a pocket at the bottom face of the block, such that the through hole attached to it in the previous step becomes shorter. This geometric interaction is detected and reported by the system. The user reacts by re-attaching the through hole to the bottom face of the pocket, and takes the opportunity to slightly increase the depth of the pocket. The final model is shown in Figure 7.12.

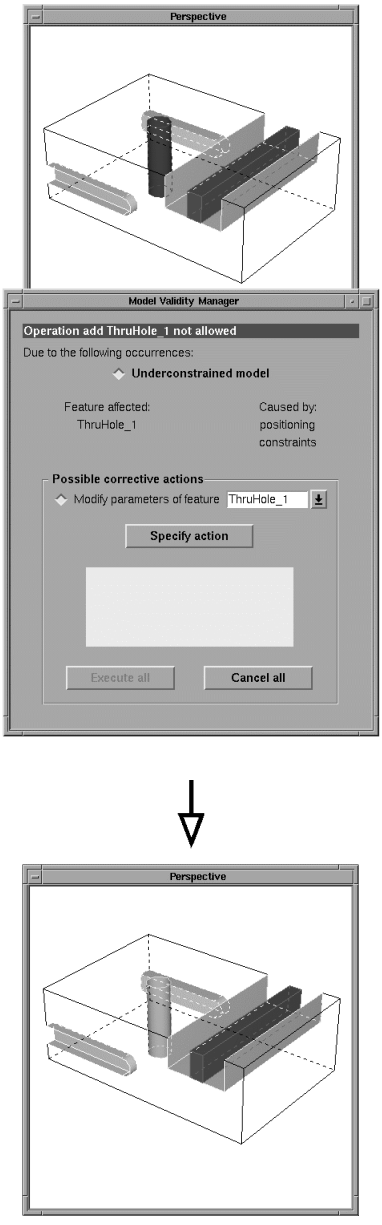


Figure 7.10 – Step 7: reporting an underconstrained situation

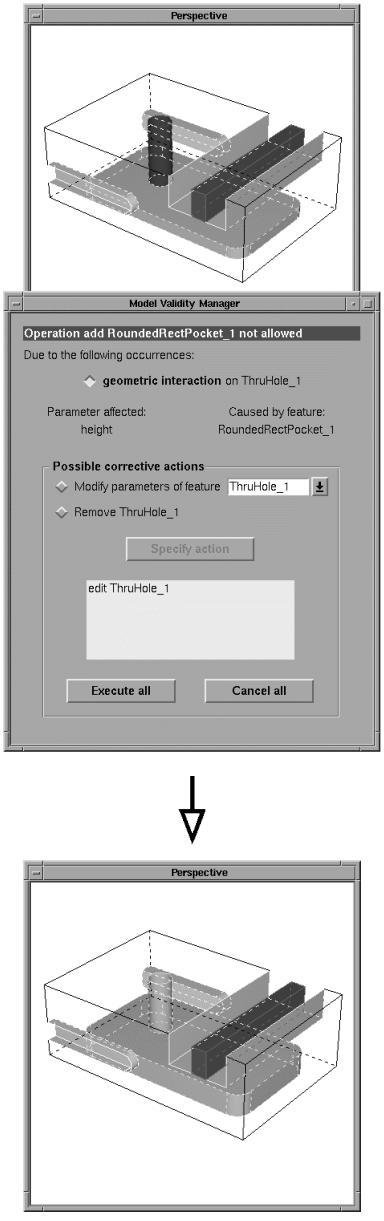


Figure 7.11 – Step 8: reporting a geometric interaction

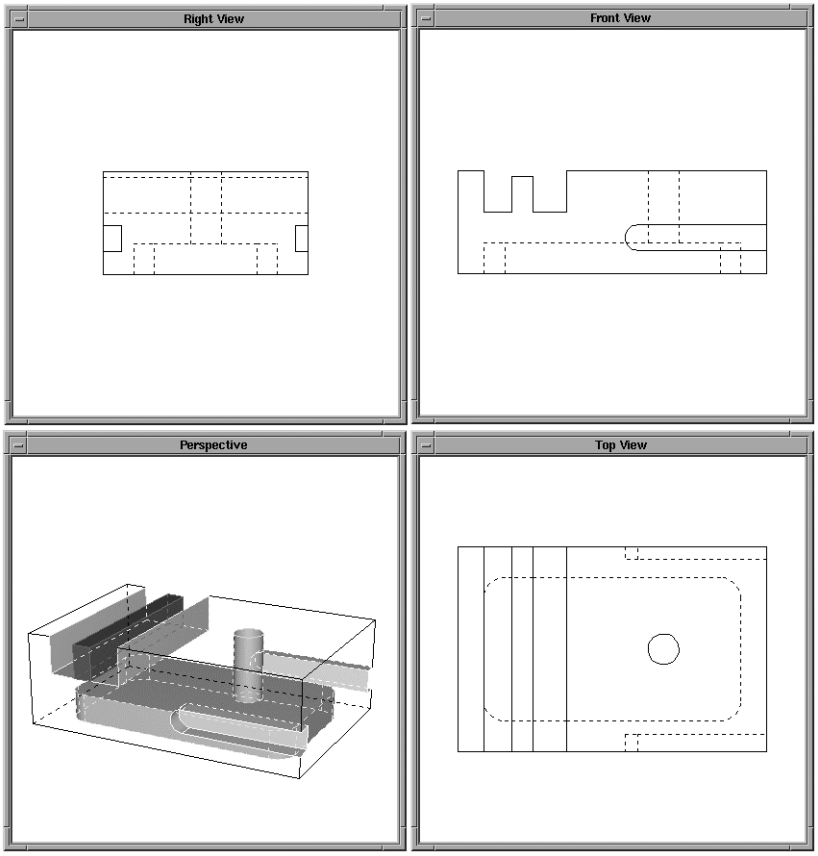


Figure 7.12 – Final model

7.5 Conclusions

Maintaining the meaning, or semantics, of features in a feature model – so-called *validity maintenance*– has been addressed in this chapter.

The validity maintenance scheme presented overcomes several drawbacks of current feature modeling systems. This is achieved by main-

taining all constraints throughout model editing with constraint solving techniques. A validity recovery mechanism analyzes any invalid situation that might result from some modeling operation, and gives the user explanations and hints to overcome this. The user gets thus valuable assistance in creating valid models only, containing features with well-defined semantics only.

Application of this approach has been exemplified with a variety of modeling situations. From these it can be concluded that maintenance of feature model validity using a consistent feature vocabulary is not only possible, but indeed effectively provides user assistance at a much higher level than current feature modeling systems do.

8

Conclusions

At the end of this thesis, we make some global considerations on the semantic feature modeling approach. First, some research directions for future work are outlined (Section 8.1). Finally, concluding remarks on the research reported are presented, comparing semantic feature modeling to other current feature modeling approaches (Section 8.2).

8.1 Future research

The ability of the Feature Library Manager to effectively create and manage a hierarchical library structure supports the process of deriving variants of a given feature class. Creation of such variants, however, can easily lead to an explosion in the number of feature classes available in the library, due to the combinatorial possibilities in this refinement process. In the current implementation, this has been alleviated by offering the user of the Feature Modeler the possibility to fine-tune some validity conditions of a feature instance, in order to take into account, or not, those criteria. This ability is not always desirable, and other alternatives should be investigated.

The implemented Feature Library Manager uses a fixed set of constraints in each constraint category (geometric, algebraic, etc.). New constraint classes can be added to each category by inheritance and/or composition of existing classes, but they need to be manually coded. This could be improved by means of a Constraint Library Manager, aimed at automating this task, similarly to what the Feature Library Manager provides for feature classes.

Related to the latter is the fact that the Interaction Manager requires dedicated interaction detection algorithms for each interaction class. Again, extension of the set of interaction classes to other, possibly domain-specific interactions, is possible, due to the declarative and modular implementation of this approach, but coding of the new detection algorithms is required.

The definition of feature interaction developed in this thesis requires overlap between feature shapes. Two possible extensions of this notion deserve further investigation. The first extension concerns the influence exerted on one feature by another feature in its "neighborhood". For example, in manufacturability analysis, tool path clearance may require, say, an access region "in front" of a blind hole, in addition to the clearance entrance face criterion currently supported. A solution for such cases might include the specification of additional *clearance* volumes in the feature class, which would then be represented in the Cellular Model, and processed accordingly (e.g. with the current "volumetric clearance interaction" criterion).

The second extension deals with a broader notion of interaction as the "mutual influence between features, whose semantics are somehow in conflict". An example of this, in an assembly modeling context, is a conflict in the degrees of freedom left by a number of connection features between two parts. Ongoing research in our group includes work in this direction (Noort and Bronsvoort 1999), benefiting from previous work carried out on assembly modeling (van Holland 1997).

Regarding the validity maintenance scheme, the possibility of, on demand, deferring validity checking should be carefully considered. Moreover, the reactions currently provided for validity recovery could be enhanced in several directions. Firstly, more information might be provided about the causes of some invalid situations (e.g. when an interaction is indirectly caused via cascaded geometric constraints among several features). Secondly, more concrete hints might be given, in some cases, to recover from specific feature interactions (e.g. suggest a particular fea-

ture face for re-attachment after a geometric interaction). Thirdly, the choice of reaction operations could itself be extended, based on domain-specific information.

The SPIFF modeling system supports multiple interpretations, or *views*, on the same product. Each view has its own feature model, containing features relevant for a specific product development activity (e.g. design, manufacturing planning or assembly planning). When several views on a product are open, the system maintains the consistency among their feature models, using feature conversion techniques (de Kraker 1997). Currently, when a modeling operation in one view leads to some unrecoverable inconsistency with another view, either the latter has to be closed and discarded, or the operation is rejected and undone.

The validity maintenance scheme presented in this thesis is generic, providing the same high-level assistance to the user of the modeling system, regardless of which particular view of the product he is working on. It would be interesting to investigate whether the multiple-view consistency framework just described could benefit from the validity maintenance scheme proposed in this thesis. On the one hand, better user assistance could then be provided when, for example, a modeling operation on one view of the model would cause some invalid situation in another opened view. On the other hand, the question arises on which terms (i.e. of features of which view) that user assistance could (or should) be provided: none of the views alone is likely to have the necessary and sufficient information for that, and some combined strategy might have to be developed.

Finally, other refinements in the semantic feature modeling approach can be expected in the future, benefiting, among other things, from further assessment of its usability in practice.

8.2 Concluding remarks

There are several important characteristics that distinguish the semantic feature modeling approach from current feature modeling approaches, in particular the history-based feature modeling approach. In this section, these approaches are compared on their merits.

The most salient characteristic of semantic feature modeling is that the semantics of all features is well defined and maintained during the whole

modeling process. The use of various constraint types for validity conditions in generic feature classes allows a complete definition of all semantic aspects of the instances of each class. Among these constraints, those specifying admissible feature interactions are of particular interest. User-added constraints can further assist in capturing the user intent in a model. Once specified, all constraints are maintained throughout model editing with constraint solving methods. A mechanism is provided to detect and analyze any invalid situation that might result from some modeling operation, and to give the user an explanation and hints to overcome this. So the user gets valuable assistance in creating valid models only, containing features with well-defined semantics only.

It might be argued that imposing rigid validity rules reduces the modeling freedom of the user. For example, the user might actually want to turn blind a through hole. In current feature modeling approaches, this can be achieved by simply closing one of the hole's entrance faces, e.g. with a protrusion, without the system objecting to this. Therefore, even if no blind hole feature class is available in the feature library, a blind hole can be created by such a geometric construction. In the semantic feature modeling approach, on the other hand, this modification can only be made by adding the protrusion and, after the system objecting against the transmutation of the through hole into a blind hole, explicitly changing the through hole into a blind hole. However, this reaction is only possible if a blind hole class is available in the feature library of the system. Thus, only models with features that are permitted, by their inclusion in the feature library, can be created. This example demonstrates how in semantic feature modeling, by imposing restrictions on the modeling freedom, the user is assisted in creating valid models only.

A correspondence with programming languages can be noted here. Geometric modeling, but also current feature modeling practice, is in a way comparable to the use of low-level programming languages, with low-level operations. These offer large freedom in what can be programmed but, as a consequence, errors can easily occur, and the programmer has much responsibility in getting a program correct. Semantic feature modeling, on the other hand, is comparable to the use of high-level programming languages, such as object-oriented languages, providing high-level operations with well-defined and powerful semantics. Practice has shown that programs in such languages contain fewer errors, i.e. correspond more to the user intent of the programs. The loss of "programming freedom" in these languages is commonly accepted because of the advantage of being forced to create more meaningful, error-free programs. Similarly, we believe that the loss of modeling freedom in

semantic feature modeling is acceptable because of the advantage of being forced by the system to create more meaningful models.

In addition to offering much better facilities for specifying and maintaining feature semantics in models, semantic feature modeling solves several other problems that occur in history-based feature modeling operations. In particular, there is no longer a dependency on the chronological order in which features are added to a model. This means an improvement in model modification and dimensioning facilities. In addition, shortcomings originating from the persistent naming problem in history-based modeling are avoided, because all modeling operations work on feature faces, instead of boundary model faces, and, therefore, ambiguities in names cannot occur. Stated differently, in semantic feature modeling the semantics of modeling operations is well defined, in contrast with history-based feature modeling.

The Cellular Model has several properties that make it very suitable for the geometric representation of feature models. In particular, the generation and interpretation of the Cellular Model is independent of the chronological order of feature creation. Furthermore, subtractive and overlapping features can be dealt with in a consistent way during model editing.

Because of the complexity of the Cellular Model, the question of the efficiency of operations on this non-manifold model is important. The structure of the Cellular Model is certainly more complex than that of a manifold boundary representation, normally used in history-based feature modeling. However, this is largely compensated by the performance improvement of Cellular Model incremental evaluation. In history-based modeling systems, re-evaluation of the boundary model has a computational cost that is proportional to either the total number of features in the model, if no intermediate evaluated models are stored, or to the number of features created after the modified or deleted feature, if intermediate models or deltas are stored. In the semantic feature modeling approach using the Cellular Model, on the other hand, the computational cost of a modeling operation is dependent on the number of features whose geometry is affected by the operation. Usually, this number is very limited, so computational cost is minimized.

To conclude, it has often been remarked that feature modeling is nothing more than advanced geometric modeling, only offering parametric and constraint-based modeling facilities, in addition to the normal geometric modeling facilities. This thesis, however, shows that semantic feature

modeling is significantly more powerful than current feature modeling approaches, and finally can bring feature modeling to a much higher level than geometric modeling, not only with regard to applications, but also with regard to modeling facilities.

Bibliography

Autodesk (1998) *Autodesk Mechanical Desktop 2.0 User's Guide*. Autodesk, Inc., San Rafael, CA, USA

Bidarra, R. and Bronsvoort, W.F. (1996) Towards classification and automatic detection of feature interactions. In: *Proceedings of the 29th International Symposium on Automotive Technology and Automation; Dedicated Conference on Mechatronics – Advanced Development Methods and Systems for Automotive Products, 3–6 June, Florence, Italy,* Roller, D. (Ed.), Automotive Automation Limited, Croydon, England, pp. 99–108

Bidarra, R. and Bronsvoort, W.F. (1999a) Validity maintenance of semantic feature models. To be published in: *Proceedings of Solid Modeling '99 – Fifth Symposium on Solid Modeling and Applications, 9–11 June, Ann Arbor, MI, USA*, Bronsvoort, W.F. and Anderson, D.C. (Eds.), ACM Press, New York

Bidarra, R. and Bronsvoort, W.F. (1999b) Semantic feature modeling. Submitted for publication

Bidarra, R. and Bronsvoort, W.F. (1999c) History-independent boundary evaluation for feature modeling. Submitted for publication

Bidarra, R., Dohmen, M. and Bronsvoort, W.F. (1997) Automatic detection of interactions in feature models. In: *CD-ROM Proceedings of the 1997 ASME Design Engineering Technical Conferences, 14–17 September, Sacramento, CA, USA,* ASME, New York

Bidarra, R., Idri, A., Noort, A. and Bronsvoort, W.F. (1998a) Declarative user-defined feature classes. In: *CD-ROM Proceedings of the 1998 ASME Design Engineering Technical Conferences, 13–16 September, Atlanta, GA, USA*, ASME, New York

Bidarra, R., de Kraker, K.J. and Bronsvoort, W.F. (1998b) Representation and management of feature information in a cellular model. *Computer-Aided Design* **30**(4): 301–313

Bidarra, R. and Teixeira, J.C. (1993) Intelligent form feature interaction management in a cellular modeling scheme. *Proceedings of Solid Modeling '93 – Second Symposium on Solid Modeling and Applications, 19–21 May, Montreal, Canada*, Rossignac, J.R., Turner, J. and Allen, G. (Eds.), ACM Press, New York, pp. 483–485

Bidarra, R. and Teixeira, J.C. (1994) A semantic framework for flexible feature validity specification and assessment. In: *Proceedings of the 1994 ASME Computers in Engineering Conference, September, Minneapolis, MN, USA,* Ishii, K., Bannister, K. and Crawford, R. (Eds.), ASME, New York, Vol. 1, pp. 151–158

Bronsvoort, W.F., Bidarra, R., Dohmen, M., van Holland, W. and de Kraker, K.J. (1997) Multiple-view feature modelling and conversion. In: *Geometric Modeling: Theory and Practice – The State of the Art,* Strasser, W., Klein, R. and Rau, R. (Eds.), Springer, Berlin, pp. 159–174

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

Brunetti, G., De Martino, T., Elter, H. and Falcidieno, B. (1996a) Modeling shape and semantics through an intermediate model. In: *Proceedings of the 29th International Symposium on Automotive Technology and Automation; Dedicated Conference on Mechatronics – Advanced Development Methods and Systems for Automotive Products, 3–6 June, Florence, Italy,* Roller, D. (Ed.), Automotive Automation, Croydon, England, pp. 71–81

Brunetti, G., Ovtcharova, J. and Vieira, A.S. (1996b) A proposal for a feature description language. In: *Proceedings of the 29th International Symposium on Automotive Technology and Automation; Dedicated Conference on Mechatronics – Advanced Development Methods and Systems for Automotive Products, 3–6 June, Florence, Italy,* Roller, D. (Ed.), Automotive Automation, Croydon, England, pp. 117–124

Capoyleas, V., Chen, X. and Hoffmann, C.M. (1996) Generic naming in generative, constraint-based design. *Computer-Aided Design* **28**(1): 17–26

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

Dixon, J.R., Libardi, E.C. and Nielsen, E.H. (1990) Unresolved research issues in development of design-with-features systems. In: *Geometric Modeling for Product Engineering*, Wozny, M.J., Turner, J.U. and Preiss, K., (Eds.), Elsevier Science Publishers, Amsterdam, pp. 183–196

Dohmen, M. (1997) Constraint-based feature validation. PhD Thesis, Delft University of Technology, Delft, The Netherlands

Dohmen, M., de Kraker, K.J. and Bronsvoort, W.F. (1996) Feature validation in a multiple-view modeling system. In: *CD-ROM Proceedings of the 1996 ASME Computers in Engineering Conference, 19–22 August, Irvine, CA, USA,* McCarthy, J.M. (Ed.), ASME, New York

Gupta, S.K. and Nau, D.S. (1995) Systematic approach to analyzing the manufacturability of machined parts. *Computer-Aided Design* **27**(5): 323–342

van Emmerik, M.J.G.M. and Jansen, F.W (1989) User interface for feature modelling. In: *Computer Applications in Production and Engineering*, Kimura, F. and Rolstadas, A. (Eds.), Elsevier Science Publishers, Amsterdam, pp. 625–632

Henderson, M.R. (1984) Extraction of feature information from three-dimensional CAD data. PhD Thesis, Purdue University, West Lafayette, IN, USA

Hoffmann, C. and Joan-Arinyo, R. (1998) On user-defined features. *Computer-Aided Design* **30**(5): 321–332

van Holland, W. (1997) Assembly features in modelling and planning. PhD Thesis, Delft University of Technology, Delft, The Netherlands

Idri, A. (1998) User-defined object-oriented features. Master's Thesis, Delft University of Technology, Delft, The Netherlands

Karinthi, R.R. and Nau, D.S. (1992) An algebraic approach to feature interactions. *IEEE Transactions on Pattern Analysis and Machine Intelligence* **14**(4): 469–484

Klein, R. (1997a) A knowledge representation perspective on geometric modeling. In: *Geometric Modeling: Theory and Practice – The State of the Art,* Strasser, W., Klein, R. and Rau, R. (Eds.), Springer, Berlin, pp. 175–196

Klein, R. (1997b) Dependency maintenance in declarative geometry modeling. In: *Proceedings of Solid Modeling '97 – Fourth Symposium on Solid Modeling and Applications, 14–16 May, Atlanta, GA, USA,* Hoffmann, C.M. and Bronsvoort, W.F. (Eds.), ACM Press, New York, pp. 31–41

de Kraker, K.J. (1997) Feature conversion for concurrent engineering. PhD Thesis, Delft University of Technology, Delft, The Netherlands

de Kraker, K.J., Dohmen, M. and Bronsvoort, W.F. (1995) Multiple-way feature conversion to support concurrent engineering. In: *Proceedings of Solid Modeling '95 – Third Symposium on Solid Modeling and Applications, 17–19 May, Salt Lake City, UT, USA,* Hoffmann, C.M. and Rossignac, J.R. (Eds.), ACM Press, New York, pp. 105–114

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. In: *Proceedings Solid Modeling '95 – Third Symposium on Solid Modeling and Applications, 17–19 May, Salt Lake City, UT, USA,* Hoffmann, C.M. and Rossignac, J.R. (Eds.), ACM Press, New York, pp. 21–30. Also in: *Computer-Aided Design* **29**(2): 113–122

Kyprianou, L.K. (1980) Shape classification in computer-aided design. *PhD Thesis*, Cambridge University, UK

Lequette, R. (1997) Considerations on topological naming. In: *Product Modeling for Computer Integrated Design and Manufacturing – Proceedings TC5/WG5.2 International Workshop on Geometric Modeling in Computer Aided Design, 19–23 May 1996, Airlie, USA*, Pratt, M., Sriram, R.D. and Wozny, M.J. (Eds.), Chapman & Hall, London, pp. 394–403

Luby, S.C., Dixon, J.R. and Simmons, M.K. (1986) Designing with features: creating and using a features data base for evaluation of manufacturability in castings. In: *Proceedings of the 1986 ASME Computers in Engineering Conference, August, Chicago, IL, USA*, ASME, New York

Mandorli, F., Cugini, U., Otto, H.E. and Kimura, F. (1995) Reflective control of attributed entities in feature-based CAD systems using a CARW system manager. In: *Preprints of the IFIP WG5.2 Workshop on Knowledge Intensive CAD–1, Espoo, Finland, September*, Tomiyama, T., Mäntylä, M. and Finger, S. (Eds.), pp. 217–244

Mandorli, F., Cugini, U., Otto, H.E. and Kimura, F. (1997) Modeling with self-validating features. In: *Proceedings of Solid Modeling '97 – Fourth Symposium on Solid Modeling and Applications, 14–16 May, Atlanta, GA, USA,* Hoffmann, C.M. and Bronsvoort, W.F. (Eds.), ACM Press, New York, pp. 88–96

Noort, A. and Bronsvoort, W.F. (1999) Enhanced multiple-view feature modelling. Submitted for publication

Noort, A. (1997) Solving over-constrained geometric models. Master's Thesis, Delft University of Technology, Delft, The Netherlands

Noort, A., Dohmen, M. and Bronsvoort, W.F. (1998) Solving over- and underconstrained geometric models. In: *Geometric Constraint Solving and Applications,* Brüderlin, B. and Roller, D. (Eds.), Springer, Berlin, pp. 107–127

Parametric (1998) *Pro/ENGINEER Modeling User's Guide, Version 19.* Parametric Technology Corporation, Waltham, MA, USA

Peeters, E.A.J. (1993) Design and implementation of an object-oriented, interactive animation system. In: *Technology of Object-Oriented Languages and Systems, TOOLS 12 & 9*, Mingins, C., Haebich, W., Potter, J. and Meyer, B. (Eds.), Prentice Hall, Englewood Cliffs, pp. 255–267

Peters, B.F. (1997) MicroStation Modeler: the design and implementation of an extensible solid modeling system. In: *Geometric Modeling: Theory and Practice – The State of the Art,* Strasser, W., Klein, R. and Rau, R. (Eds.), Springer, Berlin, pp. 361–378

Pratt, M.J. (1988) Synthesis of an optimal approach to form feature modelling. In: *Proceedings of the 1988 ASME Computers in Engineering Conference, August, San Francisco, CA, USA*, ASME, New York, Vol. 1, pp. 263–274

Pratt, M.J. (1984) Solid Modeling and the interface between design and manufacture. *IEEE Computer Graphics & Applications* **4**(7): 52–59

Raghothama, S. and Shapiro, V. (1998) Boundary representation deformation in parametric solid modeling. *ACM Transactions on Graphics* **17**(4): 259–286

Regli, B. and Pratt, M. (1996) What are feature interactions? In: *CD-ROM Proceedings of the 1996 ASME Computers in Engineering Conference, 19–22 August, Irvine, CA, USA,* McCarthy, J.M. (Ed.), ASME, New York

Rossignac, J.R. (1990) Issues on feature-based editing and interrogation of solid models. *Computers & Graphics* **14**(2): 149–172

Rossignac, J.R. and O'Connor, M.A (1990) A dimension-independent model for pointsets with internal structures and incomplete boundaries. In: *Geometric Modeling for Product Engineering*, Wozny, M.J., Turner, J.U. and Preiss, K. (Eds.), Elsevier Science Publishers, Amsterdam, pp. 145–180

Salomons, O.W., van Slooten, F., Jonker, H.G., van Houten, F.J.A.M. and Kals, H.J.J. (1994) Interactive feature definition. In: *Proceedings of IFIP WG5.3 International Conference on Feature Modelling and Recognition in Advanced CAD/CAM Systems*, Soenen, R. and Olling, G. (Eds.), Vol. 1, pp. 181–200

Salomons, O.W., Geelink, R., van Slooten, F., van Houten, F.J.A.M. and Kals, H.J.J. (1998) Definition of design and manufacturing form features. In: *Proceedings of the 31st CIRP International Seminar on Manufacturing Systems, 26–28 May, Berkeley, CA, USA*, pp. 7–39

Sannella, M. (1992) The SkyBlue constraint solver. Technical Report 92–07–02, University of Washington, WA, USA

SDRC (1998) *I-DEAS Master Series 6 User's Guide.* Structural Dynamics Research Corporation, Milford, OH, USA

da Silva, R., Wood, K.L. and Beaman, J.J. (1991) Interacting and inter-feature relationships in engineering design for manufacturing. *International Journal of Systems Automation: Research and Applications* (1): 263–286

Shah, J.J. (1991) Conceptual development of form features and feature modelers. *Research in Engineering Design* (2): 93–108

Shah, J.J. and Mäntylä, M. (1995) Parametric and feature-based CAD/CAM; concepts, techniques and applications. John Wiley & Sons, New York

Shah, J.J. and Rogers, M.T. (1988) Expert form feature modelling shell. *Computer-Aided Design* **20**(9): 515–524

Shah, J.J., Ali, A. and Rogers, M.T. (1994) Investigation of declarative feature modeling. In: In: *Proceedings of the 1994 ASME Computers in Engineering Conference, September, Minneapolis, MN, USA,* Ishii, K., Bannister, K. and Crawford, R. (Eds.), ASME, New York, Vol. 1, pp. 1–11

Shah, J.J., Rogers, M.T., Sreevalsan, P.C., Hsiao, D.W., Mathew, A., Bhatnagar, A., Liou, B.B. and Miller, D.W. (1990) The ASU Features Testbed: an overview. In: *Proceedings of the 1990 ASME Computers in Engineering Conference, August, Boston, MA, USA*, ASME, New York, Vol. 1, pp. 233–241

Spatial (1998) Acis 3D Modeling Kernel, Version 4.1. Spatial Technology Inc., Boulder, CO, USA

Suh, H. and Ahluwalia, R. (1995) Feature modification in incremental feature generation. *Computer-Aided Design* **27**(8): 627–635

Talwar, R. and Manoochehri, S. (1994) Algorithms to detect geometric interactions in a feature-based design system. In: *Proceedings of the 1994 ASME Design Engineering Technical Conferences, September, Minneapolis, MN, USA*, ASME, New York

Vieira, A.S. (1995) Consistency management in feature-based parametric design. In: *Proceedings of the 1995 ASME Design Engineering Technical Conferences, 17–21 September, Boston, MA, USA,* Gadh, R. (Ed.), ASME, New York, Vol. 2, pp. 977–987

Wirth, N. (1976) Algorithms + Data Structures = Programs. Prentice-Hall, Englewood Cliffs, NJ, USA

Index

D

F

G

H

I

K

L

M

Summary

Validity Maintenance in Semantic Feature Modeling

Rafael Bidarra

Computer-based techniques for supporting product development have evolved rapidly in the last two decades. Mainly driven by market and quality demands, concepts like rapid prototyping and concurrent engineering have been proposed. Such concepts require more, and more complex, information to be stored in so-called *product models*. For example, information about the shape of a product, traditionally stored and processed using geometric modeling techniques, is now required to be integrated with other types of product information (e.g. function, manufacturing and assembly information). Feature modeling constitutes an important milestone in this evolution.

Feature models can combine shape information and functional information, which makes them a versatile product representation suitable for the integration of many product life-cycle activities (e.g. design, manufacturing planning and assembly planning). For the success of this integration, a key role is played by a well-defined specification of the meaning, or *semantics*, of features, clearly associating the shape aspects to their desired functionality. Feature-based modeling systems should correctly interpret and maintain those associations in the feature model, throughout the whole modeling process. This is usually called *feature model validity maintenance*.

Current feature modeling systems, however, are still very much tied to methods and techniques of conventional geometric modeling systems. Among other drawbacks, they offer only restricted facilities for defining feature semantics, and often fail to preserve this semantics as the feature model evolves.

This thesis presents a new feature modeling approach –designated *semantic feature modeling*–, which overcomes the validity maintenance problems of current feature modeling systems.

In semantic feature modeling, feature specification is done declaratively in feature classes, using a variety of constraint types. A *feature class* is a structured description of all properties of a given feature type, and includes the validity conditions that all its feature instances should satisfy.

A two-level semantic feature model has been developed to represent a product. The first level –called the *Feature Dependency Graph*– consists of a set of interrelated feature and constraint instances: the entities on which all modeling operations are performed. The second level contains an evaluated geometric representation of the product in the so-called *Cellular Model*. Its most important property is that both the generation and the interpretation of the Cellular Model are independent of the chronological order of feature creation in the model. The two levels are integrated in the semantic feature model, which disposes of mechanisms for automatically maintaining the consistency between them. In addition, it supports a variety of queries at both levels, making it possible to perform, among other things, effective model validity maintenance.

Feature interactions, which arise from modeling operations such as the creation of a new feature or the modification of an existing feature, are among the main causes of feature semantics violations. Such phenomena are therefore thoroughly analyzed and classified in this thesis, and a variety of interaction detection algorithms is also presented.

The validity maintenance scheme presented here basically monitors each modeling operation, in order to assess the conformity of all features in the semantic feature model with their validity criteria. This is achieved by maintaining all constraints, using various constraint solving techniques.

Each invalid situation detected is analyzed by a validity recovery mechanism, which gives the user explanations and context-sensitive hints to overcome the situation. The user gets thus valuable assistance in creating valid models only, containing features with well-defined semantics only.

The semantic feature modeling approach has been implemented in the prototype modeling system SPIFF. This system provides interactive facilities for the specification of feature classes, and modeling facilities for the creation and manipulation of semantic feature models. The Cellular Model has been implemented using the Cellular Topology husk of the Acis Geometric Modeling kernel.

Samenvatting

Validiteitshandhaving in Semantisch Feature Modelleren

Rafael Bidarra

Computer-gebaseerde technieken voor ondersteuning van product-ontwikkeling hebben de afgelopen twee decennia een snelle evolutie doorgemaakt. Vooral door marktvraag en kwaliteitseisen zijn benaderingen als *rapid prototyping* en *concurrent engineering* ontstaan. Dergelijke benaderingen vereisen meer en complexere informatie, die in zogenoemde *productmodellen* moet worden opgeslagen. Zo wordt vandaag de dag informatie over de vorm van een product, van oudsher opgeslagen en verwerkt met geometrische modelleertechnieken, geïntegreerd met andere soorten productinformatie (o.a. functionele informatie, productieinformatie en assemblageinformatie). Feature modelleren vormt een belangrijke mijlpaal in deze ontwikkeling.

Feature modellen zijn in staat vorminformatie en functionele informatie te combineren, zodat een veelzijdige representatie van een product ontstaat die geschikt is voor de integratie van veel activiteiten in de levenscyclus van het product (b.v. ontwerp, productieplanning en assemblageplanning). Een succesvolle integratie is echter sterk afhankelijk van een goed afgebakende specificatie van de betekenis van de features, oftewel hun *semantiek*. Deze koppelt op een duidelijke wijze vormaspecten aan hun gewenste functionaliteit. Feature-gebaseerde modelleersystemen dienen te zorgen voor de juiste interpretatie en het handhaven van deze koppelingen in het feature model gedurende het

hele modelleerproces. Dit wordt gewoonlijk *feature model validiteits-handhaving* genoemd.

De huidige feature modelleersystemen zijn echter nog steeds sterk gebaseerd op technieken van conventionele geometrische modelleer-systemen. Nadelen hiervan zijn o.a. dat zij slechts beperkte faciliteiten bieden om de semantiek van features te definiëren en vaak tekort-schieten om deze te handhaven tijdens het modelleren.

Dit proefschrift stelt een nieuwe benadering van feature modelleren voor –aangeduid met *semantisch feature modelleren*– waarbij de problemen van validiteitshandhaving met de huidige feature model-leersystemen worden overwonnen.

Bij semantisch feature modelleren vindt feature specificatie op declaratieve wijze plaats in feature klassen, waarbij van diverse typen constraints gebruik wordt gemaakt. Een feature klasse is een gestruc-tureerde beschrijving van alle eigenschappen van een gegeven feature *type*, en bevat de validiteitsvoorwaarden waaraan al zijn feature instan-ties moeten voldoen.

Een semantisch feature model met twee niveaus is ontwikkeld om een product te representeren. Het eerste niveau –de *Feature Dependency Graph* genoemd– bestaat uit een verzameling van onderling gerelateerde feature en constraint instanties, de entiteiten waarop alle model-leerhandelingen worden uitgevoerd. Het tweede niveau bevat een geëvalueerde geometrische representatie van het product in het zogenoemde *Cellulaire Model*. De belangrijkste eigenschap hiervan is dat zowel het genereren als de interpretatie van het Cellulaire Model onafhankelijk zijn van de chronologische volgorde van het toevoegen van de features aan het model. De twee niveaus worden geïntegreerd in het semantische feature model, dat over mechanismen beschikt voor het automatisch handhaven van de consistentie tussen beide niveaus. Verder biedt dit model allerlei mogelijkheden om beide niveaus te raadplegen, hetgeen o.a. effectieve handhaving van de modelvaliditeit mogelijk maakt.

Feature interacties, die het gevolg kunnen zijn van model-leeroperaties zoals het creëren van een nieuwe feature of het wijzigen van een bestaande, vormen een van de belangrijkste oorzaken van schendingen van de semantiek van features. Zulke verschijnselen worden daarom in dit proefschrift grondig geanalyseerd en geclassificeerd, en een aantal algoritmen om interacties te detecteren wordt behandeld.

Het hier gepresenteerde schema voor het handhaven van de validiteit controleert elke modelleeroperatie, om te bepalen of alle features in het

model nog steeds overeenkomen met hun validiteitsvoorwaarden. Hiervoor worden alle constraints beschouwd m.b.v. verschillende technieken voor het oplossen van constraints.

Elke ongeldige situatie wordt door een mechanisme voor het herstellen van de validiteit geanalyseerd. Dit mechanisme is in staat toelichtingen en context-afhankelijke tips te verschaffen om zo'n situatie te overwinnen. De gebruiker wordt dus effectief ondersteund om alleen geldige modellen te creëren, die slechts features met goedgedefinieerde semantiek bevatten.

Semantisch feature modelleren is in het experimentele modelleersysteem SPIFF geïmplementeerd. Dit systeem ondersteunt interactieve specificatie van feature klassen en biedt ruime modelleerfaciliteiten om een semantisch feature model te creëren en te wijzigen. Het Cellulaire Model werd geïmplementeerd m.b.v. de Cellular Topology husk van de Acis Geometric Modeling kernel.

Curriculum Vitæ

António Rafael Emiliano Bidarra de Almeida was born on June 25th, 1964, in Lisbon, Portugal.

In 1987 he graduated in Electronics Engineering at the University of Coimbra, Portugal, with a specialization in Computer Science.

From 1987 until 1989 he worked at the Computer Science Department of this university, as a researcher on an artificial intelligence production scheduling project, in cooperation with Texas Instruments Inc., Portugal.

In 1989 he switched to the Mathematics Department of the same university as a research assistant, and worked in various projects on geometric and solid modeling. Since then he has focused his research work on feature-based modeling.

In 1995 he joined the Faculty of Information Technology and Systems of Delft University of Technology, The Netherlands, as a PhD student. At its Computer Graphics and CAD/CAM group, and in cooperation with the Department of Mechanical Engineering of the New University of Lisbon, Portugal, he carried out the research project on semantic feature modeling, which resulted in this thesis.

Since January 1999 he holds a post-doc position at the same research group, in Delft, where he can be reached at R.Bidarra@cs.tudelft.nl (http://wwwcg.twi.tudelft.nl/~rafa).