Proceedings of DETC'98
1998 ASME Design Engineering Technical Conferences
September 13-16, 1998, Atlanta, Georgia

DETC98/CIE-5705

DECLARATIVE USER-DEFINED FEATURE CLASSES

Rafael Bidarra, Abdelfettah Idri, Alex Noort and Willem F. Bronsvoort
Faculty of Information Technology and Systems
Delft University of Technology
Zuidplantsoen 4, NL-2628 BZ Delft, The Netherlands
Email: (Bidarra/Noort/Bronsvoort)@cs.tudelft.nl

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

(Shah, J.J. and Mäntylä, M., 1995
Parametric and Feature-based CAD/CAM; Concepts, Techniques and Applications,
John Wiley & Sons, p. 265)

ABSTRACT

Designing mechanical parts using a feature vocabulary is a very effective and rich paradigm. Its expressive power, however, is severely limited if the set of feature types available in a *feature library* is fixed. It is, therefore, desirable to be able to extend and configure a feature library according to particular requirements, either of an end-user of a CAD system or of an application area. These requirements are not limited to topologic and parametric aspects of a generic feature definition, but include also validity conditions to be verified for each feature instance in a model.

This paper proposes a new declarative scheme for the definition of *feature classes*. This scheme provides a unified description of the shape and validity issues of a feature class, as well as a flexible configuration of the feature class interface. In the definition process, the various constraint classes available play a central role, whereas an inheritance mechanism structures the feature library hierarchy. At the end of the process, validation of the class is performed, in order to avoid over- and underconstrained specifications. A graphical user interface supports the whole feature class definition process. Once defined, a feature class is automatically made available for use in a feature library of the modeling system.

1. INTRODUCTION

Many current feature-based modeling systems, both research prototypes and commercial systems, provide an attractive interface through which the user may create a part model with a convenient feature vocabulary. This ability, however, is often hampered by a number of shortcomings:

- sometimes, features only occur at the user interface level of the system, whereas the internal geometric model only records the geometry resulting from the operations performed. Such systems are in fact only enhanced geometric modeling systems, and should not be considered as full-fledged feature modelers;
- when a feature library provides only a fixed set of feature types for use in a model, the creation of complex shapes, often associated with some desired functionality and/or technological process, may become a rather unnatural task. Even if this can be achieved by composing several feature instances, the resulting composed shape can only be edited, queried and downstream processed in terms of the elementary instances, because there is no explicit interface defined for the "compound" (e.g. its dimension parameters). In addition, properties and validity conditions that were conveniently and meaningfully embedded in each of these elementary features, are of little use for the "compound" shape generated, if not completely undesirable (think, for example, of a cylindrical blind hole class, requiring the bottom face of the shape to be fully present on the part model boundary: one would not be allowed to compose two such instances in order to obtain a stepped-hole-like shape);
- some systems provide a mechanism to record a sequence of modeling steps, possibly in a parametrized way. In this procedural scheme, such *macros* might later be replayed,

in order to create a given “compound feature” in the model. This suffers from the same drawback just described: it is hard to consider such a library of macros as a real feature library, because it does not offer appropriate validity specification mechanisms;

- on the other hand, it is common experience that using pre-defined feature libraries with a very large set of feature classes is not the solution either, because an exhaustive enumeration of all possible feature classes is both unfeasible and unmanageable. Furthermore, such sets would vary significantly with the application domain, e.g. the functional requirements of the designer or the technological production processes available.

In order to overcome these drawbacks, declarative schemes, which separate generic feature specifications from feature model validation mechanisms, are receiving increasing attention. In this section, we first survey results of this research that deal with mechanisms for specification of user-defined features (UDFs).

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

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

Salomons et al. (1994) focused on interactive definition of new features during incremental modeling of a part, and on their representation by conceptual graphs. They propose combining profile sketching with geometric constraint graph editing, and manual feature identification on the solid model, in order to assist the user in the definition and insertion of new features into the model. A surface representation is used for both features and the part model. Feature models are stored in a hybrid scheme, using a database (for modeler independent data) and modeler files (for geometry-related information), which favours their intended feature recognition applications. Again, feature validity aspects of such UDFs are not considered, and their storage as feature classes in a feature library, for later use, is not mentioned.

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

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

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

This paper presents a new declarative scheme for the specification of feature classes, which not only overcomes these drawbacks, but fulfills all the requirements quoted at the beginning. It has been implemented in the Feature Library Manager of the Spiff[1] modeling system, a prototype multiple-view feature-based modeler developed at Delft University of Technology. The remainder of the paper is organized as follows. First, an overview of the proposed scheme is given, building on a sound understanding of feature semantics (Section 2). This is further elaborated, distinguishing the feature shape specification issues (Section 3), the validity specification aspects of the class (Section 4), and the feature class interface presented to the user (Section 5). Next, the main functional aspects of the Feature Library Manager are described (Section 6), and its use is illus-

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

trated with an application example (Section 7). Finally, we draw some conclusions (Section 8).

2. WHAT IS A USER-DEFINED FEATURE CLASS ?

Within the scope of this research, the focus is on *form features*, henceforth referred to as *features*. They are defined as representations of shape aspects of a physical product that are mappable to a generic shape and have functional significance. In other words, each feature has a well-defined *meaning* or *semantics*, expressed through its geometric, topologic, parametric and functional properties. A *feature class* is a structured description of all these properties, defining a template for all instances of a given feature type. As a whole, such properties specify the *validity conditions* that those feature instances should satisfy. Feature classes are sometimes also referred to as *generic feature definitions.*

Feature classes are grouped and stored in *feature libraries*, each of which is suited for a particular application. Ideally, feature libraries should be configurable by domain experts, i.e. people without special programming skills, but with knowledge of the requirements, the technology and the criteria of a given application domain. For this, they need a feature library configuration system, providing feature class specification and management facilities. On the other hand, designers, in their modeling activity, are *users* of the feature classes available in the feature library. For this, they use a CAD system, which performs modeling operations (e.g. create feature instances of selected classes) and takes care of the validity maintenance of the feature model. In this paper, we concentrate on the specification of feature classes and, thus, on a feature library configuration system such as mentioned above. Issues on validity maintenance of feature models were dealt with elsewhere, see for example (Dohmen et al. 1996), and are here, therefore, left out.

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

- *attach constraints* specify how a feature instance is attached to the model, by coupling some of its faces to faces of other features already present in the model;
- *geometric constraints* specify geometric relations, such as parallelism and distance, between feature faces;
- *dimension constraints* specify the set of values allowed for a feature parameter;
- *algebraic constraints* specify an expression for feature parameters;
- *semantic constraints* 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;
- *interaction constraints* specify whether a given interaction type should be disallowed for a feature instance.

With the scheme proposed here, all feature classes in a library use the same vocabulary, and exhibit a similar structure; see Figure 1. Every feature class can be edited, refined and customized according to specific requirements. This turns out to be the same as saying, with Shah and Mäntylä (1995), that UDFs “become full-privileged members of the feature collection of the system”.

The main characteristics of our approach are:

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

The specification schemes for the shape, the validity criteria and the interface of a feature class are elaborated in the following sections.

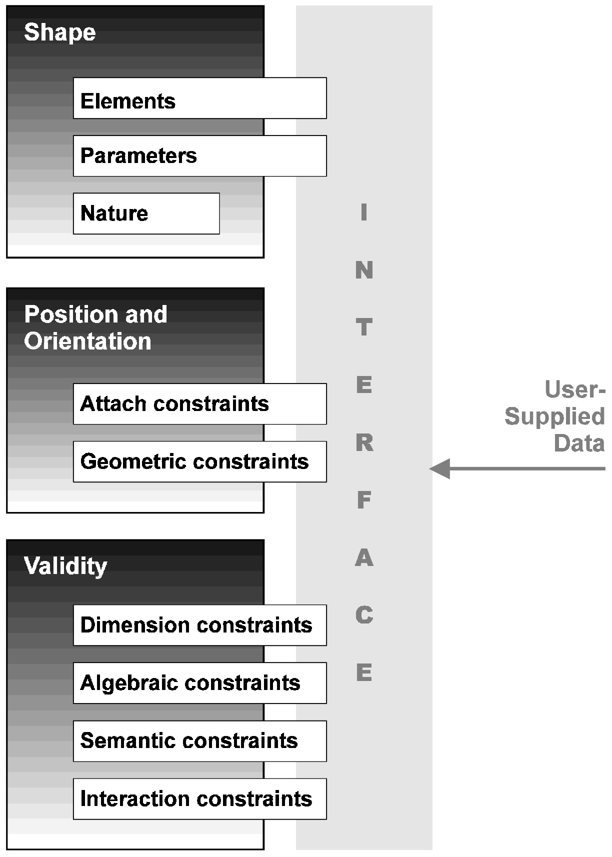
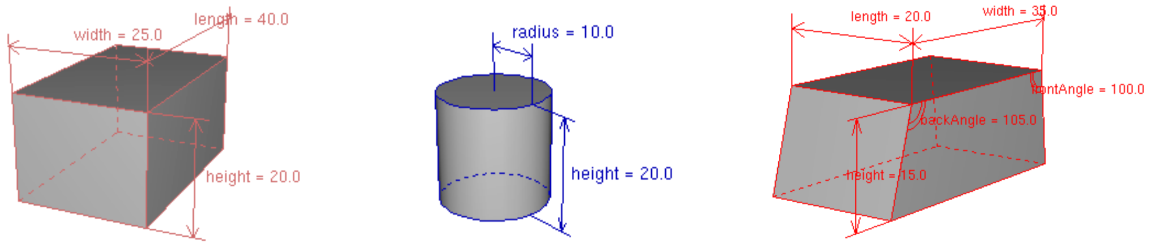


Figure 1 - Feature class structure



block **cylinder** **trapezoidal block**

Figure 2 - Parametrized basic shapes

3. FEATURE SHAPE SPECIFICATION

All constraint types presented above operate either on parameters or on faces of a feature shape. Therefore, the specification of the parametrized shape is the first necessary step in the creation of a new feature class.

We use parametrized *basic shapes*. A basic shape encapsulates a set of geometric constraints that relate its parameters to the corresponding shape elements, see (Dohmen et al. 1996) for details. Currently, three basic shape classes are available in SPIFF, a rectangular block, a cylinder, and a trapezoidal block; see Figure 2.

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

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

Shape inheritance

Shape inheritance is used if the desired feature shape matches exactly that of one of the available basic shapes. In this case, the feature class is made to inherit directly from that basic shape, thus acquiring all its attributes (e.g. parameters and faces) and functionality (e.g. initialization and query methods).

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

Shape composition

If the desired feature shape is not directly available from any of the basic shapes, shape composition is used. For this, several basic shape instances, possibly overlapping, are specified and related, in order to achieve the desired feature shape. With this constructive sheme, a large domain of shapes is achieved, not limited to that of linearly swept profile-based shapes, as in many feature modeling systems. Figure 3 shows examples of features whose shapes were obtained by combining instances of block and cylinder basic shapes.

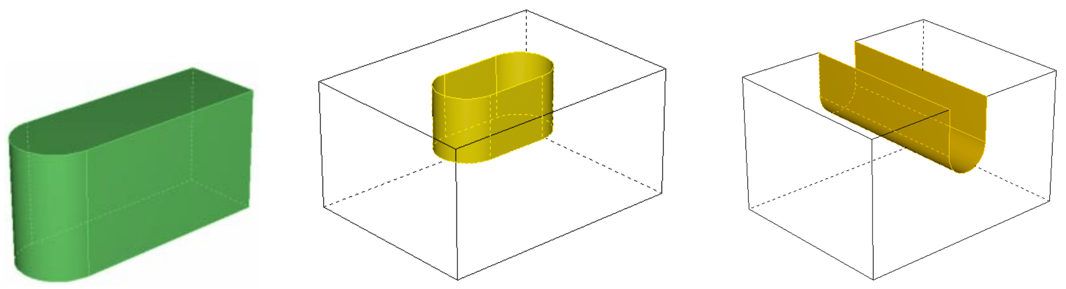
In order to achieve this increased shape complexity, basic shapes are encapsulated inside the feature shape, as depicted in Figure 4.

The shape composition process consists of four steps:

1) Selection of a number of basic shapes. For a *stepped blind hole* class, see Figure 5, two cylinder shapes may be chosen, say *cylinder1* and *cylinder2*.

2) Relative positioning and orientation of these shapes, applying geometric constraints among their faces. For the *stepped blind hole* example, this can be achieved with two constraints:

```
coaxial(cylinder1.top, cylinder2.top)
coplanar(cylinder1.bottom, cylinder2.top)
```



keyshaft **rounded pocket** **rounded through slot**

Figure 3 - Shape composition examples

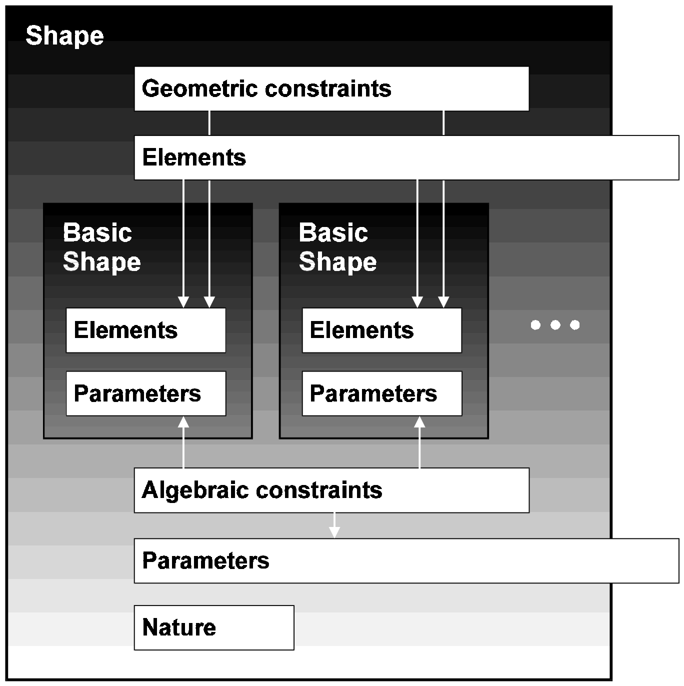


Figure 4 - Internal structure of shape composition

3) Specification of the compound shape parameters, as a function of the elementary parameters of the basic shapes. This is achieved by means of algebraic constraints, and should produce a set of parameters that fully determines the dimensions of all basic shapes. In the case of the *stepped blind hole* class, such a set could be:

```
TotalDepth = cylinder1.height+cylinder2.height
EntranceDepth = cylinder1.height
EntranceDiameter = 2*cylinder1.radius
BodyDiameter = 2*cylinder2.radius
```

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

```
EntranceTop = {cylinder1.top}
EntranceSide = {cylinder1.side}
EntranceBottom = {cylinder1.bottom}
BodySide = {cylinder2.side}
BodyBottom = {cylinder2.bottom}
```

As shown in Figure 4, both geometric and algebraic constraints, as well as the list of basic shapes, are only internally used in the shape composition process, in order to produce the desired feature shape. Therefore, the resulting shape has the same interface as in Figure 1, presenting a set of shape elements, a set of parameters and a nature attribute. These characteristics will be further elaborated in Section 5.

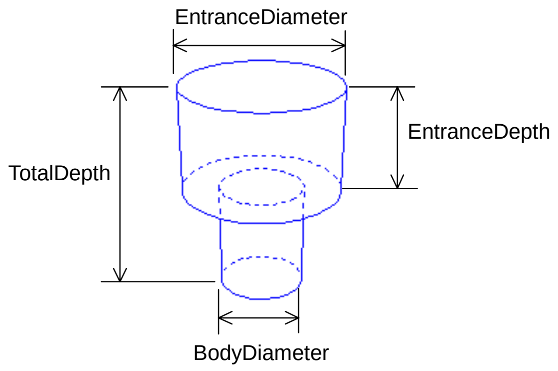


Figure 5 - Compound shape for a stepped blind hole class

4. FEATURE VALIDITY SPECIFICATION

Once the specification of the feature class geometry has been completed, it is essential to describe under which conditions the feature instances of that class can be considered valid. Specification of feature validity conditions is indispensable to perform validity maintenance later, during the modeling process. Without this, features would be no more than high-level geometric modeling primitives.

Criteria for specifying validity of features can, for example, take into account requirements of a technological and functional character, often dependent on the specific application area. Therefore, it is advantageous if each of those criteria can be specified independently and in a flexible way.

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

Geometric validity specification

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

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

Topologic validity specification

The specification of a feature shape, as described in Section 3, yields a set of shape faces that provide full coverage of the boundary of a volumetric feature. However, for most features, not all faces are meant to effectively contribute to the boundary of the modeled part, but some faces, instead, have a closure role, delimiting the feature volume without contributing to the model boundary. The specification of these properties is called topologic validity specification.

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

Functional validity specification

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

Such functional requirements can be violated by feature interactions caused during incremental editing of the model. *Feature interactions* are modifications of the shape aspects represented by a feature that affect its functional meaning; see (Bidarra and Bronsvoort 1996) for a taxonomy.

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

5. FEATURE CLASS INTERFACE SPECIFICATION

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

On the other hand, several feature constraints and parameters —the so-called *feature interface parameters*— may require some user-supplied data to be provided at feature instantiation stage, as depicted in Figure 1. These components constitute the *feature class interface*. The specification of the feature class interface determines how each of its instances will be presented to the user of the modeling system and, thus, how the user will be able to interact with it.

Essential in the feature class interface is the positioning and orientation scheme, which is specified by means of attach and geometric constraints, as depicted in Figure 1.

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

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

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

Feature validation constraints may also take part in the interface of a feature class, when some of their parameters are left unspecified until the creation of a specific feature instance. Examples of this are user-supplied data aimed at (i) initializing the bounds of an interval of a dimension constraint, (ii) determining whether an *onBoundary* semantic constraint should be configured as *partly* or *completely*, or (iii) setting the constant proportionality factor between two shape parameters in an algebraic constraint.

In the SPIFF system, all interface parameters have a name, which will be used later to designate them at the user interface, when the feature is instantiated or modified. Furthermore, each interface parameter can be configured as *fixed* or *editable*, meaning that its value can only be set at instantiation stage, or that it can also be modified later, when editing the feature.

Finally, it is possible to declare some interface parameters as *switchable*. This means that, whenever desired during the modeling session, they can be deactivated, and therefore not taken into account in the validation process. While this possibility is not likely to be used for most feature validity constraints, it is convenient for shape parameters and for

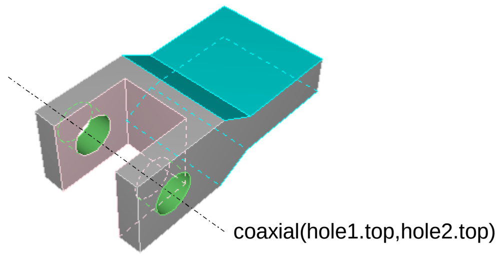


Figure 6 - Aligning features by switching off positioning geometric constraints

positioning geometric constraints. In this way, they may be overruled by additional geometric constraints explicitly created later between their features. As an example of this, consider the fixture part in Figure 6. Although the two through holes may have been positioned independently, each one using a different pair of reference model faces (e.g. the block front and top faces), it might be desirable to keep them aligned. This can be achieved by switching off the positioning geometric constraints of one of them, and replacing these by a coaxial constraint between the two entrance faces of the two through holes.

6. THE FEATURE LIBRARY MANAGER

The Feature Library Manager comprises a Feature Class Manager and a Graphical User Interface (Idri 1998), as depicted in Figure 7. The Feature Class Manager in turn comprises a Parser and a Generator to process Class Models. Through the Graphical User Interface, the user may create a new feature class, or modify an existing one.

When a new feature class is created from the outset, the Feature Class Manager incrementally builds the Class Model, as the user follows the steps outlined in Sections 3 to 5. Alternatively, existing feature classes can be used as a basis for the new class. Two cases are then distinguished.

In the first case, the new feature class inherits from a single feature class. Therefore, the new class gets the same shape and validity conditions as its parent. Its specification can then be completed by defining additional validity conditions and a new interface (see Sections 4 and 5). An example of this is the creation of a square passage class based on a rectangular passage class, by just adding an algebraic constraint that specifies its length and width parameters to be equal.

In the second case, the new feature class is composed from multiple existing feature classes. Here, the positioning constraints of the component classes are used to compose the shape of the new class, instead of explicitly building it with additional constraints, as when the class is defined from the outset. An advantage of using other feature classes to compose (the shape of) a new class is that both components with additive and components with subtractive natures can be used. Further specification steps are similar to those used for building classes from the outset, as outlined in Sections 4 and 5.

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

After a feature class has been fully specified, it is stored in the feature library, by means of the Generator. This translates the class model into a class in LOOKS, an object-oriented imperative programming language (Peeters 1993). SPIFF contains a LOOKS interpreter, into which feature libraries are loaded and, thus, made available in the modeling system. Eventually, the Feature Library Manager validates a feature class just specified, in order to avoid over- and underconstrained specifications. This is done by creating a prototype feature instance, and checking it with the solver of the modeling system (Noort 1997).

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

7. APPLICATION EXAMPLE

This section describes the use of the Feature Library Manager for the specification of a *rounded blind slot* feature class; see Figure 8.

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

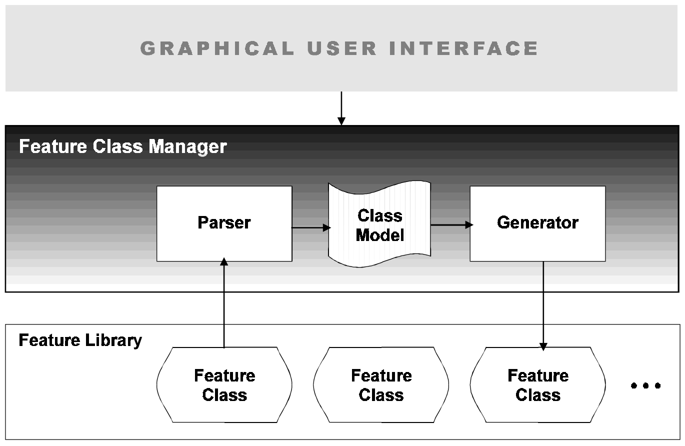


Figure 7 - Architecture of the Feature Library Manager

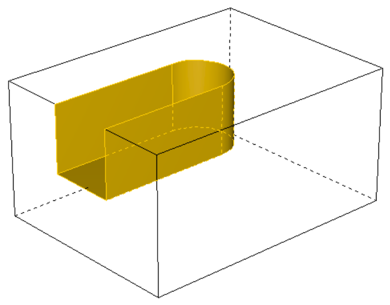


Figure 8 - Example of a rounded blind slot feature instance

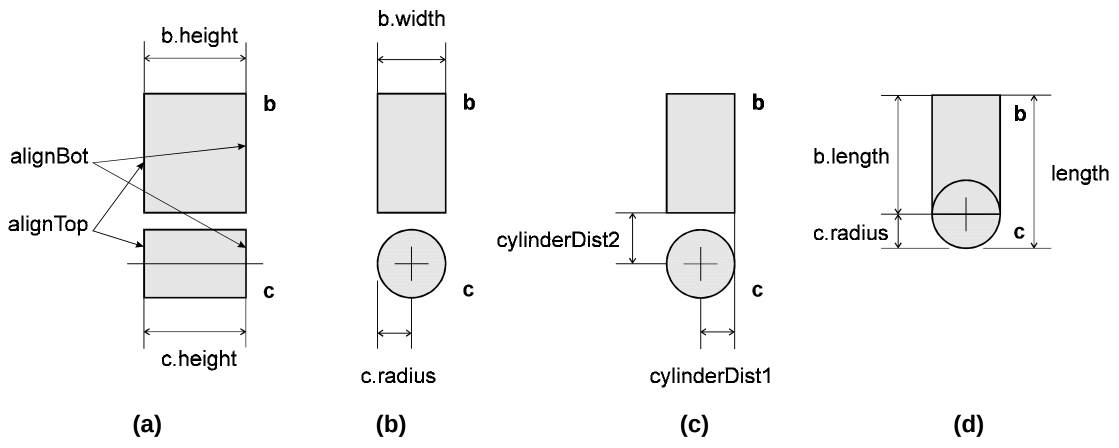


Figure 9 - Shape specification for the rounded blind slot class

1) Align the two shapes vertically and give them the same height, by making their top and bottom faces coplanar, with *coplanar* geometric constraints; see Figure 9 (a):
```
alignTop(b.top, c.top)
alignBot(b.bottom, c.bottom)
```
2) Make the diameter of the cylinder equal to the width of the block, with an algebraic constraint; see Figure 9 (b):
```
b.width = 2 * c.radius
```
3) Position the cylinder at the middle of the back face of the block, with two *distPointFace* geometric constraints; see Figure 9 (c):
```
cylinderDist1(c.top, b.right, c.radius)
cylinderDist2(c.top, b.back, 0)
```

The *rounded blind slot* interface parameters are then defined in terms of the basic shapes' parameters, with the algebraic constraints; see Figure 9 (d):
```
length = b.length + c.radius
width = b.width
depth = b.height
```

Analogously, the *rounded blind slot* interface faces are defined in terms of the basic shapes' faces as follows:
```
top = {b.top, c.top}
bottom = {b.bottom, c.bottom}
front = {b.front}
back = {c.side}
left = {b.left}
right = {b.right}
```

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

Finally, the position becomes fully specified with a *distFace-Face* geometric constraint setting the user-supplied distance between one of the *rounded blind slot* side faces and a user-supplied reference face:
```
position(left, modelFace3, distance)
```

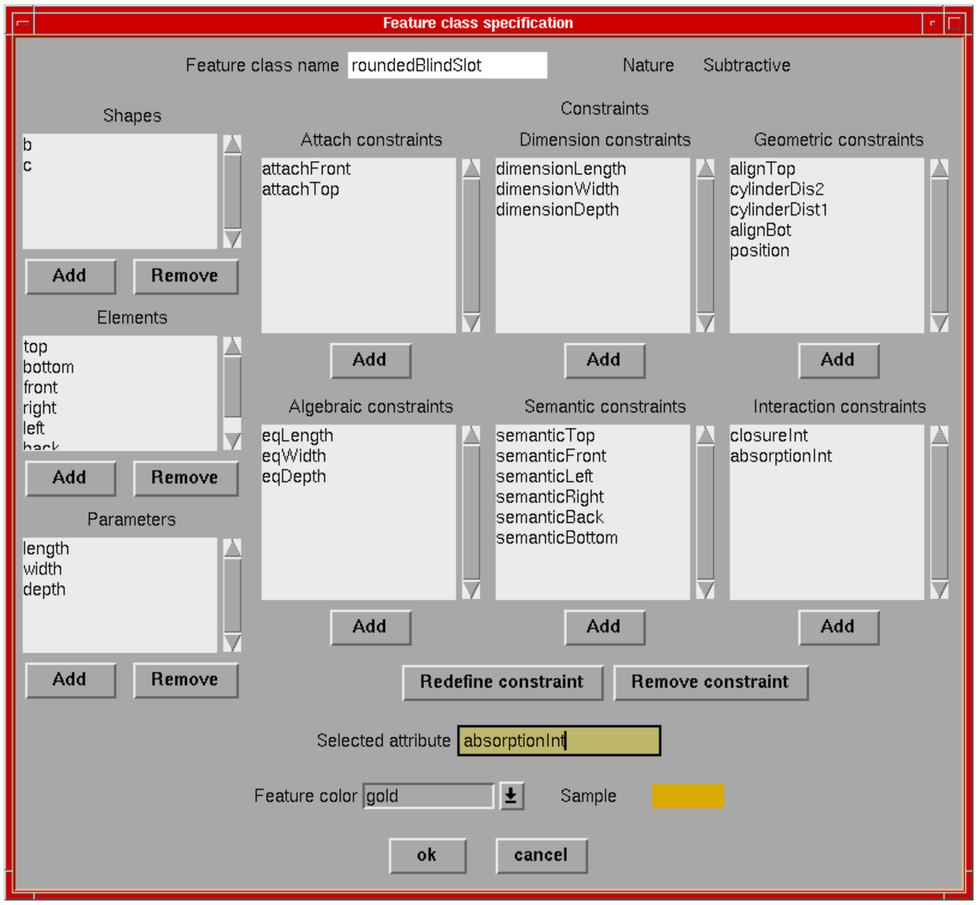
Several validity conditions can now be specified for the *rounded blind slot* feature class, e.g.:

1) To preserve the desired feature shape properties, the block should be prevented from degenerating (that would be the case if the length of the *rounded blind slot* was smaller than the cylinder radius); similarly, the *rounded blind slot* width and depth should be restricted to positive values. This is achieved by setting lower bounds for these parameters, with dimension constraints:
```
dimensionLength(length, c.radius)
dimensionWidth(width, 0)
dimensionDepth(depth, 0)
```
2) Faces top and front of the *rounded blind slot* should not be on the model boundary, whereas each of the remaining faces should have some contribution to the model boundary. For this, the following semantic constraints can be used:
```
notOnBoundary(top, completely)
notOnBoundary(front, completely)
onBoundary(left, partly)
onBoundary(right, partly)
onBoundary(back, partly)
onBoundary(bottom, partly)
```
3) Closure and absorption interactions could be disallowed for *rounded blind slot* instances, by including the respective interaction constraints in the class specification.

The final specification of the *rounded blind slot* feature class is shown in Figure 10.

8. CONCLUSIONS

Current schemes for defining new feature classes are very limited, mainly due to either excessive low-level input requirements, or incomplete specification of feature validity.



Feature class name: roundedBlindSlot
Nature: Subtractive

Shapes: b, c
Elements: top, bottom, front, right, left
Parameters: length, width, depth

Attach constraints: attachFront, attachTop
Dimension constraints: dimensionLength, dimensionWidth, dimensionDepth
Geometric constraints: alignTop, cylinderDis2, cylinderDist1, alignBot, position
Algebraic constraints: eqLength, eqWidth, eqDepth
Semantic constraints: semanticTop, semanticFront, semanticLeft, semanticRight, semanticBack, semanticBottom
Interaction constraints: closureInt, absorptionInt

Selected attribute: absorptionInt
Feature color: gold

Figure 10 - Final specification for the rounded blind slot class

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

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

ACKNOWLEDGMENTS

Rafael Bidarra's work is supported by the Praxis XXI Program of the Portuguese Foundation for Scientific and Technological Research (FCT).
Thanks to the Department of Mathematics and Computer Science of Eindhoven University of Technology for providing the LOOKS interpreter software.
Thanks to Maurice Dohmen and Klaas Jan de Kraker for implementing the LOOKS support for simple feature classes.

REFERENCES

Bidarra, R. and Bronsvoort, W.F. (1996) "Towards classification and automatic detection of feature interactions". In: Roller, D., editor, *Proceedings of the 29th International Symposium on Automotive Technology and Automation*, Automotive Automation, Croydon, England, pp. 99-108.

Bidarra, R., Dohmen, M. and Bronsvoort, W.F. (1997) "Automatic detection of interactions in feature models". In: *CD-ROM Proceedings of the ASME 1997 Computers in Engineering Conference*, ASME, New York.

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

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

Dohmen, M., de Kraker, K.J. and Bronsvoort, W.F. (1996) “Feature validation in a multiple-view modeling system”. In: McCarthy, J.M., editor, *CD-ROM Proceedings of the ASME 1996 Computers in Engineering Conference*, ASME, New York.

Idri, A. (1998) “User-Defined Object-Oriented Features”. Master’s Thesis, Delft University of Technology, The Netherlands.

de Kraker, K.J., Dohmen, M. and Bronsvoort, W.F. (1995) “Multiple-way feature conversion to support concurrent engineering”. In: Hoffmann, C. and Rossignac, J., editors, *Proceedings of the Third Symposium on Solid Modeling and Applications*, ACM Press, New York, pp. 105-114.

Hoffmann, C. and Joan-Arinyo, R. (1998) “On user-defined features”. To be published in: *Computer-Aided Design*.

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

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

Salomons, O.W., Slooten, F. van, Jonker, H.G., van Houten, F.J.A.M. and Kals, H.J.J. (1994) “Interactive feature definition”. In: Soenen, R. and Olling, G., editors, *Proceedings IFIP WG 5.3 International Conference on Feature Modelling and Recognition in Advanced CAD/CAM Systems*, Vol. 1, pp. 181-200.

Shah, J.J., Rogers, M.T., Sreevalsan, P., Hsiao, D. and Mathew, A. (1990) “The ASU Features Testbed: an overview”. In: *Proceedings of the ASME 1990 Computers in Engineering Conference*, ASME, New York, Vol. 1, pp. 233-241.

Shah, J.J., Ali, A. and Rogers, M.T. (1994) “Investigation of declarative feature modeling”. In: Ishii, K., Bannister, K. and Crawford, R., editors, *Proceedings of the ASME 1994 Computers in Engineering Conference*, ASME, New York, Vol. 1, pp. 1-11.

Shah, J.J. and Mäntylä, M., (1995), “Parametric and Feature-based CAD/CAM; Concepts, Techniques and Applications”, John Wiley & Sons, New York.