ELSEVIER

Computer-Aided Design 32 (2000) 201–225

COMPUTER-AIDED DESIGN

www.elsevier.com/locate/cad

Semantic feature modelling

R. Bidarra, W.F. Bronsvoort*

Faculty of Information Technology and Systems, Delft University of Technology, Zuidplantsoen 4, NL-2628 BZ Delft, The Netherlands

Received 13 June 1998; received in revised form 13 September 1999; accepted 15 September 1999

Abstract

Almost all current feature modelling systems are parametric, history-based modelling systems. These systems suffer from a number of shortcomings with regard to the modelling process. In particular, they lack a complete specification of feature semantics, and therefore fail to maintain the meaning of the features during modelling. Also, modelling operations sometimes are hampered by the model history, and occasionally even have ill-defined semantics.

In the semantic feature modelling approach presented here, the semantics of features is well defined and maintained during all modelling operations. The result of these operations is independent of the order of feature creation, and is well defined. The specification of feature classes, the structure and functionality of the feature model, in particular the Cellular Model, and the model validity maintenance scheme are described. The advantages and disadvantages of the approach, compared to current feature modelling approaches, are pointed out.

Keywords: Feature modelling; History-based modelling; Feature semantics; Declarative modelling; Validity maintenance

1. Introduction

Feature modelling is increasingly being used for modelling products. One of its main advantages over conventional geometric modelling 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 functionally significant for some product life-cycle phase*. An essential aspect of a feature is that it has a well-defined meaning, or *semantics*, for a particular life-cycle activity.

Two important aspects of the above definition are not well covered by most current feature modelling systems. First, feature semantics is poorly defined, limiting the capability of capturing design intent in the model. Second, feature semantics is poorly maintained, permitting previous design intent to be overruled. Such systems are said to lack *validity maintenance* facilities.

Current feature modelling systems do provide the user with "engineering rich" dialogs aimed at the creation and manipulation of feature instances. In some systems, however, these "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 modellers. Most other feature modelling systems, although they do store information about features in the product model, fail to adequately maintain the meaning of features throughout the modelling process. For example, a modelling operation on one feature may significantly affect the semantics of other features, without the user even being notified by the system, let alone assisted in overcoming this undesirable situation. Assessing the extent to which feature semantics is kept in a model is therefore a crucial issue in feature model validity maintenance.

Fig. 1 illustrates this idea. 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 Fig. 1a. If the width of the step is increased, the rounded pocket overlaps with the two blind holes, "removing" their circular bottom faces from the model boundary (see Fig. 1b). 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.

* Corresponding author. Tel.: +31-15-278-2533; fax: +31-15-278-7141.
E-mail address: Bronsvoort@cs.tudelft.nl (W.F Bronsvoort).

0010-4485/00/$ - see front matter
PII: S0010-4485(99)00090-1

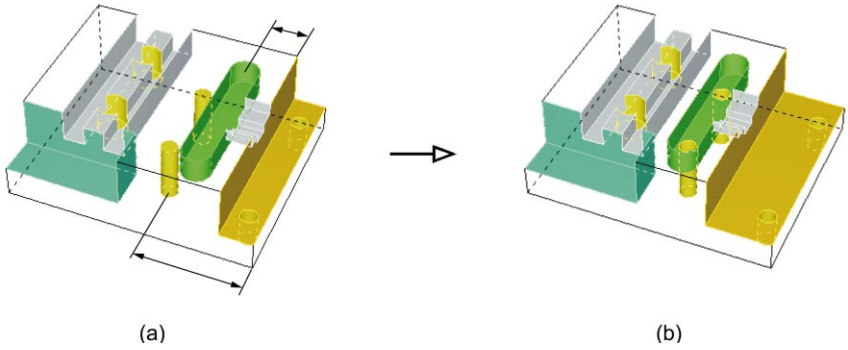


Fig. 1. Changing feature semantics with a modelling operation.

In addition to lacking feature model validity maintenance, current feature modelling systems also present unexpected results after some modelling operations that are therefore said to have *ill-defined semantics*. One of the reasons for this shortcoming is that these systems are too tied to methods and techniques of conventional geometric modelling, e.g. they strongly rely on a history-based notion of the modelling process.

Raising the level of assistance provided to the user in maintaining and recovering model validity is essential to bring feature technology to maturity. In this article, we present an alternative way of feature modelling, in which the problems pointed out for current feature modelling systems are overcome. In particular, the semantics of each feature is clearly specified and maintained during the whole modelling process, and the semantics of each modelling operation is well defined. This new approach is therefore called *semantic feature modelling*.

In Section 2, current approaches to feature modelling are surveyed, and their shortcomings identified. The basic idea of semantic feature modelling is presented in Section 3, and elaborated in subsequent sections: the specification of feature classes in Section 4, the structure and functionality of the feature model in Section 5, and the feature model validity maintenance scheme in Section 6. In Section 7, a few examples taken from a modelling session illustrate the usefulness of this approach. Finally, in Section 8, current feature modelling approaches and semantic feature modelling are compared on their merits.

2. Current approaches to feature modelling

Almost all current feature modelling systems are parametric, history-based modelling 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 Autodesk Mechanical Desktop [1], Pro/Engineer [27], MicroStation Modeller [28] and I-DEAS Master Series [34].

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

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 created by sequentially re-executing the operations in the modified history. With this scheme, variants of a feature model can easily be

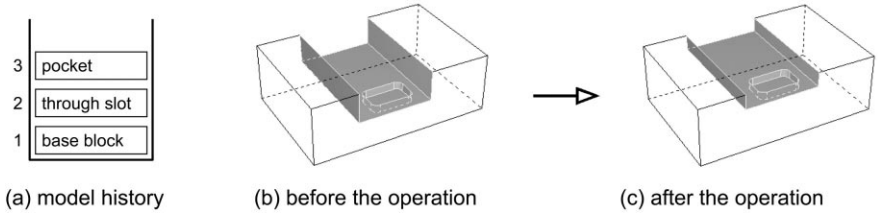


Fig. 2. Example of re-executing the model history.

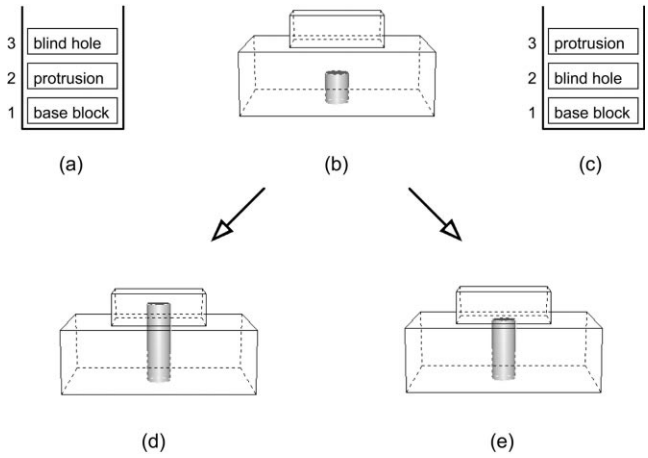


Fig. 3. Set operations problem in history-based model re-evaluation.

created. A simple example of this is given in Fig. 2. The model has a base block, a through slot and, attached to the latter, a pocket (see Fig. 2b). If the depth of the through slot is decreased, the model history in Fig. 2a is re-executed, yielding the model in Fig. 2c.

Most current feature modelling systems have, at least, six major shortcomings, which 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 evolution of the entities in the evaluated boundary model. The sixth problem is mainly due to the use of a manifold boundary representation.

2.1. Computational cost

The first shortcoming is that re-executing the whole model history after modifying or deleting a feature has a high computational cost, roughly proportional to the model history size. Several methods have been devised to improve this, e.g. storing all intermediate boundary representations between history steps. 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, roughly 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 re-executed history steps almost always includes more features than those actually modified by the operation in question.

2.2. Non-associative set operations

The second shortcoming is that history-based re-evaluation of the model does not guarantee that the evaluated model matches the specified parameters of features that overlap. This is illustrated in the model of Fig. 3b, 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 Fig. 3a or that in Fig. 3c. 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 the expected depth (Fig. 3d), whereas, in case (c), the blind hole will be "truncated" by the protrusion, its depth becoming equal to the block height (Fig. 3e). This problem is caused 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 the last operation prevents the blind hole from exhibiting its nominal depth in the model of Fig. 3e.

2.3. Entity references in the model history

The third shortcoming is that history-based re-evaluation of the model 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 Fig. 4. The model consists of a block, a through hole and a protrusion (see Fig. 4b). The history of this model could be either that in Fig. 4a or that in Fig. 4c. In the first case, re-attachment of the through hole to the top of the protrusion and the bottom of the block can be achieved by modifying the corresponding attach reference of the through hole (see Fig. 4d), and re-executing the history. However, if this reference modification would be made in the model history of Fig. 4c, re-evaluation of the model would not be possible, because the through hole creation cannot be re-executed with a reference to a face

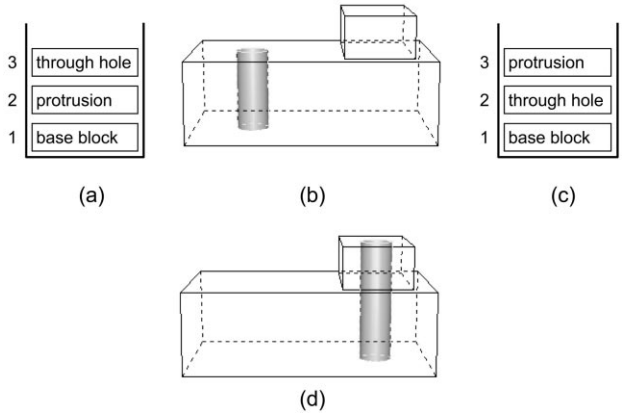


Fig. 4. Entity reference problem in history-based model re-evaluation.

(the top of the protrusion) that will be created in the model at a later stage of its history.

Evaluation of the 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 modelling 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.

2.4. Type of constraints

The fourth shortcoming of current feature modelling 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 Fig. 5a, 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 Fig. 5b). However, such constraints are, in most systems, unidirectional. In the example of Fig. 5, the blind hole depth is dependent on the slot depth, but not the other way round. This implies that if the blind hole depth is modified in a subsequent modelling operation, the depth of the slot is not adapted accordingly, and is no longer equal to the depth of the blind hole. This inability to cope with bi-directional constraints makes the dimensioning of the model undesirably rigid.

2.5. Persistent naming problem

The fifth shortcoming is that in history-based modelling systems the semantics of modelling operations is not always well defined. The main cause of this is the so-called *persistent naming problem*. Each modelling operation uses references to topologic entities in the boundary representation of the current model, which is the combined result of all previous modelling 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, also when the operation is re-executed. However, a general property of boundary representations is that topologic entities may be split, merged or deleted because of modelling operations. Persistent naming is the process of identifying and tracking topologic entities when a geometric model is modified [21]. Although some schemes for persistent naming have been implemented [12,21,23], there are some fundamental problems related to this issue, of which two typical examples will now be given. See Ref. [29] for a formal approach to these problems.

The first example has been taken from Chen and Hoffmann [13] (see Fig. 6). The model consists of a block to which subsequently a through slot (Fig. 6a) and a chamfer (Fig. 6b) have been added. The next modelling 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

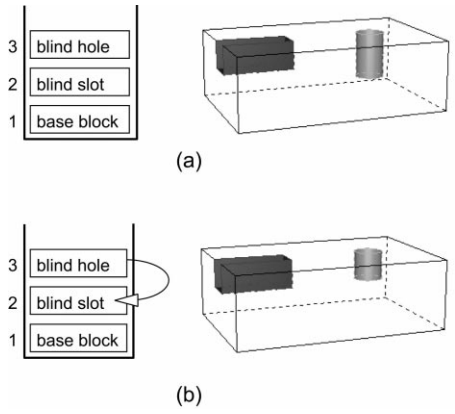


Fig. 5. Dimensioning of a model with unidirectional constraints.

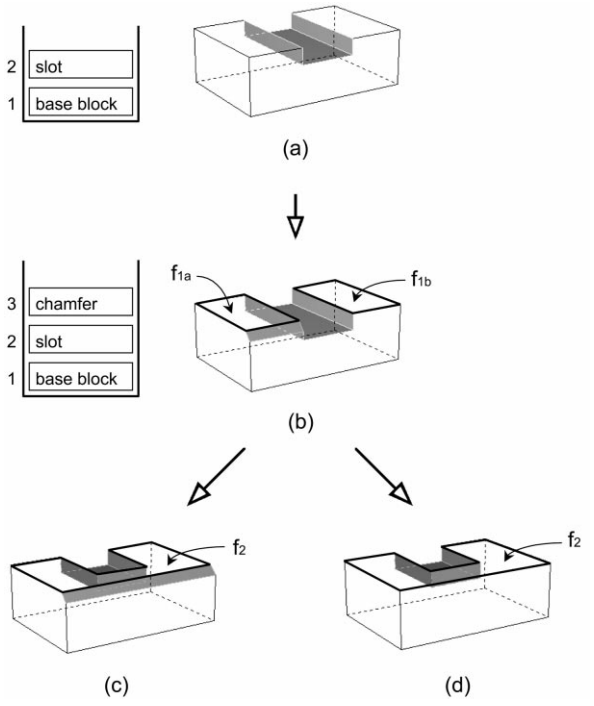


Fig. 6. Semantic problem related to entity naming: splitting and merging of faces.

be extended along the whole edge (see Fig. 6c) or be completely deleted (see Fig. 6d). 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 [23]. 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 Fig. 7a). 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_{1a} and f_{1b} (see Fig. 7b). The next modelling 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 (see Fig. 7c) or intrude into the block (see Fig. 7d).

In both examples, the model resulting from a sequence of modelling 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 modelling operations, it is *ambiguous*, in the sense that it is definitely not always as expected by the user of the modelling system. Stated differently, the semantics of some operations is not well defined.

2.6. Maintenance of feature semantics

The sixth, and in a way most serious, shortcoming of current feature modelling systems is that they do not maintain feature semantics. Each feature type specifies its own

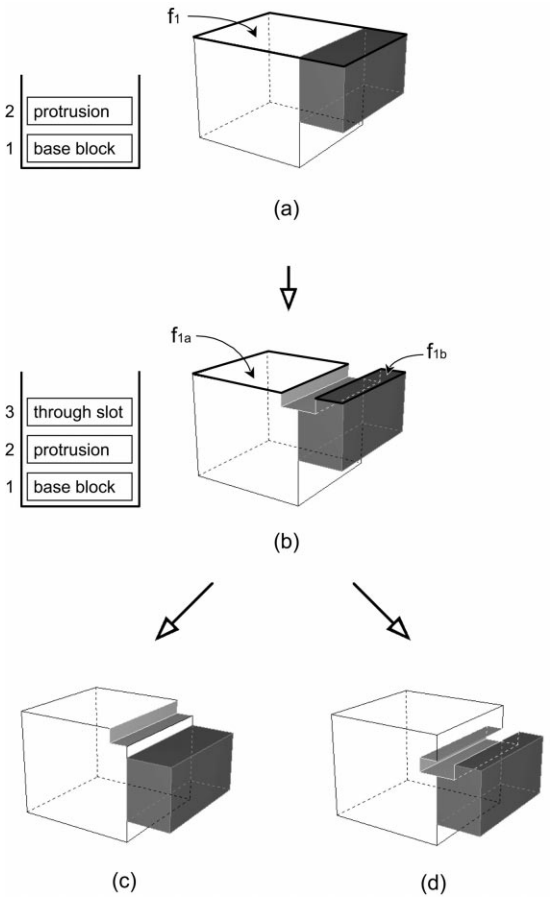


Fig. 7. Semantic problem related to entity naming: merging and splitting of faces.

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 analyse 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 be accessed in this analysis. Consequently, 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 this. Systematically analysing 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.

Rossignac [32] provides a good insight into some high-level feature validity issues, alerting for inconsistencies that may arise from a naive interpretation of usual editing commands on feature models. Some more recent research work has focused on the validation of features, both on various validity specification issues [11,18] and on validity maintenance [15,25]. One of the main conclusions of that

research is that a declarative scheme is preferable over the conventional procedural modelling 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 modelling system, 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 Vieira [37] and Dohmen [14], are limited to the detection of a number of predefined invalid situations, for which the only solution offered by the modelling system is the rejection of the concerning modelling operation. This rigid scheme hinders the modelling process, yet permits many unanticipated inconsistencies in the model.

A first example of problems with changing the semantics of features has already been given in the previous section, Fig. 1, where two blind holes were turned into through holes. Another example is that of Fig. 7, in which none of the two results contains a real through slot, with two sides and a fully open top, as was specified. These problems are due to the inability to store in a manifold boundary representation all feature information, e.g. closure faces of subtractive features. This in turn excludes the possibility of analysing the topology of the boundary of those features, which is essential to detect and prevent modifications in feature semantics. Such validity violations, due to modelling operations that cause overlapping features to affect each other's semantics, are usually called *feature interactions*. Feature interaction phenomena are regarded as a major problem affecting feature semantics [31], but are not dealt with in current feature modelling systems.

In fact, current feature modelling systems offer more a geometric modelling approach, to create a boundary representation, than a genuine feature modelling approach. One of the basic ideas of feature modelling 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 design intent, is significantly modified due to a subsequent modelling operation. In other words, arbitrarily modifying the semantics of a feature should be disallowed if one wants to make feature modelling really more powerful than geometric modelling.

Summarising, current feature modelling systems have computational cost problems, suffer from dimensioning and modelling limitations due to their strong dependency on the chronological order of feature creation and to the use of unidirectional constraints, occasionally suffer from ill-defined semantics of modelling operations, and do not adequately maintain the semantics of features.

In this article, we propose a new feature modelling approach: *semantic feature modelling* [2]. This approach will be outlined in the next section, and elaborated in the subsequent sections, emphasising how it overcomes the problems pointed out for current feature modelling approaches.

3. What is semantic feature modelling?

Semantic feature modelling is a declarative feature modelling approach. This means that, in contrast to many current 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. The main advantage of declarative modelling is the freedom in the type of constraints that can be specified, and therefore in the way a model can be edited and maintained.

In the semantic feature modelling 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.

Most current systems have a rudimentary form of validity conditions, but our 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's cylindrical shape should remain open, or, stated differently, 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 modelling process; without them, features can never be more than high-level geometric modelling 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 modelling session. Feature class specification is elaborated in Section 4.

Another characteristic of semantic feature modelling is that the whole modelling process is uniformly carried out in terms of features and their entities (e.g. faces and parameters), and of constraints among these. So all modelling actions performed by the user are effectively *feature-based*, and the same applies to all output, both graphical and textual, generated by the modelling system. An advantage of this is that a feature and, in particular, its faces and their names are persistent. These remain valid, and therefore also all references to them, as long as that feature instance remains in the model. This is in contrast to history-based modelling approaches, in which references to entities of the evaluated model are kept in the model history, with

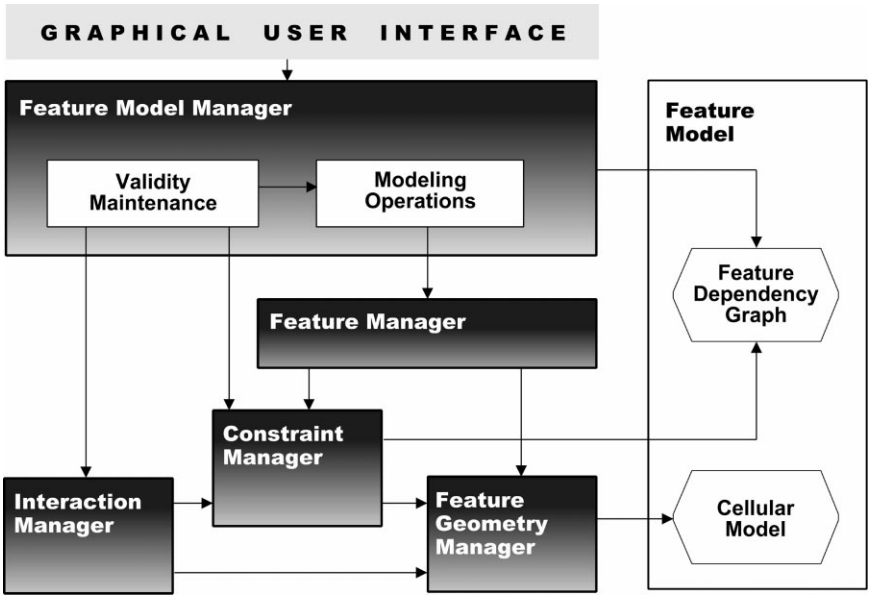


Fig. 8. Architecture of the SPIFF Feature Modeller.

the drawbacks identified in the previous section; unlike boundary faces in such systems, feature faces are never split, merged or deleted, even though their geometric representation may be.

Probably the most important characteristic of the semantic feature modelling approach is that the semantics of all features is effectively maintained throughout model evolution, for all modelling operations. Some essential aspects of feature semantics, e.g. the through hole clearance described above, cannot be maintained without an evaluated geometric model, also 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 modelling just mentioned lead to a two-level structure in the semantic feature model, clearly distinguishing *modelling entities* from *entities in the evaluated geometric model*. The former, i.e. the entities on which all modelling 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 modelling operations performed on the first level. The semantic feature model, and mechanisms for maintaining the consistency between both levels, are elaborated in Section 5.

Maintaining the feature model throughout the modelling process requires not only managing all its constraints, but also monitoring each modelling operation in order to assess the conformity of each feature in the model with its validity criteria. For example, most changes in the meaning of features are due to modelling operations that cause overlapping features to affect each other's semantics, so-called feature interactions, as illustrated in the examples of Figs. 1 and 7. Managing feature interaction phenomena plays an essential role in the validity maintenance scheme of the semantic feature modelling approach, so that all relevant interaction situations can be detected, reported and handled in an appropriate way. Only 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 modelling 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 modelling process. Feature model validity maintenance is elaborated in Section 6.

The semantic feature modelling approach has been implemented in the SPIFF system [9], a prototype multiple-view feature modelling system developed at Delft University of Technology.

SPIFF consists of two main functional subsystems: the Feature Library Manager and the Feature Modeller. The *Feature Library Manager* provides interactive facilities for specification of feature classes and for their organisation in application-specific feature libraries. These class specifications can be loaded into the Feature Modeller at runtime. The functionality and architecture of the Feature Library Manager have been described by Bidarra et al. [6], and are summarised in Section 4.

The *Feature Modeller* provides modelling facilities for

creation and manipulation of feature models, according to the architecture depicted in Fig. 8. Several system modules have been described elsewhere [5,15,19], and will be only briefly summarised 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 maintains the *Feature Dependency Graph*, a high-level representation of the structure of the product (see Sections 5.2 and 5.4). Further, the Feature Model Manager is responsible for controlling all modelling operations, and for maintaining model validity (see Section 6).

The *Feature Manager* supervises the model processing tasks of each modelling 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 Cellular Model, and takes care of updating it as required by each modelling operation (see Sections 5.3 and 5.5). The *Interaction Manager* is responsible for the analysis of the Cellular Model, in order to detect any disallowed feature interactions possibly resulting from a modelling operation (see Section 6.1).

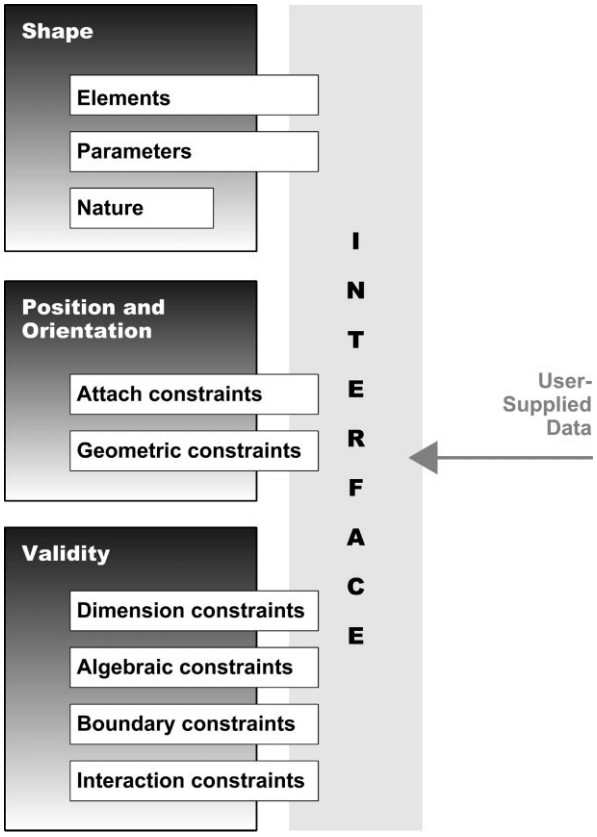


Fig. 9. Feature class structure.

4. Specification of feature semantics

Feature class specification involves specification of its shape, its validity conditions, and its interface to the feature model, according to the structure depicted in Fig. 9. For all aspects, constraints are used. These *feature constraints* are members of the feature class, and are therefore instantiated automatically with each new feature instance.

The basis of a feature class is a parameterised shape. For a simple feature, this is a *basic shape*, e.g. a cylinder for a hole. A basic shape encapsulates a set of geometric constraints that relate its parameters to the corresponding shape faces. For a compound feature, the shape is a combination of several, possibly overlapping, basic shapes, e.g. two cylinders for a stepped hole.

The geometry of a feature, 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 labelled with its own generic name, to be used in modelling operations. For example, a cylinder shape has a *top*, a *bottom* and a *side* face.

A feature class also associates 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).

The specification of validity conditions in a feature class can be classified into three categories: geometric, topologic and functional.

One way of constraining the geometry of a feature class is by specifying the set of values allowed for a shape parameter. We use *dimension constraints* applied on shape parameters. For instance, the radius parameter of a 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*.

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 modelled 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 *boundary constraints* on each shape face. Boundary constraints, first proposed by Bidarra and Teixeira [8] under the name *semantic constraints*, specify which topological variants of a feature instance are allowed, by stating

Table 1
Classification of feature interactions

Interaction type	Description
Splitting	Splits the boundary of a feature into two (or more) disconnected subsets
Disconnection	Causes the volume of an additive feature (or part of it) to become disconnected from the model
Boundary clearance	Causes (partial) obstruction of a closure face of a subtractive feature
Volume clearance	Causes partial obstruction of the volume of a subtractive feature
Closure	Causes some subtractive feature volume(s) to become a closed void inside the model
Absorption	Causes a feature to cease completely its contribution to the model shape
Geometric	Causes a mismatch between a nominal parameter value and the actual feature geometry
Transmutation	Causes a feature instance to exhibit the shape imprint characteristic of another feature class
Topologic	Corresponds to the violation of a boundary constraint in a given feature

the extent to which its feature faces should be on the model boundary. Boundary 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. Further, both types of boundary constraints are parameterised, 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.

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 some class should somehow contribute to the shape of the part model. Just like boundary constraints, such functional requirements can be violated by *feature interactions* caused during incremental editing of the model. Feature interactions are modifications of the shape aspects of a feature that affect its functional meaning. An example of this is the *transmutation* interaction of the blind holes into through holes in Fig. 1. A comprehensive classification of feature interactions can be found in Ref. [2]. For completeness, it is briefly summarised in Table 1. We use *interaction constraints* in a feature class in order to indicate that a particular interaction type is not allowed for its instances [5]. Topologic interaction, corresponding to the violation of a boundary constraint, is, by definition always disallowed.

Feature constraints and parameters may require external data to be provided at feature instantiation stage—the so-called *user-supplied data*. 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 modelling system and how the user will be able to interact with them. 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 Fig. 9.

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. For example, the top and bottom faces of a through hole are used to attach it to, say, the top and bottom faces of a block, respectively.

Geometric constraints position and orient a feature relative to faces of other features 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 extra 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.

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.

A detailed description of feature class specification following the semantic feature modelling approach can be found in Ref. [6].

5. The semantic feature model

This section describes the *semantic feature model*, on which the semantic feature modelling approach is based. First, the important notion of *dependency* between model entities is introduced (Section 5.1). Next, the two levels integrated in the feature model—the Feature Dependency Graph and the Cellular Model—are elaborated (Sections 5.2 and 5.3), and mechanisms for model maintenance are presented for both levels (Sections 5.4 and 5.5). Finally, a history-independent scheme for the interpretation of the Cellular Model is presented (Section 5.6).

5.1. The dependency relation

Many researchers in feature modelling have pointed out the convenience of keeping track of the model structure in terms of the relations among its features, in addition to a low-level, evaluated geometric model. A variety of structures has been proposed, expressing attachment, adjacency,

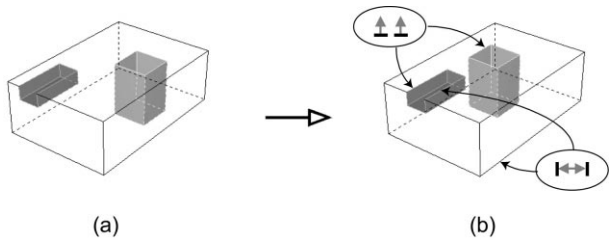


Fig. 10. Relative repositioning of features by means of model constraints.

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 [10,16,17,22,24,35]. The *dependency relation* defined in this section has such a graph representation. It is clearly defined, has rich semantics, and has direct application in feature model maintenance.

5.1.1. Dependencies among features

As described in Section 4, instantiation of a new feature requires the user to supply a set of parameter values, aimed at initialising all its constraints and parameters. Some of these values consist of references to faces of other features, 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 3, 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. So, for example, if a blind hole is attached to the bottom face of a slot, it will be displaced when the depth parameter of the slot is modified. Also, the blind hole attachment has to be readjusted, or the blind hole itself removed, when the slot feature is removed from the model. A dependency between two features is unidirectional: one can always distinguish the feature that is determined from the feature that determines. In this sense, 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.

5.1.2. Dependencies between constraints and features

In feature modelling, it is very convenient to be able to define, besides the constraints in feature classes, constraints on or 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 4, and are called *model constraints*.

A simple example of the use of model constraints is given in Fig. 10. The slot and the passage features, which were independently positioned relative to the block side faces (see Fig. 10a), are repositioned and aligned by 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 therefore also of the passage, to the block right face (see Fig. 10b).

This example illustrates three important properties of model constraints:

1. Unlike the feature constraints used throughout Section 4, they 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 Fig. 10b, 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 in Section 2.4).
3. They mostly overrule some feature constraints previously specified. In the example of Fig. 10b, the two model constraints prevail over the original positioning constraints of the two features relative to the block side faces. If this would not be done, an over-constrained 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 Fig. 10b 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 following sections. It is a dynamic relation among modelling entities, and can therefore evolve as these entities are modified, in contrast to the static chronological feature creation order used in most history-based feature modelling systems.

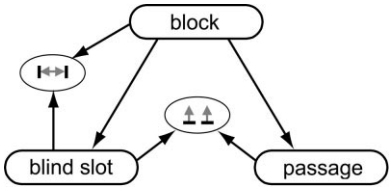


Fig. 11. Feature Dependency Graph of the model in Fig. 10b.

5.2. 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 a dependency relation, and is oriented towards the dependent feature or model constraint. As an example, Fig. 11 depicts the Feature Dependency Graph of the model in Fig. 10b.

The Feature Dependency Graph provides 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 modelling system and the model takes place in terms of the features and model constraints in the Feature Dependency Graph. In addition, all modelling 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 dependencies. Further, 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.

5.3. 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 Ref. [7], which also contains a survey on other research proposals for the geometric representation of feature models.

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 therefore represented in the Cellular Model by a connected subset of cells.

Further, 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 in either of them. Consequently, 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.

As described in Section 4, the boundary of a feature's shape extent is decomposed into functionally meaningful subsets, the shape faces, each one labelled with its own generic name. Each shape face is represented by a connected set of cell faces. In order to be able to search and analyse features and their faces in the Cellular Model, each cell has an attribute—called *owner list*—indicating which shape extents it belongs to (see Fig. 12). Similarly, each cell face has also an owner list, indicating to which shape faces it belongs.

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 Section 5.6. For example, in the model of Fig. 12, cells 1, 2 and 7 have additive nature (i.e. the nature of the block, the cylindrical protrusion or the rectangular protrusion, respectively), whereas cells 3, 5 and 6 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.

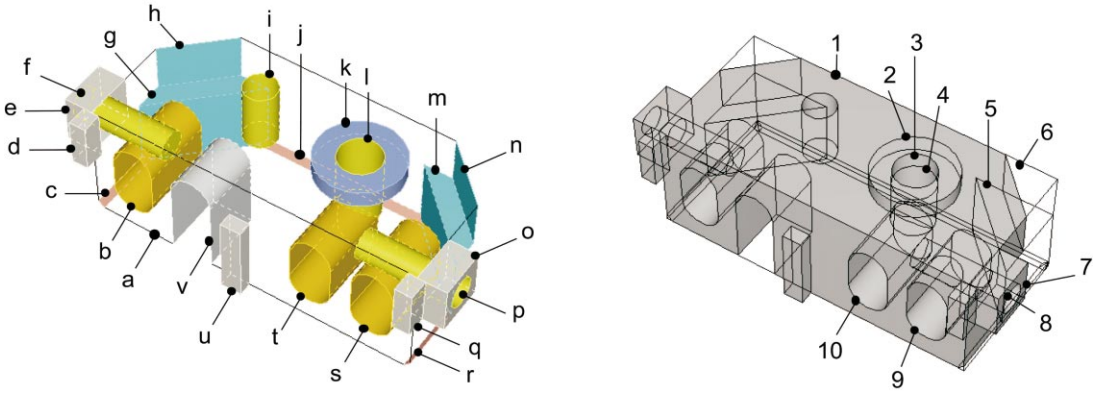
The Cellular Model, including its attribute mechanism to maintain and propagate the owner lists of cells and cell faces, was implemented using the Cellular Topology husk of the Acis Geometric Modeller [36].

5.4. Feature Dependency Graph maintenance

Feature model maintenance is the process of updating the feature model, according to the requirements of each modelling operation. It is performed at both levels of the semantic feature model: first, the Feature Dependency Graph is modified as described in this section; next, the Cellular Model is updated accordingly as described in the next two sections.

Modelling 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 initialisation



(a) part (b) Cellular Model

a block
b pocketLeft
c chamferLeft
d ribLeft
e protrusionLeft
f blindHoleLeft
g cornerBotLeft
h cornerTopLeft
i blindHoleBack
j chamferBack
k cylindricalProtrusion
l blindHoleTop
m cornerBotRight
n cornerTopRight
o protrusionRight
p blindHoleRight
q ribRight
r chamferRight
s pocketRight1
t pocketRight2
u ribFront
v blindSlot

(c) features

1 <block>
2 <cylindricalProtrusion>
3 <cylindricalProtrusion, blindHoleTop>
4 <block, blindHoleTop>
5 <block, cornerBotRight>
6 <block, cornerTopRight>
7 <protrusionRight>
8 <protrusionRight, blindHoleRight>
9 <block, pocketRight1>
10 <block, pocketRight2>
.
.
.

(d) some cell owner lists

Fig. 12. Cell owner lists in the Cellular Model.

parameter values for the new feature. Together with this, all constraint members specified in its class are also instantiated, and initialised with the corresponding user-supplied parameter values (e.g. distance parameters and external feature faces for attach and positioning constraints, see Section 4).

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 Fig. 10b and property (3), in Section 5.1, for an example).

In Section 6 the generic scheme for modelling operations will be analysed in detail, distinguishing in them a number of steps (Fig. 19). 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 modelling operations (except for feature removal operations) is the internal geometric and algebraic constraint solving process. When this process has been 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 modelling operation.

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

Adding a new feature instance to the model. The new feature instance is added to the Feature Dependency Graph, and all its dependencies are stored, according to the user-supplied interface parameter 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 removal operation is, however, only allowed if it has no dependent entities (features or constraints) in the Feature Dependency Graph; otherwise, the user is given

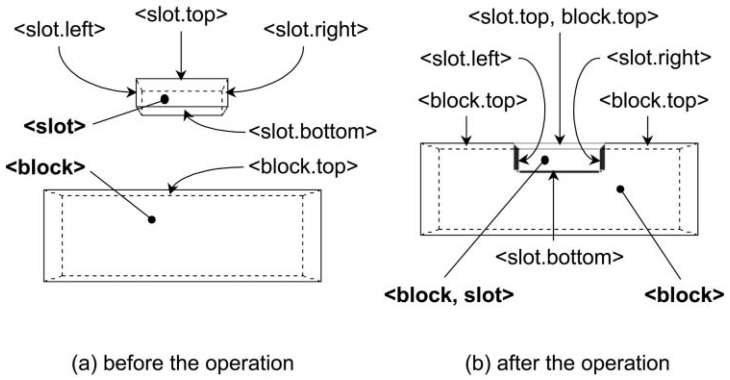


Fig. 13. Propagation of owner data in a cellular union operation.

the possibility of modifying these in order to eliminate their dependencies (e.g. by changing their attachments, see Section 6.2).

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

5.5. Cellular Model maintenance

The next step of a modelling 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 be able to check boundary and interaction constraints, which are concerned with the concrete geometry and topology exhibited in the Cellular Model by the features involved in the operation (see Section 6 for more on this *feature interaction detection* process).

For each modelling 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 Section 5.6.

These two phases make it possible to satisfy the following goals, essential to overcome the first three shortcomings pointed out in Section 2:

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.

5.5.1. Incremental Cellular Model evaluation

In contrast to history-based systems, which use two non-associative set operations (union and difference) to evaluate the geometric model (see example in Section 2.2), in the semantic feature modelling 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

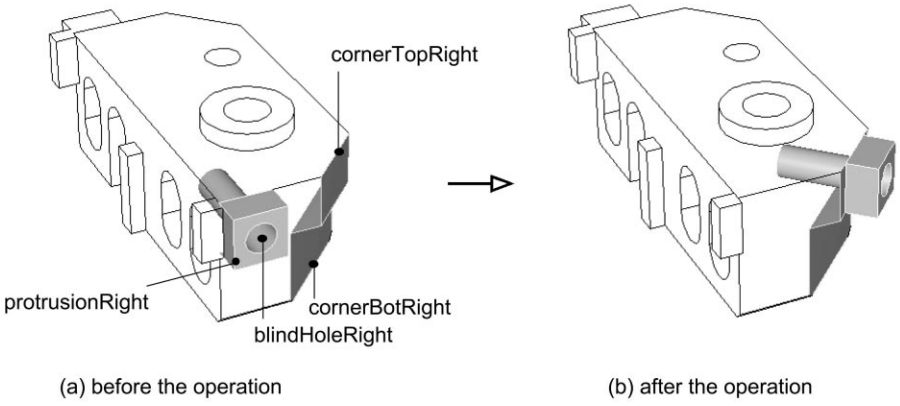


Fig. 14. Incremental evaluation of the model after a feature re-attachment.

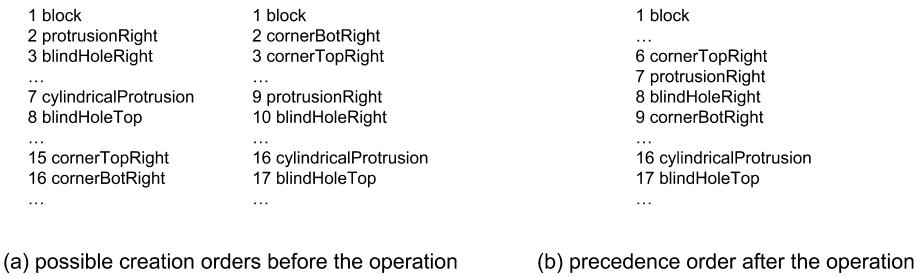


Fig. 15. Feature creation and precedence order examples for the models of Fig. 14.

processed is irrelevant for the final Cellular Model obtained. By these non-regular cellular operations, between (the single cell representing) a shape extent and the other cells generated so far, the cellular decomposition described in Section 5.3 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 Fig. 13, where a rectangular slot is inserted into a Cellular Model consisting of a single block. Before the cellular union, the owner lists of both cells are as shown in Fig. 13a (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 shown in Fig. 13b. The owner lists of the cell faces in Fig. 13a are also propagated, when these faces are split, as shown in Fig. 13b.

Re-evaluation of the Cellular Model after each modelling operation makes extensive use of the ability to process the cellular topology. A detailed description of Cellular Model processing algorithms can be found in Ref. [7]. According to the particular feature operation, these can be summarised as follows:

Adding a new feature instance to the model. The shape extent of the new feature is added to the Cellular Model. For this, the non-regular 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.

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 of the model in Fig. 12 is given in Fig. 14, where the attachment of the protrusionRight is modified, from the block to the cornerTopRight face. The blindHoleRight is also displaced, due to its attachment to the protrusionRight, whereas all other features maintain all their parameters and their position. The scope of modified features is easily obtained by keeping track of which features are actually modified during the geometric and algebraic constraint solving process. So, in this example, *only* the protrusionRight and its dependent blindHoleRight need to be updated in the Cellular Model. This is carried out by removing the cells of the protrusionRight and the blindHoleRight from their original position, 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 protrusionRight was the first feature attached to the block or not (see Fig. 15a). So the computational cost of modification modelling operations is dependent on the number of modified features, and not on the number of features in the model, as in history-based modelling (see Section 2.1). In the latter, after the protrusionRight displacement operation, the whole model history (at least since the protrusionRight creation) is re-executed, including features whose imprint remains unaltered, e.g. cylindrical-Protrusion and blindHoleTop in the model history at the right-hand side of Fig. 15a. Even worse, a history-based modelling system would not be able to perform this operation if the model history were that at the left-hand side of Fig. 15a, because the cornerTopRight is there more recent than the protrusionRight (see discussion of this drawback in Section 2.3).

5.6. History-independent interpretation of the Cellular Model

Interpretation of the Cellular Model consists of determining whether the point set represented by each cell does belong to, or represent "material" of, the product, i.e. the

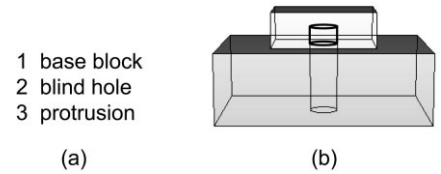


Fig. 16. The precedence sequence (a) for the model of Fig. 3e yields an incorrect nature for the cell highlighted in (b).

nature of that cell. This requires deciding which of the features in its owner list "prevails", either as additive or as subtractive [4]. It is only at this stage that the precedence among features needs to be taken into account.

5.6.1. 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 therefore numerable.

According to the above, the nature of a cell, whose owner list has n elements, is exclusively determined by the nth element (the last feature) in the owner list. Obviously, the nature of a cell is independent of features that do not occur in its owner list. For example, referring to the model in Fig. 14, the nature of the cell of the cornerBotRight is independent of whether the precedence numbers of, say, the cylindricalProtrusion, the protrusionRight and the blindHoleRight are higher or lower than its own precedence number.

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 therefore enough to have a procedure that is always able to generate *one* such sequence. We now discuss appropriate precedence criteria to achieve this goal.

5.6.2. Precedence criteria

The example in Figs. 14 and 15 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 attachment of the protrusionRight requires the cornerTopRight to precede the protrusionRight after the operation. Otherwise, the precedence number of the protrusionRight would be lower than that of the cornerTopRight, 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 modelling operation. Stated differently, 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 features highlighted in the model of Fig. 14a, for example, one can identify the following two precedence relations, based on an attachments' analysis: (i) the protrusionRight precedes the blindHoleRight (i.e. the latter is dependent on the former); and (ii) the block precedes all other features (i.e. they are all dependent on it). Relative precedence among most other features is irrelevant when it comes to interpret this model. So, for example, both feature precedence sequences of Fig. 15a produce the same model interpretation of Fig. 14a. Fig. 15b, on the contrary, shows a possible sequence of precedence numbers for the modified model in Fig. 14b. Whatever the sequence of precedence numbers before the operation, it can be remarked that the cornerTopRight now precedes the protrusionRight (i.e. has a lower precedence number), as required by the new attachment of the latter.

In short, the dynamic dependency relations in the Feature Dependency Graph permanently reflect the current structure of the feature model. Therefore, they make 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 determine a precedence relation between dependent features only. However, for the modelling operation described in Section 2.2, Fig. 3, a precedence problem was pointed out between two independent features, the blind hole and the protrusion: if the precedence numbers after the operations were kept as

```
FeaturesInvolved = {features involved in the modelling operation}
for each fi in FeaturesInvolved
   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
```

Fig. 17. Precedence detection algorithm for overlapping independent features.

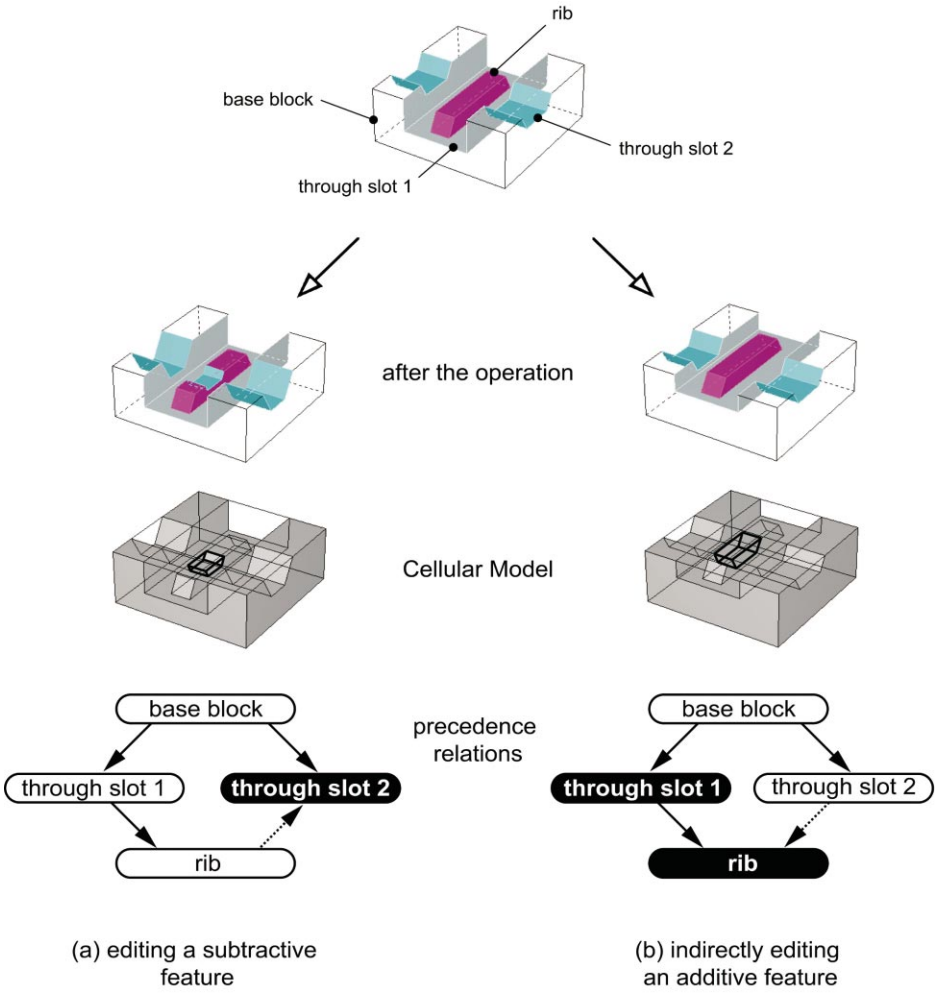


Fig. 18. Cellular Model interpretation after a modification operation.

shown in Fig. 16a, i.e. following the sequence of the history in Fig. 3c, the top interaction cell of the blind hole (highlighted in Fig. 16b) would be additive, i.e. have the nature of the protrusion. This nature is, however, incorrect, because it is not in accordance with the semantics of the modelling 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 Figs. 3 and 16, is that the modelling operation in question causes an *overlap* between two *independent features of different natures*. To avoid incorrect interpretation of a model such as that shown in Fig. 16, an explicit precedence relation should be established when, as a result of a modelling operation, two independent features with different natures come to overlap. The question that arises is then *which* orientation should be assigned to this precedence relation. The above example suggests that, to preserve the semantics of a modelling operation in such cases, 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. So after the operation in Fig. 3, the protrusion should precede the blind hole. Hence the following:

> *Criterion II.* To each *new* overlap between two independent features f_1 and f_2 of different natures, caused by some modelling 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 overlap between independent features, a global sorting of all features in the model can be achieved. We now show how such precedence criteria are used to produce a correct interpretation of the Cellular Model, which is unambiguously determined without invoking any model history considerations.

5.6.3. Computation of feature precedence relations

The precedence relation defined above is an example of a so-called *partial ordering* relation, i.e. a relation that defines

an ordering between *some* pairs of elements in a set S, but not among all of them.

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 modelling 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 modelling operation, i.e. those which are actually processed in the incremental re-evaluation of the Cellular Model (see Section 5.5), is analysed according to the algorithm in Fig. 17. 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. Determination of $OS(f)$ is straightforward and requires no geometric computations: it is simply computed as the union of the owner lists of all cells of feature f. $OS(f_i)$ of each feature f_i involved in the operation is computed and stored before the Cellular Model is re-evaluated, and compared with $OS(f_i)$ determined after the re-evaluation, in order to detect new overlaps.

Once the precedence relations have been established, using the two criteria described, global sorting of the 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, see for example Wirth [38]. Such an algorithm builds a new sorted sequence by iteratively selecting from the old sorted sequence a feature whose precedents are already sorted. Eventually, the features in the resulting sorted sequence have new precedence numbers assigned, and the nature of all cells becomes therefore automatically determined.

The example of Fig. 18 illustrates the interpretation of the Cellular Model. In this example, two modelling operations are performed that involve changes in one or more features. The initial model (see top of Fig. 18) has two crossing slots of different depths attached to a base block, and a rib attached to the bottom face of the deeper slot, through slot 1. The resulting models are shown for both operations, together with the corresponding Cellular Model and the graph of precedence relations used in its interpretation. In the graphs, 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 of Fig. 17, are drawn with a dotted line, to distinguish them from the other precedence relations, derived from the dependencies in the Feature Dependency Graph.

In the first operation (see Fig. 18a), the depth of the split through slot 2 is increased, so that it overlaps with the rib. As these two features are independent, their overlap leads to a precedence relation being established between them. Consequently, the rib receives a lower precedence number than through slot 2, and their interaction cell is therefore subtractive. With history-based boundary re-evaluation, the resulting model of Fig. 18a would not be feasible if through slot 2 had been created before the rib.

In the second operation (see Fig. 18b), the depth of through slot 1 is decreased, so that its dependent rib becomes in interaction with through slot 2. In this case, from the analysis of the precedence detection algorithm, the (indirectly) modified rib is preceded by the independent through slot 2, resulting in an additive nature for their interaction cell, highlighted in Fig. 18b. Again, the detection of the new overlap, and the precedence relation established, yields a model interpretation in which the nature of the modified features prevails over that of the other overlapping features. To achieve the model of Fig. 18b using a history-based modelling system, through slot 2 should have been created before the rib (which is exactly the history sequence that would make the resulting model of Fig. 18a unfeasible).

Summarising, precedence numbers are revised after every modelling operation. For this, precedence relations are updated in the model, and a new sorting is performed among all its features, which then get new precedence numbers assigned, reflecting the new model structure.

6. Feature model validity maintenance

Embedding validity criteria in each feature class, as described in Section 4, enhances the modelling process in the sense that at the creation of a feature instance its semantics matches the specific requirements of its class. However, this might no longer be the case when in any subsequent operation, the shape imprint of the instance would be arbitrarily modified, and therefore the latter should be prevented by the modelling system.

Feature model validity maintenance is the process of monitoring each modelling operation in order to ensure that all features conform to the validity criteria specified in their respective classes [3]. Maintaining feature model validity throughout the modelling process guarantees that all aspects of the design intent once captured in the model are permanently maintained. Together with the declarative validity specification scheme described in Section 4, feature model validity maintenance forms the core of the semantic feature modelling approach.

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

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

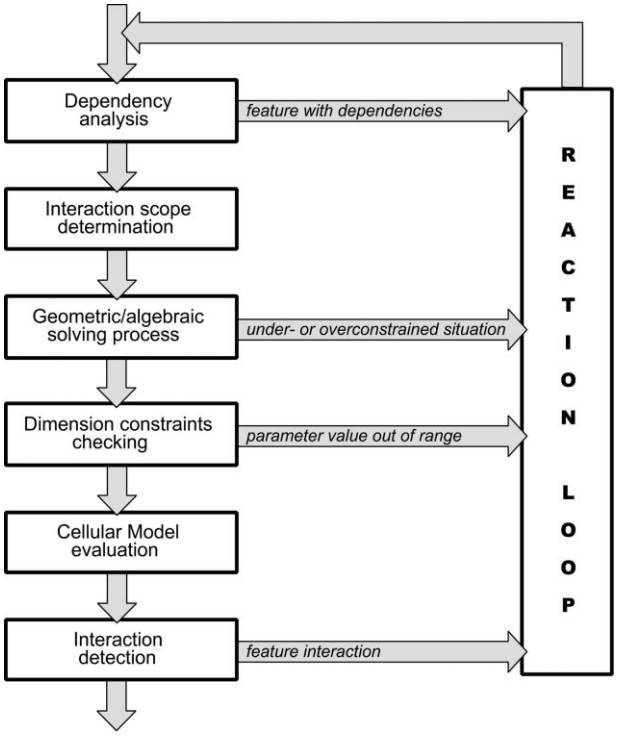


Fig. 19. Generic scheme of a modelling operation.

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

2. After an *invalid* modelling operation, the user is 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 modelling process.

Validity maintenance tasks can be classified into two types: *validity checking*, performed at key stages of each modelling 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.

6.1. Validity checking

As mentioned above, the first basic principle of model validity maintenance is that a *valid* modelling 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 modelling operation, the feature model conforms to all its constraints.

Modelling operations were introduced in Section 5.4, and classified as feature operations and constraint operations. The generic scheme of the execution of a modelling operation is presented in Fig. 19, 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 that 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.

6.1.1. Dependency analysis

This step is only required for 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, they are collected and the operation enters the reaction loop.

6.1.2. Interaction scope determination

The determination of the *feature interaction scope* (FIS) is performed at this stage, by collecting all feature instances in the model that may potentially be affected by the operation. Its purpose is to optimise the interaction detection phase (last step in Fig. 19), by avoiding checking features that are known in advance to be left unaffected by the operation [5].

6.1.3. Geometric and algebraic solving process

This step is required by all modelling 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 [20], and a SkyBlue algebraic constraint solver [33]. The iterative co-operation of these solvers is described by Dohmen [14].

At this stage, modelling 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 interface parameter values provided by the user, are not sufficient to uniquely determine and fix the degrees of freedom of all features in the model [26].

In both cases, the operation enters the reaction loop.

6.1.4. 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 modelling operation is considered invalid if some feature dimension parameter is out of the range specified by the respective constraint.

6.1.5. Cellular Model 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 therefore be updated, so that the effects of the operation are also reflected in the evaluated geometric model. This process has already been described in Sections 5.5 and 5.6.

6.1.6. Interaction detection

Once the Cellular Model has been updated, detection of disallowed feature interactions takes place. At this stage, a modelling operation is considered invalid if any boundary or interaction constraint is violated for some feature in the FIS, previously determined. Details on the interaction detection methods and algorithms can be found in Ref. [5]. Eventually, the set of constraint violations, if any, is analysed, and their causes are identified and passed to the reaction loop.

6.2. Validity recovery

When a modelling operation is *invalid*, for any reason pointed out in the previous section, a valid model should be achieved again. This is straightforward if the modelling operation is cancelled: all that is needed is to backtrack to the valid model state just before executing it, by "reversing" the invalid operation.

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, 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 emphasises 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*, see Fig. 19—which is activated whenever an operation turns out to be invalid. The user can then specify several modelling 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 Fig. 19, which means that their outcome is analysed, checking for validity at each stage, just as for "direct" modelling operations. The reaction loop is only exited when, because 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 first operation that entered the reaction loop.

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 Fig. 19, we have the following:

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 concrete effects (e.g. a feature face or a parameter affected). According to the particular interaction type (see Table 1 in Section 4), specific reaction choices are given, for example:

- *transmutation interaction*—replace the transmutated feature by another feature instance of the identified feature class (for example, after enlarging the step in the model in Fig. 1, the user might replace the blind holes by through holes);
- *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—*Step 1*);
- *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

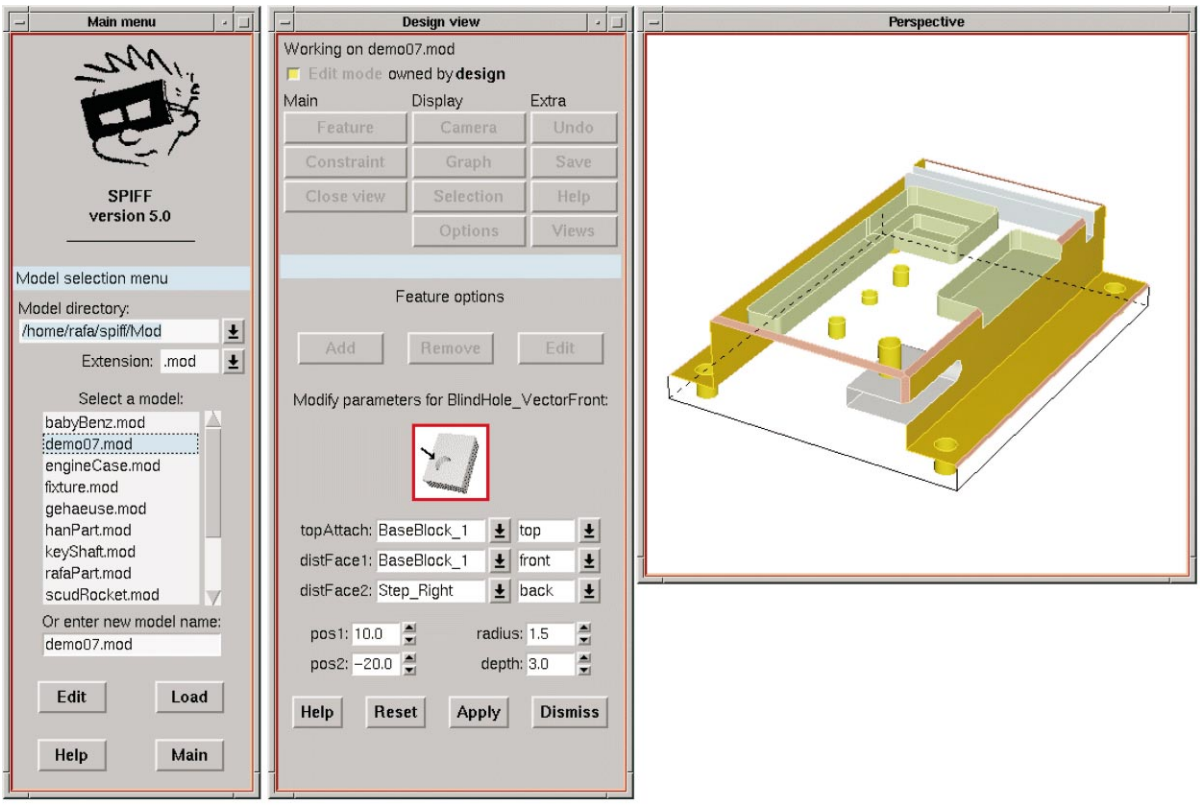


Fig. 20. Starting a modelling session in SPIFF.

constraints that are somehow involved in the invalid situation (i.e. features that overlap or have a dependency relation with the affected feature). This helps the user in concentrating validity recovery efforts on effective and meaningful reactions.

7. Example modelling session

The usefulness of the validity checking and recovery mechanisms is illustrated in this section with examples taken from a modelling session with the SPIFF system. For this, we use a model that is a variant of the part DEMO07, originally from ICEM-CDC, and made available at the NIST Design, Planning and Assembly Repository [30], a large collection of parts from industry and academia.

The user starts the modelling session by opening the model (see Fig. 20). For each subsequent modelling step, the invalid situation reported occurs because the underlying feature classes do specify the violated validity criteria reported at that stage.

Step 1 (Fig. 21): The user creates a rounded pocket on the top face of the model, overlapping with the pattern of blind holes. As a result, one of the blind holes is totally absorbed, whereas the depth of the other two is decreased. The system reports these absorption and geometric interactions, and the user corrects them by re-attaching all three blind holes to the bottom face of the rounded pocket, as shown in the model of Fig. 22a.

Step 2 (Fig. 22): Next, the user attaches a cylindrical protrusion on the top face of the model, covering the top entrance of the through hole (see Fig. 22b). Because the through hole becomes blind, the system reports a transmutation interaction. To recover from this interaction, the user decides to re-attach the through hole, from the top face of the block to the top face of the cylindrical protrusion, making it a through hole again (see Fig. 23a).

Step 3 (Fig. 23): Subsequently, the user chooses a variant of the part without the rounded blind slot, highlighted in Fig. 23a, and issues its removal from the model. Because both the through hole and the chamfer are attached to it, the system requires these dependencies to be eliminated prior to removing the blind slot. Removal of the dependent features from the model and their modification are among the possible reactions suggested by the system. In this case, the user chooses to re-attach the through hole to the bottom face of the block, and to remove the chamfer, after which the model in Fig. 23b is achieved.

Step 4 (Fig. 24): Finally, the user increases the width of the step at the right-hand side of the model (see Fig. 24a). Because the pocket, the cylindrical protrusion and, consequently, the blind hole are positioned relative to the step,

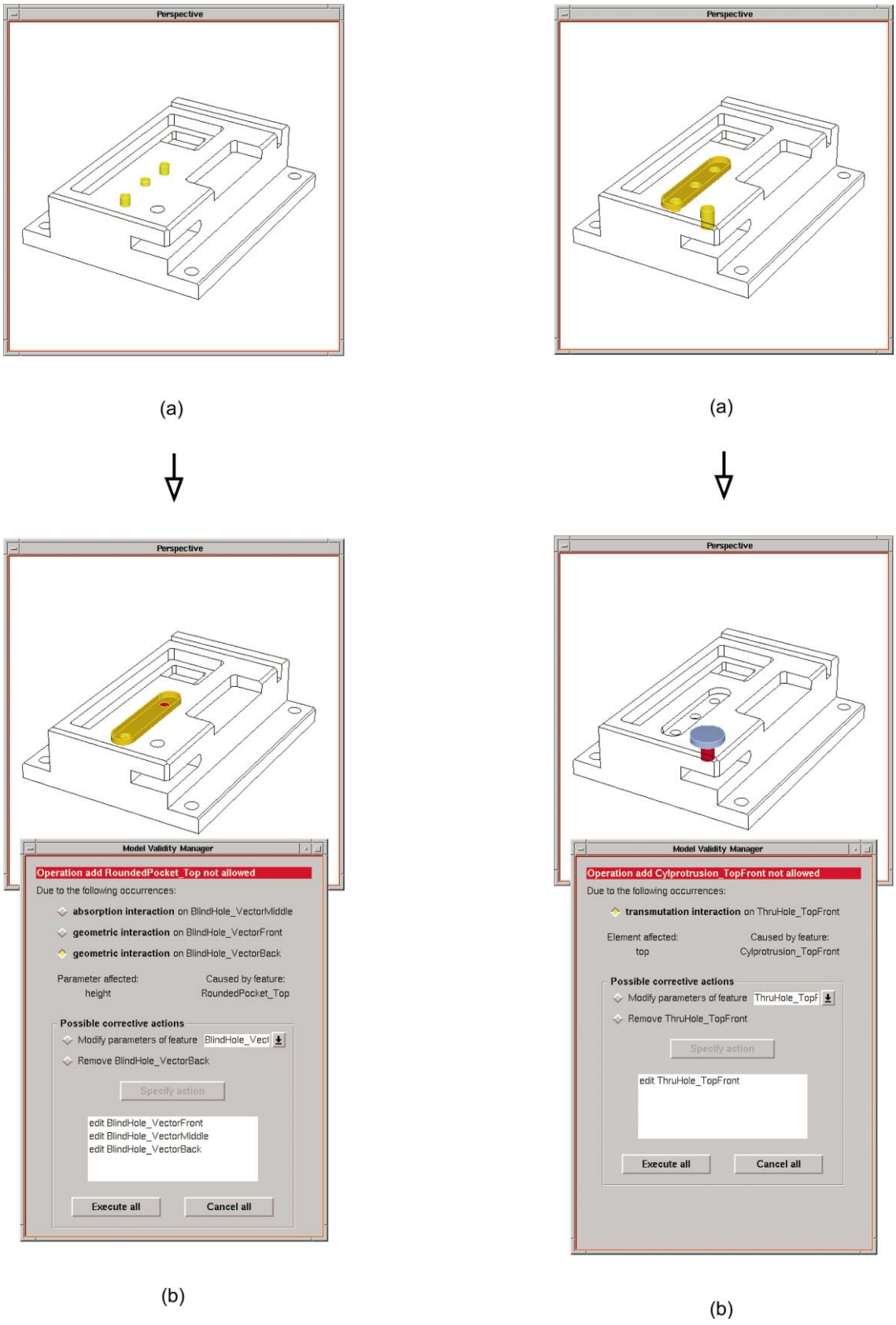


Fig. 21. *Step 1:* reporting geometric and absorption interactions.

Fig. 22. *Step 2:* reporting a transmutation interaction.

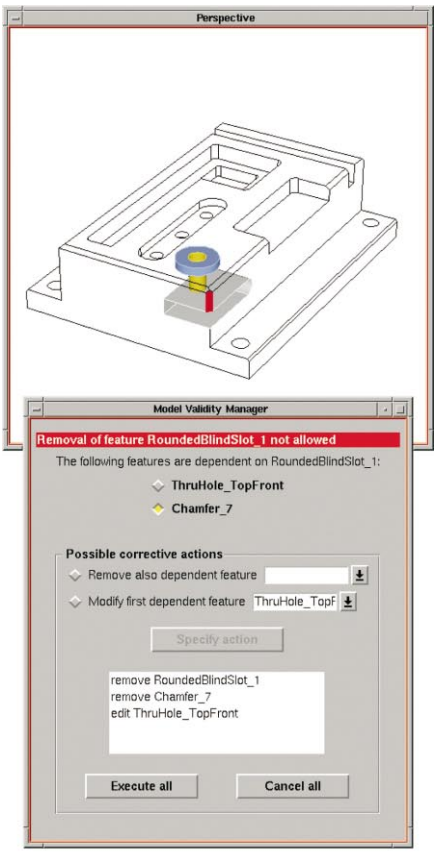


Fig. 23. *Step 3:* reporting dependencies before a feature removal.

they are displaced accordingly. Consequently, the cylindrical protrusion partially occludes the rounded pocket, and this boundary clearance interaction is reported by the system. As a reaction to this, the user may readjust the step width, displace the cylindrical protrusion or reduce its radius (or choose a combination of these reactions), after which a valid model is achieved again (see Fig. 24b).

8. Conclusions

There are several important characteristics that distinguish the semantic feature modelling approach from current feature modelling approaches. In this section, these approaches are compared on their merits.

The most salient characteristic of semantic feature modelling is that the semantics of all features is well defined and maintained during the whole modelling process. The use of various constraint types for validity conditions in generic feature classes allows specification of many semantic aspects for the instances of each class. Among these constraints, those specifying disallowed 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 analyse any invalid situation that might result from some modelling 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 modelling freedom of the user. For example, the user might actually want to turn blind a through hole. In current feature modelling 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 modelling approach, on the contrary, 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. Obviously, this reaction is only possible if a blind hole class is available in the feature library of the system. So only models with features that are allowed, by their inclusion in the feature library, can be created. This example demonstrates how in semantic feature modelling, by imposing restrictions on the modelling freedom, the user is assisted in creating valid models only.

A correspondence with programming languages can be noted here. Geometric modelling, but also current feature modelling practice, is in a way comparable to the use of low-level programming languages, with low-level

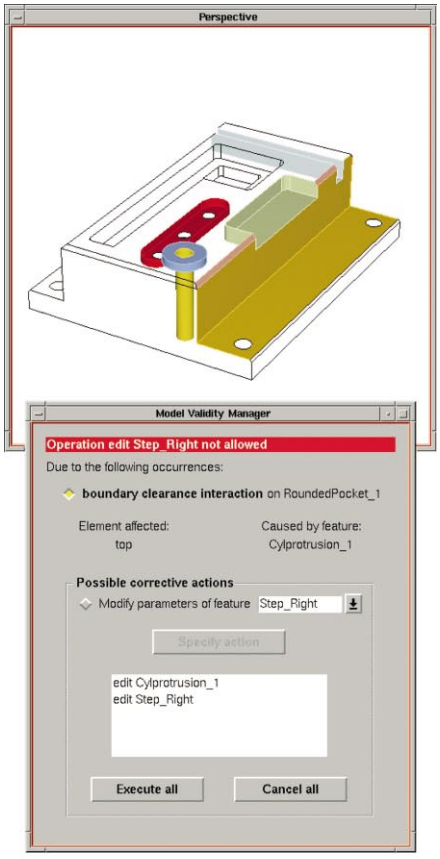


Fig. 24. *Step 4:* reporting a boundary clearance interaction.

operations. Such languages offer much 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 modelling, on the contrary, 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 better 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, less error-prone programs. Similarly, we believe that the loss of "modelling freedom" in semantic feature modelling 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 modelling solves several other problems that occur in current feature modelling systems. In particular, there is no longer a dependency on the chronological order in which features are added to a model. This results in a significant improvement in model modification and dimensioning facilities. Shortcomings originating from the persistent naming problem in history-based modelling are also avoided, because all modelling operations work on feature faces, instead of boundary model faces, and, therefore, ambiguities in names cannot occur. Stated differently, in semantic feature modelling the semantics of modelling operations is in this respect well defined, in contrast with history-based feature modelling.

The Cellular Model has several properties that make it very suitable for the geometric representation of feature models. In particular, the evaluation and interpretation of the Cellular Model are independent of the chronological order of feature creation, which solves several problems inherent to history-based modelling. Further, the Cellular Model contains all information required for interaction detection, and deals with subtractive and overlapping features in a consistent way during model editing.

The structure of the Cellular Model is certainly more complex than that of a manifold boundary representation, commonly used in current feature modelling systems. In addition, attribute storing and propagation mechanisms demand some additional processing not required by set operations on such a boundary representation. However, this is far outweighed by the performance improvement of incremental re-evaluation of the Cellular Model.

Building the whole Cellular Model from scratch has a computational cost that is roughly proportional to the model history size, as is the case for boundary re-evaluation in history-based modelling. Fortunately, this is only required when the Cellular Model needs to be built in one step, e.g. when starting a modelling session with a

previously created model file. Once this has been done, the computational cost of re-evaluating the Cellular Model after a modelling operation is not dependent of the total number of features in the model, but on the number of features whose geometry is actually affected by the operation. Usually, this number is very limited, so computational cost is minimised.

The semantic feature modelling approach has been successfully implemented in the SPIFF system. The implementation offers the full functionality described in this article. In particular because of the integration of the Cellular Model with the constraint solvers, interactive modelling of complex models, as used in some of the examples, turned out to be easily feasible.

To conclude, it has often been remarked that feature modelling is nothing more than advanced geometric modelling, only offering parametric and constraint-based modelling facilities in addition to the common geometric modelling facilities. This article, however, shows that semantic feature modelling is significantly more powerful than current feature modelling approaches, and effectively brings feature modelling to a much higher level than geometric modelling, not only with regard to applications, but also with regard to modelling facilities.

Acknowledgements

We thank Maurice Dohmen, Winfried van Holland, Erik Jansen, Klaas Jan de Kraker and Alex Noort for their contributions to the research on feature modelling done in our group. We also thank them, and Jiri Kripac, for valuable comments on a preliminary version of the manuscript. Rafael Bidarra's work has been supported by the Praxis XXI Program of the Portuguese Foundation for Scientific and Technological Research (FCT).

References

[1] Autodesk. Autodesk Mechanical Desktop 4.0 User's Guide. Autodesk, Inc., San Rafael, CA, USA, 1999.

[2] Bidarra, R. Validity maintenance in semantic feature modeling. PhD Thesis. Delft University of Technology, The Netherlands, 1999.

[3] Bidarra R, Bronsvoort WF. Validity maintenance of semantic feature models. In: Bronsvoort WF, Anderson DC, editors. Proceedings of Solid Modeling'99—Fifth Symposium on Solid Modeling and Applications, 9–11 June, Ann Arbor, MI, USA, New York: ACM Press, 1999. p. 85–96.

[4] Bidarra R, Bronsvoort WF. History-independent boundary evaluation for feature modeling. CD-ROM Proceedings of the 1999 ASME Design Engineering Technical Conferences, 12–15 September, Las Vegas, NV, USA, New York: ASME, 1999.

[5] Bidarra R, Dohmen M, Bronsvoort WF. Automatic detection of interactions in feature models. CD-ROM Proceedings of the 1997 ASME Design Engineering Technical Conferences, 14–17 September, Sacramento, CA, USA, New York: ASME, 1997.

[6] Bidarra R, Idri A, Noort A, Bronsvoort WF. Declarative user-defined feature classes. CD-ROM Proceedings of the 1998 ASME Design Engineering Technical Conferences, 13–16 September, Atlanta, GA, USA, New York: ASME, 1998.

[7] Bidarra R, de Kraker KJ, Bronsvoort WF. Representation and management of feature information in a cellular model. Computer-Aided Design 1998;30(4):301–13.

[8] Bidarra R, Teixeira JC. A semantic framework for flexible feature validity specification and assessment. In: Ishii K, Bannister K, Crawford R, editors. Proceedings of the 1994 ASME Computers in Engineering Conference, September, Minneapolis, MN, USA, 1. New York: ASME, 1994. p. 151–8.

[9] Bronsvoort WF, Bidarra R, Dohmen M, van Holland W, de Kraker KJ. Multiple-view feature modelling and conversion. In: Strasser W, Klein R, Rau R, editors. Geometric modelling: theory and practice—the state of the art, Berlin: Springer, 1997. p. 159–74.

[10] Bronsvoort WF, Jansen FW. Feature modelling and conversion—key concepts to concurrent engineering. Computers in Industry 1993;21(1):61–86.

[11] Brunetti G, Ovtcharova J, Vieira AS. A proposal for a feature description language. In: Roller D, editor. 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, Croydon, England: Automotive Automation, 1996. p. 117–24.

[12] Capoyleas V, Chen X, Hoffmann CM. Generic naming in generative, constraint-based design. Computer-Aided Design 1996;28(1):17–26.

[13] Chen X, Hoffmann CM. On editability of feature-based design. Computer-Aided Design 1995;27(12):905–14.

[14] Dohmen M. Constraint-based feature validation. PhD thesis. Delft University of Technology, The Netherlands, 1997.

[15] Dohmen M, de Kraker KJ, Bronsvoort WF. Feature validation in a multiple-view modeling system. In: McCarthy JM, editor. CD-ROM Proceedings of the 1996 ASME Design Engineering Technical Conferences and Computers in Engineering Conference, 19–22 August, Irvine, CA, USA, New York: ASME, 1996.

[16] van Emmerik MJGM, Jansen FW. User interface for feature modelling. In: Kimura F, Rolstadas A, editors. Computer applications in production and engineering, Amsterdam: Elsevier, 1989. p. 625–32.

[17] Henderson M.R. Extraction of feature information from three-dimensional CAD data. PhD thesis. Purdue University, West Lafayette, IN, USA, 1984.

[18] Hoffmann CM, Joan-Arinyo R. On user-defined features. Computer-Aided Design 1998;30(5):321–32.

[19] de Kraker KJ, Dohmen M, Bronsvoort WF. Multiple-way feature conversion to support concurrent engineering. In: Hoffmann CM, Rossignac JR, editors. Proceedings of Solid Modeling'95—Third Symposium on Solid Modeling and Applications, 17–19 May, Salt Lake City, UT, USA, New York: ACM Press, 1995. p. 105–14.

[20] Kramer GA. Solving geometric constraint systems: a case study in kinematics. Cambridge, MA: MIT Press, 1992.

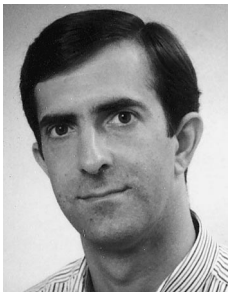
[21] Kripac J. A mechanism for persistently naming topological entities in history-based parametric solid models. In: Hoffmann CM, Rossignac JR, editors. Proceedings of Solid Modeling'95—Third Symposium on Solid Modeling and Applications, 17–19 May, Salt Lake City, UT, USA, New York: ACM Press, 1995. p. 21–30. Also in: Computer-Aided Design 1997;29(2):113–22.

[22] Kyprianou L.K. Shape classification in computer-aided design. PhD thesis. Cambridge University, UK, 1980.

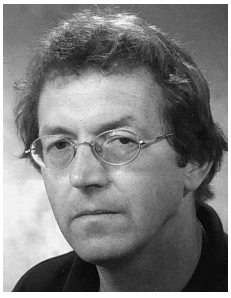
[23] Lequette R. Considerations on topological naming. In: Pratt M, Sriram RD, Wozny MJ, editors. Product Modeling for Computer Integrated Design and Manufacturing—Proceedings TC5/WG5.2 International Workshop on Geometric Modeling in Computer Aided Design, 19–23 May 1996, Airlie, VA, USA, London: Chapman & Hall, 1997. p. 394–403.

[24] Luby SC, Dixon JR, Simmons MK. Designing with features: creating and using a features data base for evaluation of manufacturability in castings. Proceedings of the 1986 ASME Computers in Engineering Conference, August, Chicago, IL, USA, New York: ASME, 1986.

[25] Mandorli F, Cugini U, Otto HE, Kimura F. Modeling with self-validating features. In: Hoffmann CM, Bronsvoort WF, editors. Proceedings of Solid Modeling'97—Fourth Symposium on Solid Modeling and Applications, 14–16 May, Atlanta, GA, USA, New York: ACM Press, 1997. p. 88–96.
[26] Noort A, Dohmen M, Bronsvoort WF. Solving over- and under-constrained geometric models. In: Brüderlin B, Roller D, editors. Geometric Constraint Solving and Applications, Berlin: Springer, 1998. p. 107–27.
[27] Parametric. Pro/ENGINEER Modeling User's Guide, Version 20. Parametric Technology Corporation, Waltham, MA, USA, 1999.
[28] Peters BF. MicroStation modeler: the design and implementation of an extensible solid modeling system. In: Strasser W, Klein R, Rau R, editors. Geometric modeling: theory and practice—the state of the art, Berlin: Springer, 1997. p. 361–78.
[29] Raghothama S, Shapiro V. Boundary representation deformation in parametric solid modeling. ACM Transactions on Graphics 1998; 17(4):259–86.
[30] Regli W, Gaines DM. A repository for design, process planning and assembly. Computer-Aided Design 1997;29(12):895–905.
[31] Regli W, Pratt M. What are feature interactions? In: McCarthy JM, editor. CD-ROM Proceedings of the 1996 ASME Computers in Engineering Conference, 19–22 August, Irvine, CA, USA, New York: ASME, 1996.
[32] Rossignac JR. Issues on feature-based editing and interrogation of solid models. Computers & Graphics 1990;14(2):149–72.
[33] Sannella M. The SkyBlue constraint solver. Technical Report 92-07-02. University of Washington, Seattle, WA, 1992.
[34] SDRC. I-DEAS Master Series 7 User's Guide. Structural Dynamics Research Corporation, Milford, OH, USA, 1999.
[35] Shah JJ, Rogers MT. Expert form feature modelling shell. Computer-Aided Design 1988;20(9):515–24.
[36] Spatial. Acis 3D Modeling Kernel, Version 5.3. Spatial Technology Inc., Boulder, CO, USA, 1999.
[37] Vieira AS. Consistency management in feature-based parametric design. In: Gadh R, editor. Proceedings of the 1995 ASME Design Engineering Technical Conferences, 17–21 September, Boston, MA, USA, 2. New York: ASME, 1995. p. 977–87.
[38] Wirth N. Algorithms + Data Structures = Programs. Englewood Cliffs, NJ: Prentice-Hall, 1976.



Rafael Bidarra is a postdoctoral researcher at Delft University of Technology. He graduated in electronics engineering at the University of Coimbra, Portugal, in 1987, and received his PhD degree from Delft University of Technology in 1999. The subject of his PhD thesis was validity maintenance in semantic feature modelling. His main research interests are feature modelling and collaborative modelling.



Willem F. Bronsvoort is an Associate Professor at Delft University of Technology. He received his master's degree in computer science from the University of Groningen in 1978 and his PhD degree from Delft University of Technology in 1990. His main research interests are geometric modelling, including display algorithms, and feature modelling, including feature validity maintenance and multiple-view feature conversion. He has published numerous papers in international journals, books and conference proceedings, is book review editor of Computer-Aided Design, and has served as program co-chair of Solid Modeling'97 and '99 and as a member of several program committees.