Validity Maintenance of Semantic Feature Models

Rafael Bidarra and Willem F. Bronsvoort
Faculty of Information Technology and Systems
Delft University of Technology

Abstract
One of the most powerful characteristics of feature-based modeling is the ability to associate functional and engineering information to shape information in a product model. Current feature modeling systems embody this paradigm in their graphical user interfaces, providing the user with "engineering rich" dialogs aimed at the creation of feature instances. Most systems, however, fail to consistently maintain the meaning of the features throughout the modeling process. For example, a modeling operation on one feature may affect the semantics of other features without the user being notified by the system, let alone assisted in overcoming the situation.

Semantic feature modeling is a declarative modeling approach that not only provides a well-defined specification of feature semantics, but also effectively maintains this semantics during the modeling process, for all feature instances in the model. This paper describes the validity maintenance mechanisms of the semantic feature modeling approach. These include (i) detecting each invalid situation, (ii) reporting it to the user, with appropriate explanation on its causes and effects, and (iii) providing the user with a convenient choice of reaction hints, aimed at recovering validity in the model. An example modeling session is also given, illustrating which high-level user assistance is provided under this approach.

Keywords: feature modeling, feature semantics, validity maintenance, validity recovery

1 INTRODUCTION

Current feature modeling systems provide the user with "engineering rich" dialogs aimed at the creation and manipulation of feature instances. In some systems, "features" occur solely at the user interface level, whereas in the product model only the resulting geometry is stored. Such systems are in essence only geometric modelers. Most other feature modeling systems, see for example [15], although they store information about features in the product model, fail to adequately maintain the meaning of features throughout the modeling process. For example, a modeling operation on one feature may affect the semantics of other features, without the user even being notified by the system, let alone assisted in overcoming the situation.

This is illustrated in the example of Figure 1 on the next page. Assume that the two longer blind holes in the part were positioned relative to the block right side face, whereas the rounded pocket was positioned relative to the step side face, as indicated in Figure 1.a. If the width of the step is now increased, the rounded pocket overlaps with the two blind holes, "suppressing" their circular bottom faces from the model boundary, see Figure 1.b. Consequently, the two blind holes have now the shape imprint of through holes. Although geometrically this is correct, it is incorrect in the sense that the meaning, or 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.

Reference [16] provides a good insight into some high-level feature validity issues, alerting for inconsistencies that might arise from a naive interpretation of usual editing commands on feature models. Some more recent research work has focused on validation of features, both validity specification [7,10] and validity maintenance issues [9, 13]. One of the main conclusions of this research is that a declarative scheme is preferable over the conventional procedural modeling approaches. In a declarative approach, the specification of each feature class includes the validity criteria that determine the semantics of all its feature instances. The feature modeler, in turn, is responsible for the maintenance of all features in the product model, in conformity with those criteria.

Research prototype systems that do have some form of validity maintenance, see for example [18, 8], are limited to the detection of a number of predefined invalid situations, for which the only solution offered by the modeling system is the rejection of the concerning modeling operation. This rigid scheme considerably hinders
the modeling process, yet permitting many unanticipated inconsistencies in the model.

Semantic feature modeling is a new declarative modeling approach [1]. It provides a well-defined constraint-based framework for the specification of feature semantics in each feature class. Furthermore, it effectively maintains this semantics for all features in the product model, throughout the modeling process.

This paper focuses on the validity maintenance aspects of the semantic feature modeling approach. In particular, it emphasizes how assistance can be offered to the user of the modeling system, in order to keep the design intent in a model in a large variety of situations.

First, the main aspects of validity specification in a feature class are presented (Section 2). Next, the main principles of validity maintenance are discussed (Section 3). These are further elaborated into validity checking (Section 4) and validity recovery (Section 5). An example modeling session is also given, illustrating the high-level user assistance provided (Section 6). Finally, some conclusions are drawn on the present work (Section 7).

2 VALIDITY SPECIFICATION IN FEATURE CLASSES

Feature class specification involves specification of its shape, its validity conditions, and its interface to the feature model, according to the general structure depicted in Figure 2. 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 parameterized 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 labeled with its own generic name, to be used in all modeling operations. For example, a cylinder shape has a top, a bottom and a side face.

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

Figure 1 – Changing feature semantics with a modeling operation

Figure 2 – Feature class structure
The specification of validity conditions in a feature class
can be classified into three categories: geometric, toto-
logic 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 be-
tween 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, alge-
braic 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 modeled part. Some faces, instead, have
a closure role, delimiting the feature volume without
contributing to the model boundary. The specification of
such properties is called totoologic validity specification.

To specify totoologic validity in a feature class, we use
semantic constraints on each shape face. Semantic con-
straints, first proposed in [5], specify which topological
variants of a feature instance are allowed, by stating the
extent to which its feature faces should be on the model
boundary. Semantic constraints are of two types: on-
Boundary, which means the shape face should be present
on the model boundary, and notOnBoundary, which
means the shape face should not be present on the model
boundary. Furthermore, both types of semantic con-
straints are parameterized, stating whether the presence
or absence on the model boundary is completely or only
partly required. An example of this is a blind hole class
for which the top face has a notOnBoundary(comple-
tely) constraint, the side face has an onBoundary(partly)
constraint, and the bottom face has an onBoundary(com-
pletely) constraint.

Geometric and totoologic 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.

Such functional requirements can be violated by fea-
ture interactions caused during incremental editing of
the model. Feature interactions are modifications of the
shape aspects represented by a feature that affect its
functional meaning. An example of this is the transmu-
tation interaction of the blind hole into a through hole in
Figure 1. A classification of feature interactions can be
found in [1]. For completeness, it is briefly summarized
here:

**Splitting** Interaction that splits the boundary of a
feature into two (or more) disconnected subsets.

**Disconnection** Interaction that causes the volume
of an additive feature (or part of it) to become
disconnected from the model.

**Boundary clearance** Interaction that causes (par-
tial) obstruction of a closure face of a subtractive
feature.

**Volume clearance** Interaction that causes partial
obstruction of the volume of a subtractive fea-
ture.

**Closure** Interaction that causes some subtractive
feature volume(s) to become a closed void inside
the model.

**Absorption** Interaction that causes a feature to
cause completely its contribution to the model
shape.

**Geometric** Interaction that causes a mismatch be-
tween a nominal parameter value and the actual
feature geometry.

**Transmutation** Interaction that causes a feature
instance to exhibit the shape imprint character-
istic of another feature class.

**Topological** Interaction that causes the violation
of a semantic constraint in a given feature.

We use interaction constraints in a feature class in
order to indicate that a particular interaction type is not
allowed for its instances [2].

Feature constraints and parameters may require exter-
nal 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 in-
stances will be presented to the user of the modeling sys-
tem and, thus, how the user will be able to interact with
them. Essential in the feature class interface is the posi-
tioning and orientation scheme, which is specified by
means of attach and geometric constraints, as depicted in
Figure 2.

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. At-
tach constraints are a kind of coplanar geometric con-
straints 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
dynamic 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 posi-
tion a through slot, a faceFaceFace 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 modeling approach can be found in [3].

3 VALIDITY MAINTENANCE

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

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

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

(i) A modeling operation, to be considered as valid, should yield a feature model that conforms to all constraints. This ensures that every feature in the model conforms to the designer intent explicitly specified up to that moment.

(ii) After an invalid modeling operation, the user should be assisted in overcoming the constraint violations in order to recover model validity again. This can reduce the frequency of backtracking by enlarging the choice of possible reactions towards validity recovery. In particular, explanations on what is causing a constraint violation, and context-sensitive corrective hints, can significantly improve the modeling process.

Together with the declarative validity specification scheme presented in Section 2, feature model validity maintenance forms the core of the semantic feature modeling approach.

This approach has been implemented in the SPIFF system, a prototype multiple-view feature-based modeller developed at Delft University of Technology [6]. Figure 3 depicts the architecture of the system. Several system modules have been described elsewhere [11, 9, 2], and will be only briefly summarized here.

The Feature Model Manager receives commands from the user via a graphical user interface, and trans-
lates them into elementary tasks, which are then dispatched to the other Managers. It is responsible for the control of all modeling operations, and for maintaining model validity. Furthermore, the Feature Model Manager maintains the Feature Dependency Graph, a high-level representation of the structure of the product [1].

The Feature Dependency Graph contains all feature instances in the product model, each of them with its own set of entities (e.g. shape elements, parameters and constraints), and all model constraint instances (i.e. constraints that are separately defined by the user, possibly between different features in the model, with the goal of further specifying design intent). These instances are interrelated by the dependency relation, yielding a directed acyclic graph structure, consisting of the set of all model entities (features and model constraints), and the set of dependency relations among these entities. A feature \( f_1 \) depends on a feature \( f_2 \) whenever \( f_1 \) is attached, positioned, or in some other way constrained relative to \( f_2 \) (i.e. some feature constraint of \( f_1 \) has a reference to some entity of feature \( f_2 \)).

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

The Cellular Model is a non-manifold representation of the feature model geometry, integrating the contributions from all features in the Feature Dependency Graph [4]. The Cellular Model represents a part's geometry as a connected set of volumetric quasi-disjoint cells, in such a way that each one either lies entirely inside a shape extent or entirely outside it. The cells represent the point sets of the shape extents of all features in the model. Each shape extent is, thus, represented in the Cellular Model by a connected subset of cells. 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 ones lie in either of them. In order to be able to search and analyze features and their faces in the Cellular Model, each cell has an attribute –called owner list– indicating which shape extents it belongs to, see Figure 4. Similarly, each cell face has also an owner list, indicating which shape faces it belongs to.

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

In the remainder of the paper, we will concentrate on the role of the Feature Model Manager. In particular, its validity maintenance tasks will be described in detail. These can be classified into two types of tasks:

1. **validity checking**, performed at key stages of each modeling operation;
2. **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.

### 4 VALIDITY CHECKING

As mentioned in Section 3, the first basic principle of model validity maintenance is that a valid modeling operation should entirely preserve the designer intent specified so far with each feature, as well as with all model constraints. In other words, after a valid modeling operation, the feature model conforms to all its constraints.

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

**Adding a new feature instance to the model** This operation creates a new feature instance of the chosen feature class, and requests from the user a full set of initialization parameter values for the new feature. Together with this, all constraint members specified in its class are also instantiated, and initialized with the

![Figure 4 – Cell owner lists in the Cellular Model](image)
corresponding user-supplied values for interface parameters (e.g. distance parameters and external feature faces for attach constraints).

**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.

The generic scheme of the execution of a modeling operation is presented in Figure 5, showing its main internal steps. Also shown in the diagram are the various points at which the operation can turn out to be invalid. Whenever this occurs, the operation branches into the reaction loop, instead of following the normal flow, and we say the model has entered an invalid state. We now concentrate on the description of the main steps in the diagram, and on the circumstances under which specific invalid situations may arise in each of these steps. An important goal here is to enter the reaction loop, if required, with sufficient knowledge of the current status of the model, so that it can be appropriately handled, reported to the user and, ultimately, overcome. The reaction loop itself will be dealt with in the next section.

### 4.1 Dependency Analysis

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

### 4.2 Interaction Scope Determination

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

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

- the set of features that overlap with f, either volumetrically or with their boundaries; these features make up the overlapping set of f, denoted OS(f), and they are identified by querying the Feature Geometry Manager, which keeps track of all feature shapes and their intersections in the Cellular Model (see Figure 4);

- the set of features that depend on f; these features make up the dependency set of f, denoted DS(f), and they are identified by querying the Constraint Manager, which recursively traces in the Feature Dependency Graph the dependency relations on f.

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

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

\[ \text{FIS} \leftarrow \{f\} \cup \text{OS}(f) \]

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

\[ \text{FIS} \leftarrow \{f\} \cup \text{DS}(f) \cup \text{OS}(f) \cup \bigcup_{f_i \in \text{DS}(f)} \text{OS}(f_i) \]

Figure 5 – Generic scheme of a modeling operation
in order to include those features whose overlap with feature \( f \) or with its dependent features will possibly cease after the operation.

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

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

With this scheme, interactions caused or suffered indirectly by any dependent feature are also detected.

**Removing a feature instance from the model** As pointed out above, this operation requires that the feature to be removed has no dependent features, i.e. \( DS(f) = \emptyset \). The FIS is therefore determined as

\[
FIS \leftarrow OS(f)
\]

4.3 Geometric and Algebraic Solving Process

This step is required by all modeling operations, except feature removal. Its goal is to determine or update the dimensions, position and orientation of all features in the model. This task is performed by the Constraint Manager, which deploys two dedicated constraint solvers: a geometric constraint solver based on extended 3D degrees of freedom analysis [12], and a SkyBlue algebraic constraint solver [17]. The iterative cooperation of these solvers, under the control of the Constraint Manager, is described in [8].

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

- (i) an overconstrained situation, i.e. some feature(s) have conflicting geometric and/or algebraic constraints, or
- (ii) 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 [14].

In both cases, the operation enters the reaction loop.

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

4.5 Cellular Model Re-evaluation

When this step is reached, each feature in the Feature Dependency Graph has all its parameters successfully updated. In particular, all feature shape extents have their dimensions, position and orientation fully determined. The Cellular Model may thus be updated, so that the effects of the operation are also reflected in the evaluated geometric model. Detailed Cellular Model processing algorithms can be found in [4]. According to the particular feature operation, these can be summarized as follows:

**Adding a new feature instance to the model** The shape extent of the new feature is added to the current Cellular Model. For this, the nonregular cellular union operation is used, which computes the cellular decomposition described in Section 3, 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 in $DS(f)$ that are also modified by the operation, need to be taken into account. These are removed from the Cellular Model and then re-added with their new parameters, using the add and remove operations just described.

4.6 Interaction Detection

Once the Cellular Model has been updated, detection of disallowed interactions takes place. At this stage, a modeling operation is considered invalid if any semantic or interaction constraint is violated for some feature in the FIS, previously determined. Each constraint violation is recorded by the Interaction Manager. The other managers are queried, in order to obtain the specific data required by each interaction detection algorithm. Details on the interaction detection methods and algorithms can be found in [2]. Eventually, the set of constraint violations, if any, is analyzed, and their causes are identified and passed to the reaction loop.

5 VALIDITY RECOVERY

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

- **Adding a new feature instance to the model** The added feature is removed from the model, using the feature removal operation.
- **Removing a feature instance from the model** The removed feature is added back to the model, using the feature adding operation with the original parameter values.
- **Editing a feature instance in the model** The original parameter values of the edited feature are restored, using another feature editing operation, in all regards similar to the first operation.
- **Constraint operations** Each of them is reversed similarly to the feature operations (i.e. added constraints are removed, modified constraints are restored, etc.).

Reversing a modeling operation can be done very efficiently under our approach. The parameter values possibly required for undoing each modeling operation are kept in a log, the so-called operations stack. Every modeling operation is registered in this stack, as well as the information whether it led the model to a valid state or not. Undoing is therefore always possible, at any moment in a modeling session, by popping operations from the stack and executing their reverse operation until a “valid state” marker is found. This is depicted in Figure 6: assuming the insertion of the stiffener is invalid, that operation (the last on the operations stack) is popped from the stack and undone to restore the original situation.

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

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

To achieve this, a corrective mechanism was devised – the reaction loop, represented in Figure 5 – which is activated whenever an operation turns out to be invalid. The user can then specify several modeling operations in a batch (typically editing features and/or model con-
In all cases above, the scope of the reaction choices made available to the user is restricted to those features and model constraints that are somehow involved in the invalid situation (i.e. features that overlap or have a dependency relation with the affected feature). This helps the user in concentrating validity recovery efforts on effective and meaningful reactions.

6 EXAMPLE MODELING SESSION

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

The user starts the modeling session with opening an existing model, see Figure 7.a. For each subsequent modeling step, the invalid situation reported occurs because the underlying feature classes do specify the validity criteria violated at that stage.

Step 1 (Figure 7) The user attaches a rib feature to the bottom of the through slot. The rib feature class, however, prescribes a minimum width value, not obeyed by this instance, thus the system reports a dimension constraint violation. The user corrects this by adjusting the rib width to the minimum value allowed, as shown in the model of Figure 8.a.

Step 2 (Figure 8) Subsequently, the user attempts an alternative design for the part, re-attaching the through slot from the top of the block to the bottom of the step, see Figure 8.b. Consequently, the rib feature, which is dependent on the through slot, is also displaced with it. However, the upper region of the rib intrudes into the subtractive volume of the step. This is disallowed by the validity criteria of the step (by means of a volumetric clearance interaction constraint), thus the operation is notified as invalid, and the situation is reported to the user.

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

Step 3 (Figure 9) By mistake, the user supplies too high a value for the slot depth, causing the model to become disconnected. Although the previous clearance interaction on the step is indeed overcome, now a new invalid situation—the model disconnection—occurs and is reported. As a reaction to this, the user may readjust the slot depth, specify a larger height for the block, or decrease the step depth (or a combination of these reactions). In this case, he chooses to decrease the slot depth, see Figure 9.b.
As remarked in the previous section, features that are irrelevant to overcome the invalid situation, for example the two blind slots, are not editable at this stage of the reaction loop.

**Step 4** (Figure 10) At this stage, the user chooses for a variant of the part without the step feature, and issues its removal from the model. Because the through slot is dependent on the step, and thus indirectly also the rib, the system requires these dependencies to be eliminated prior to removing the step. Removal of the dependent features from the model and modification of their attachments are among the possible reactions suggested by the system. The user chooses to re-attach the through slot to the top face of the block, by which its dependent rib is also automatically displaced, as shown in Figure 11a.
**Step 5** (Figure 11) The user proceeds with the design by attaching a through hole between the top and bottom faces of the block. By mistake, however, the two model faces chosen for positioning the through hole are parallel (the front and back faces of the block), and thus insufficient to determine its position. The underconstrained situation is reported to the user, who is asked to specify appropriate faces for positioning the through hole, after which a valid model is achieved again, see Figure 11.b.

**Step 6** (Figure 12) Finally, the user creates a pocket at the bottom face of the block, such that the through hole attached to it in the previous step becomes shorter. This geometric interaction is detected and reported by the system. The user reacts by re-attaching the through hole to the bottom face of the pocket, and takes the opportunity to slightly increase the depth of the pocket. The final model is shown in Figure 12.b.
7 CONCLUSIONS

Maintaining the meaning, or semantics, of features in a feature model –so-called validity maintenance– has been addressed in this paper. Validity maintenance is an essential aspect in feature modeling. Without it, feature modeling is nothing more than advanced geometric modeling, only offering parametric and constraint-based modeling facilities in addition to the normal geometric modeling facilities.

The approach to validity maintenance presented here has been developed within the semantic feature modeling approach, which provides a powerful and well-defined scheme for constraint-based specification of validity conditions in feature classes.

The most salient characteristic of semantic feature modeling is that the semantics of all features, once specified, is maintained during the whole modeling process. This is done by maintaining all constraints throughout model editing. A validity recovery mechanism analyzes any invalid situation that results from some modeling operation, and gives the user explanations and hints to overcome this. The user gets thus valuable assistance in creating valid models only, containing features with well-defined semantics only.

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

Acknowledgments

Rafael Bidarra’s work has been supported by the Praxis XXI Program of the Portuguese Foundation for Scientific and Technological Research (FCT).

We thank Alex Noort for valuable comments on earlier versions of the manuscript.

References