

DETC97/CIE-4275

AUTOMATIC DETECTION OF INTERACTIONS IN FEATURE MODELS

Rafael Bidarra, Maurice Dohmen and Willem F. Bronsvooort

Faculty of Technical Mathematics and Informatics
Delft University of Technology

Zuidplantsoen 4, NL-2628 BZ Delft, The Netherlands
Email: (Bidarra/Dohmen/Bronsvooort)@cs.tudelft.nl

ABSTRACT

Current feature-based modeling systems fail to adequately maintain feature semantics. This is partly due to inappropriate specification of validity conditions in feature classes, but mainly due to a lack of effective validity maintenance mechanisms throughout the modeling process. A critical aspect in this is feature interaction management. This paper presents a new approach to the detection of feature interactions, which uses semantic and interaction constraints in feature class specification. Validity maintenance is automatically performed after each modeling operation by checking these constraints, thus being able to detect a variety of interaction types. Such interactions are then analyzed, and their causes identified and reported to the user.

1 INTRODUCTION

Feature modeling systems offer the possibility to build a product model with features. These are representations of shape aspects of a physical product that are mappable to a generic shape and are functionally significant. Stated differently, each feature has a well-defined meaning, which should be represented and preserved in a product model.

Several proposals have been made to approach the problems of specification and maintenance of feature validity. These were surveyed in (Dohmen et al. 96), and it was concluded that a declarative approach, where validity specification is done separately from validity maintenance, provides the best solution. Currently, no feature modeling system provides a really powerful and self-contained scheme for validity specification of feature classes. Most approaches are based on a variety of constraint types, each of which gives a specific contribution in the description of the behavior desired for instances of each class in the feature library. Whenever a feature is instantiated, instances of its validation constraints are automatically created. Such constraints have to

be solved and maintained, both at a feature's creation and in subsequent modeling steps.

Without effective validity maintenance, or in case the validation constraints specified are insufficient, the modeler will fail in preserving feature semantics and, thus, in capturing designer intent. This is the case in many commercial "feature-based" systems which, for example, fail to notify the transmutation of a blind hole into a through hole, as depicted in Figure 1.

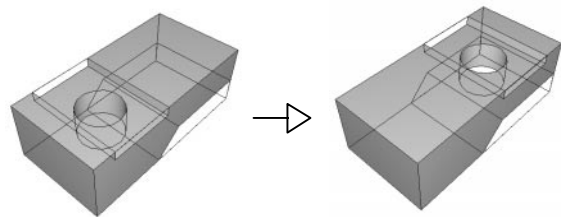


Figure 1 - Blind hole transmutation due to slot displacement

Current research prototypes that do perform validity maintenance, see for example (de Kraker et al. 95), (Mandorli et al. 95) and (Vieira 95), simply reject any user modeling operation that yields inconsistencies in the feature model. The user then has to take some alternative action, e.g. changing parameters of a feature, or even taking a feature of a different type.

However, this scheme seems too rigid, as it may often be hard to trace why invalidity arose or to find a way around it. The least that is desirable is that the user gets a good explanation on what has caused the invalidation of a feature. A further improvement would be that he gets hints on how to avoid or overcome the problem. Ideally, the system would automatically adapt the model to get a valid model again in cases this is possible and desirable, although this should

probably not be done without consulting the user. In the previous example of Figure 1, the blind hole might be automatically converted into a through hole, if the user permits this.

Most validity violations are caused by feature interactions, which arise from modeling operations such as the creation of a new feature or the modification of an existing feature. It is therefore important to manage feature interaction phenomena, so that all relevant interaction situations can be detected, reported and handled in an appropriate way (Regli and Pratt 96).

In short, a global solution to the validation problems pointed out so far should include:

- a) a *declarative* scheme for flexible specification of feature classes, allowing a fine tuning of the behavior desired for their instances;
- b) a *separate* validity maintenance mechanism;
- c) monitoring each modeler operation issued by the user in order to *detect feature interactions*;
- d) reporting to the user the *causes* of any invalidity situation, together with a detailed analysis of its consequences;
- e) providing the user with a reasonable choice of *reaction mechanisms* to overcome invalidity situations.

Items a) and b) have already been dealt with in (Dohmen et al. 96). This previous work will be summarized and somewhat extended here (Section 2). In this paper we will focus on item c), the *feature interaction detection mechanism* (Sections 3 and 4). This is regarded as an indispensable step in raising the level of assistance (required in items d) and e)) of a feature-based modeling system. Current implementation of these ideas within the SPIFF¹ modeling system, a prototype multiple-view feature-based modeler developed at Delft University of Technology, is also described. The operation of interaction detection algorithms is also illustrated with an example model (Section 5). Finally, some conclusions are drawn on the present approach, pointing out some further developments in this research (Section 6).

2 SPECIFICATION AND MAINTENANCE OF FEATURE VALIDITY

An effective proposal for specification and maintenance of feature validity in feature models has been presented in (Dohmen et al. 96), and implemented in the prototype system SPIFF. This system has a mechanism for feature validity maintenance based on constraint solving. Several types of validation constraints are available; here only a brief description of each one is given:

- *semantic constraints* specify how a feature instance is allowed to topologically deviate from its canonical behavior, by stating the extent to which its feature

elements (e.g. faces) should belong to the model boundary;

- *attach constraints* specify how a feature instance is attached to the model, by coupling some of its feature elements (i.e. faces or edges) to elements of other features already present in the model;
- *geometric constraints* specify geometric relations, such as parallelism and distance, between feature elements;
- *dimension constraints* specify an interval for the value of feature parameters;
- *algebraic constraints* specify an expression for feature parameters.

Such constraints are created in two ways. First, they may be embedded as attributes of a feature class - the *generic feature definition* - and are, thus, instantiated together with each new feature instance. Second, they may be explicitly added by the user, to further constrain or relate specific feature instances in the model. In either case, this is called *specification of validity conditions* (either of individual features or of the feature model as a whole). Among the constraint types presented above, dimension and semantic constraints are always established on a single feature instance (*intra-feature constraints*); attach constraints, on the other hand, always couple feature elements of two different features (*inter-feature constraints*); geometric and algebraic constraints may be established between feature elements or parameters either of the same feature or of different features (intra- or inter-feature constraints). Inter-feature constraints may be regarded as defining a *dependency relation* among feature instances of the model. Such a relation reflects which features are directly dependent on (the elements or parameter values of) any given feature.

The basic idea of our approach is that after a modeling operation has been performed, the model is required to conform to all existing constraints. The operation is unsuccessful, and thus rejected, if any of the constraints is violated. This is called *validity maintenance*.

Several advantages can be pointed out for this approach:

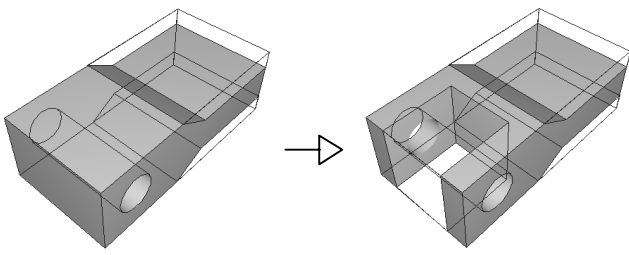
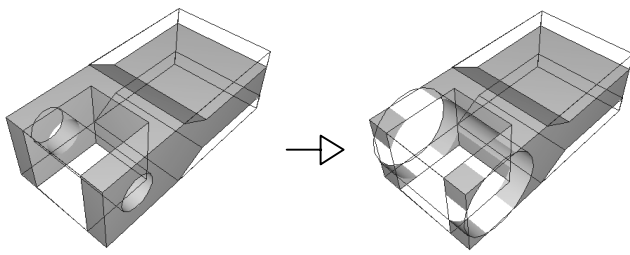
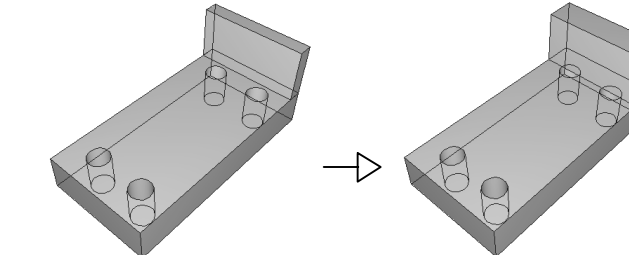
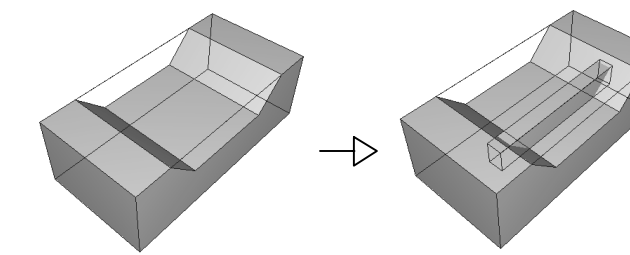
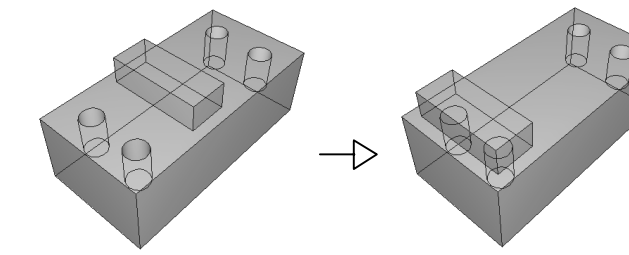
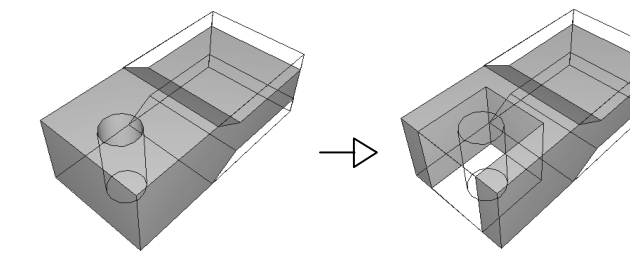
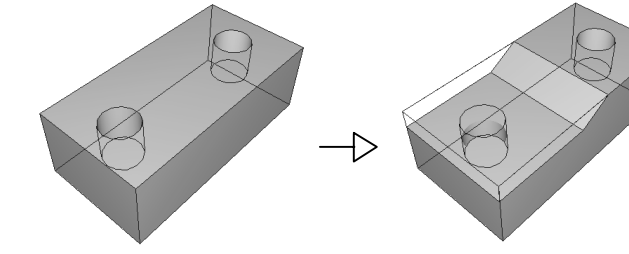
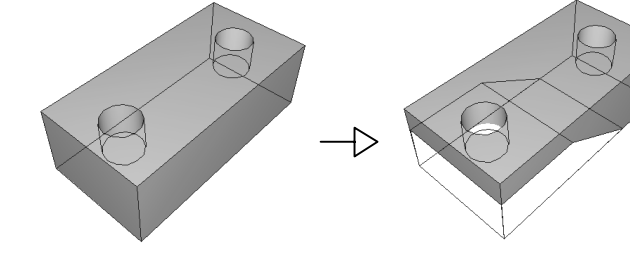
- the use of various constraint types for validity specification in generic feature definitions permits a more complete definition of all semantic aspects of each feature class;
- user-added constraints can further assist in capturing designer intent, still applying uniform constraint management;
- once specified, validity is *always maintained* throughout model editing, thus ensuring that all its feature instances are kept valid;
- separated validity maintenance, performed during incremental evolution of the model, allows for the application of various techniques, including an explanation mechanism for inconsistencies encountered in this process.

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

This approach has now been extended in three directions. First, interactions among features, caused during incremental editing of the model, are also taken into account. In particular, a variety of interaction classes that may affect feature semantics have been identified (Bidarra and Bronsvort 96). A summary of these interaction types is presented in Table 1.

Second, a new type of constraints, *interaction constraints*, is now used in feature classes to specify whether a given interaction type should be disallowed for its instances. Third, analysis of all invalid situations is performed in order to provide the user of the system with proper explanations of their causes.

Table 1 - Interaction classes handled in SPIFF

<p style="text-align: center;">SPLITTING INTERACTION</p>  <p style="text-align: center;">insertion of the slot splits the through hole boundary into disconnected components</p>	<p style="text-align: center;">DISCONNECTION INTERACTION</p>  <p style="text-align: center;">enlargement of the through hole diameter disconnects part of the block from the remaining of the model</p>
<p style="text-align: center;">BOUNDARY CLEARANCE INTERACTION</p>  <p style="text-align: center;">enlargement of the protrusion width obstructs entrance face of the through holes</p>	<p style="text-align: center;">VOLUME CLEARANCE INTERACTION</p>  <p style="text-align: center;">insertion of a protrusion intrudes into the subtractive volume of the V-slot</p>
<p style="text-align: center;">CLOSURE INTERACTION</p>  <p style="text-align: center;">displacement of the protrusion causes the whole volume of two blind holes to become a closed void inside the model</p>	<p style="text-align: center;">ABSORPTION INTERACTION</p>  <p style="text-align: center;">insertion of a slot suppresses contribution of the through hole to the model shape</p>
<p style="text-align: center;">GEOMETRIC INTERACTION</p>  <p style="text-align: center;">insertion of a V-step changes the depth of the blind hole</p>	<p style="text-align: center;">TRANSMUTATION INTERACTION</p>  <p style="text-align: center;">insertion of a V-step turns the blind hole into a through hole</p>

Validity maintenance is performed in SPIFF by means of a Constraint Manager, a Feature Geometry Manager and an Interaction Manager, under the control of a Feature Manager, according to the architecture depicted in Figure 2.

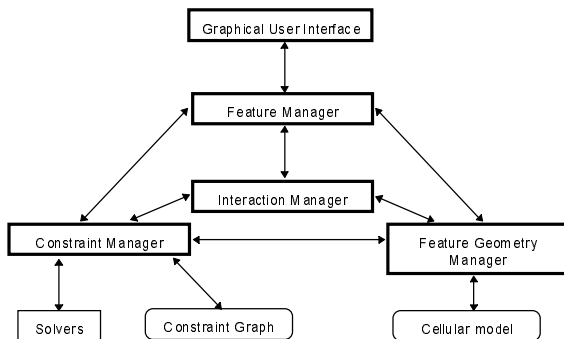


Figure 2 - Architecture of the SPIFF system

The Feature Manager receives commands from the user, issued via a graphical user interface, and sends appropriate requests to the respective Managers, after which the result of the operation is returned to the user.

The Constraint Manager maintains all constraints in a constraint graph, and solves them by calling dedicated solvers. The constraint graph is mapped onto two primitive constraint graphs, one for primitive algebraic constraints and another for primitive geometric constraints. The primitive algebraic constraint graph is solved using the SkyBlue approach (Sannella 92), the primitive geometric constraint is solved using the Degrees of Freedom analysis approach (Kramer 92). After a primitive graph has been solved, the constraint graph maintained by the Constraint Manager is updated. In this way, constraint solving is done efficiently by dedicated solvers, while the Constraint Manager takes care of the interdependence of the primitive constraint graphs.

The Feature Geometry Manager maintains the geometric representation of the feature model in a cellular model. It is responsible for performing those operations, issued by the Feature Manager, that modify the cellular model, for instance, adding a new feature to the model and removing or modifying an existing feature (de Kraker et al. 97a). Each feature is associated to one or more instances of *shape* classes. Each shape instance accounts for a bounded region of space - the *shape extent*. A through hole, for example, is associated to a cylinder shape. The cellular model represents a part as a connected set of volumetric quasi-disjoint *cells*, in such a way that each cell either lies entirely inside a shape extent or entirely outside it. Feature shapes are decomposed into cells; overlapping feature shapes share one or more cells. The complete boundary of a feature is decomposed into shape elements, which are also explicitly represented in terms of *cell faces* and *edges* (or simply *cell elements*). For the through hole example above its boundary is decomposed into the cylinder top, side and bottom faces, as well as the top and bottom loop

edges. Each cell element stores in an owner list which shape elements it belong to; analogously, each cell stores in an owner list which shape(s) it belongs to. In this way, the geometric representation of feature shapes and their elements can be selectively accessed at any time, allowing for the analysis of actual feature semantics, as described in (Bidarra et al. 97).

The Interaction Manager performs the last stage of the validation process, after each modeling operation: determining whether any feature interaction occurs, and taking appropriate action. For this purpose, the other two Managers are queried, according to the analysis required by the interaction detection mechanism described in the next section.

3 GLOBAL INTERACTION DETECTION MECHANISM

The global procedure of the Interaction Manager may, for each of the main modeling operations - *insertion*, *modification* and *removal* of a feature -, be subdivided into three main phases:

- determination of the interaction scope of each operation;
- detection of specific feature interactions arising from the operation;
- individual analysis of each interaction, which includes reporting its causes.

The *feature interaction scope* (FIS) of a modeling operation on a feature f is determined by identifying all feature instances in the model that may potentially be affected by it. Two important notions, with regard to a given feature f , are:

- the set of features that overlap with f , either volumetrically or between 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;
- 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 constraint graph the dependency relations on f .

Depending on the modeling operation, the feature interaction scope will consist of different combinations of overlapping and dependency sets.

Feature interactions taking place on any feature of FIS are detected by checking their interaction and semantic constraints. Detection algorithms for each type of interaction constraint are described in the next section.

Each constraint violation is recorded by the Interaction Manager. Eventually, the set of constraint violations is analyzed, in order to identify their causes, which are then reported to the user. Such explanations typically include references to the feature elements or parameters involved in the invalidity situation and possibly conflicting constraints (Noort 97).

Insertion of a new feature

After instantiation of a new feature, together with its validation constraints, the Constraint Manager invokes its dedicated solvers. First, algebraic constraints are solved in order to obtain values for unspecified parameters. Next, attach and geometric constraints are solved, in order to obtain all shape parameter values. Finally, the values obtained for parameters are checked against the dimension constraints. After this, the corresponding feature shape is instantiated and inserted into the cellular model by the Feature Geometry Manager.

The Interaction Manager then determines the FIS of the operation. At the insertion of a new feature f , there are no dependencies of other features on f yet, i.e. $DS(f)=\emptyset$, and thus $FIS=OS(f)$. Finally, interaction detection is performed on f and all features in FIS.

Modification of a feature

Modification of a feature involves changing any of its positioning or dimension parameters. After all changes to the feature have been specified by the user, the Constraint Manager re-solves the modified constraint graph of the model, and the Feature Geometry Manager updates the cellular model with the modified shape (or shapes).

The FIS of a modification operation on a feature f is the union of $OS(f)$, $DS(f)$ and the sets $OS(f_i)$ of each feature f_i in $DS(f)$.

The Interaction Manager performs the detection phase on feature f and on each feature in FIS. All interactions detected are reported and classified into three categories: (i) interactions on the modified feature f ; (ii) interactions on its dependent features, $DS(f)$; and (iii) interactions on any overlapping feature, either of f or of its dependent features. In this way, occasional interactions caused indirectly by any dependent feature are not only detected, but also properly reported. In the example of Figure 3, displacement of the upper V-slot implies the displacement of the attached slot, which in turn causes the transmutation of the blind hole.

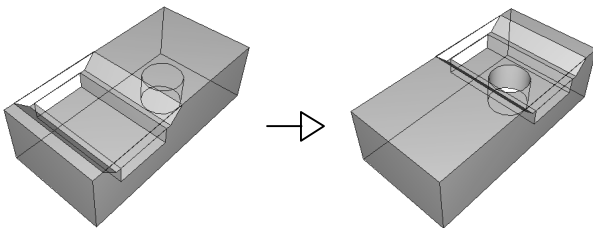


Figure 3 - Indirect interaction caused by a dependent feature

Removal of a feature

Removal of a feature f from the model is, from the interaction management viewpoint, similar to its modification. However, here the Interaction Manager has to make sure that none of the

inter-feature constraints, relating features of $DS(f)$ with f , is kept “pointing” to the removed feature. For this purpose, several alternative reactions may be devised, although most of these should not be performed without requiring user confirmation and/or input. Example reactions are, according to the particular type of inter-feature constraint:

- *attach constraints* - features attached to f may have their attachment moved to any other suitable feature of the model; alternatively, the user may choose to remove them also, together with f ;
- *geometric constraints* - such features must be made geometrically dependent on features that remain in the model; alternatively, the constraint might be removed, in cases their positioning does not become under-constrained;
- *algebraic constraints* - the algebraic expression should be changed so that it is made independent of any parameters of f ; alternatively, the constraint could be removed after fixing the values of all feature parameters it involves.

In cases where $DS(f)$ is empty, there is no further validation required on the remaining features, after f and its constraints have been removed from the model and the cellular model has been updated. Otherwise, the Constraint Manager first re-solves the modified constraint graph, after which the Feature Geometry Manager updates the cellular model accordingly. The Interaction Manager then checks interaction and semantic constraints of all features in FIS, just as for the modification operation.

4 DETECTION OF EACH INTERACTION CLASS

In this section, detection procedures are presented for the interaction classes presented in Table 1. For each of them, it is also pointed out how additional information is collected, in order to provide the user with a detailed explanation.

Each of these algorithms is aimed at checking the respective interaction constraint. For simplicity, the detection algorithms shown here operate on features with only one shape; however, their extension to features consisting of several shapes is straightforward. Only the detection algorithm for disconnection interactions operates on the whole model, provided that such interactions may take place without actually splitting any single shape, but rather disconnecting it from the remaining model volume, see (Bidarra and Bronsvort 96) for an example.

The algorithms shown make use of the functionality provided by the Constraint Manager and the Feature Geometry Manager in order to query their data. Most of these methods are described in detail in (Bidarra et al. 97); for completeness, Table 2 gives a summary of those methods.

Splitting interaction

Splitting interactions can be described in terms of the nature of

Table 2 - Summary of methods used in the detection algorithms

CELLULAR MODEL, <i>cm</i>	
cm.cells(nature)	returns the list of cells with specified nature in the cellular model
SHAPE, <i>s</i>	
s.nature	returns the nature (additive or subtractive) specified for shape <i>s</i>
s.elements	returns the list of shape elements of shape <i>s</i>
s.cells	returns the list of all cells that lie in the shape extent of <i>s</i>
s.boundary(nature)	returns the list of cell faces with specified nature that lie in the extent of shape elements of <i>s</i>
s.overlappingSet(nature)	returns the list of shapes of specified nature that overlap with shape <i>s</i> (either volumetrically or between their boundaries - cell faces and edges)
s.constraints(type)	returns the list of constraints of specified type established on shape <i>s</i>
SHAPE ELEMENT, <i>e</i>	
e.shape	returns the shape to which the element <i>e</i> belongs
e.cellFaces	returns the list of cell faces that lie in the extent of shape element <i>e</i>
CELL, <i>c</i>	
c.ownerlist	returns the list of shapes that own cell <i>c</i>
c.boundary	returns the list of cell faces that bound the volume of cell <i>c</i>
CELL FACE, <i>cf</i>	
cf.cell	returns the cell bounded by cell face <i>cf</i>
cf.partner	returns the partner cell face of <i>cf</i> that bounds an adjacent cell (if this exists)
cf.ownerlist	returns the list of shape elements that own cell face <i>cf</i>
cf.nature	returns additive if the cell face <i>cf</i> lies on the model boundary, and subtractive otherwise
OWNER LIST, <i>l</i>	
l.last	returns the last element of the owner list <i>l</i>
l.after(element₁, element₂)	returns true if element ₁ occurs after element ₂ in the owner list <i>l</i>

feature boundaries. They occur to a feature shape whenever the cellular decomposition of its boundary is such that the subset of its additive cell faces is not connected.

Splitting interaction detection algorithm

```
boundary ← s.boundary(additive)
cf1 ← boundary.first
for each cell face cf2 in boundary
    if not boundary.accessible(cf1, cf2)2
        return true
return false
```

additional data returned

- the split subsets of additive faces

Disconnection interaction

Disconnection interactions are analogous to splitting

² The accessible(*e*₁, *e*₂) method of a **set** of entities returns TRUE iff:
a) for the two specified elements, *e*₁ and *e*₂, either *e*₁=*e*₂ or *e*₁.adjacent(*e*₂) holds; or
b) there is a third element *e*₃ in the **set** such that *e*₁.adjacent(*e*₃) and **set**.accessible(*e*₃, *e*₂), and FALSE otherwise.

interactions, but they are better described in terms of the behavior of additive shape volumes. They occur to additive features whenever the cellular decomposition of the model is such that the subset of its additive cells is not connected.

Disconnection interaction detection algorithm

```
cells ← cm.cells(additive)
c1 ← cells.first
for each cell c2 in cells
    if not cells.accessible(c1, c2)
        return true
return false
```

additional data returned

- the split subsets of additive cells of the model

Boundary clearance interaction

Some semantic constraints, in particular those of type notOnBoundary(**completely**), are intended, for example, to guarantee clearance on toolpath entrance faces of subtractive features. A clearance interaction occurs to a subtractive feature whenever such a semantic constraint on one of its shape elements is not satisfied.

Boundary clearance interaction detection algorithm

```
semanticConstraints ← s.constraints(semantic)
for each sc in semanticConstraints
  if sc.type = nob(completely) and not sc.check
    return true
return false
```

additional data returned

- *the shape element with the unsatisfied semantic constraint*
- *the shape(s) causing the constraint violation*

Volume clearance interaction

A volume clearance interaction occurs to a subtractive feature whenever a subset of its volume is later occupied by an additive feature. The detection of this interaction relies on checking the ownerlist of all cells in the subtractive feature shape.

Volume clearance interaction detection algorithm

```
for each cell c in s.cells
  list ← c.ownerlist
  for each shape  $s_i$  in list
    if list.after( $s_i$ ,s) and  $s_i$ .nature = additive
      return true
return false
```

additional data returned

- *the additive feature shape causing the interaction*

Closure interaction

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

In the case of *single* closure, there is only one feature shape involved and, hence, a necessary and sufficient condition is that its whole shape boundary is totally present on the model boundary, i.e. it has no subtractive cell faces. In multiple closure, however, such cell faces may occur on the involved features' boundaries, but only separating their overlapping volumes. Therefore, the detection algorithm exits as soon as it finds one cell face of these boundaries that is not separating two subtractive cells.

Closure interaction detection algorithm

```
closedShapes ← s ∪ s.overlappingSet(subtractive)
for each shape  $s_i$  in closedShapes
  for each cell face cf in  $s_i$ .boundary(subtractive)
    if not exists cf.partner
      return false
    else
      closedShapes.add(cf.partner.ownerlist.last.shape)
return true
```

additional data returned

- *the set of closed feature shapes*

Absorption interaction

Absorption interactions are better described in volumetric rather than in boundary terms. They occur to either an additive or a subtractive feature, whenever it ceases to contribute to the model shape. A sufficient and necessary condition is that all cells of the absorbed feature shape are contained in, i.e. owned by, one or more other interacting shapes. This information is explicitly stored in the ownerlist of a cell, whose last element stands for the shape that most recently occupied the cell volume.

Absorption interaction detection algorithm

```
for each cell c in s.cells
  if c.ownerlist.last = s
    return false
return true
```

additional data returned

- *the set of interacting shapes causing the absorption*

Geometric interaction

Geometric interactions on a subtractive feature are described by a combination of volumetric and boundary conditions on shape elements. Informally, they can be described as the removal of a "slice" of the feature shape adjacent to one of its shape elements. The detection algorithm, thus, analyzes, for each shape element, the boundary of all cells in its neighborhood.

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

Geometric interaction detection algorithm

```
for each shape element e in s.elements
  geom_int ← true
  for each cell face cf in e.cellFaces
    for each cell face  $cf_i$  in cf.cell.boundary
      if  $cf_i$ .nature = additive
        geom_int ← false
        exit
    if not geom_int
      exit
  if geom_int
    return true
return false
```

additional data returned

- *the shape element(s) involved*
- *the actual parameter value(s)*

Transmutation interaction

Transmutation interactions are analogous to geometric interactions, in that they also act on a shape element. When a shape element e has a semantic constraint, its *semantic nature*, denoted $e.semanticNature$, is defined as:

- additive**, if it has a semantic constraint of type **onBoundary**;
- subtractive**, if it has a semantic constraint of type **notOnBoundary**; and
- nil**, otherwise.

With a transmutation, the nature of all cell faces of a shape element is opposite to its semantic nature. Shape elements on which there are no semantic constraints specified (meaning that their presence/absence on the model boundary is irrelevant for feature semantics) are, thus, not subject to this interaction class. To determine the potential new class of the transmuted feature, a dedicated module is used that performs incremental identification of features in the cellular model, see (de Kraker et al. 97b).

Transmutation interaction detection algorithm

```

for each shape element e in s.elements
  n ← e.semanticNature
  if n ≠ nil
    transm_int ← true
    for each cell face cf in e.cellFaces
      if cf.nature = n
        transm_int ← false
        exit
    if transm_int
      return true
return false

```

additional data returned

- *the shape element with the unsatisfied semantic constraint*
- *the identified feature class of the transmuted feature*

5 INTERACTION DETECTION EXAMPLES

In this section we illustrate with an example several classes of interactions that are detected by the algorithms presented above. We start up with the model in Figure 4, which consists

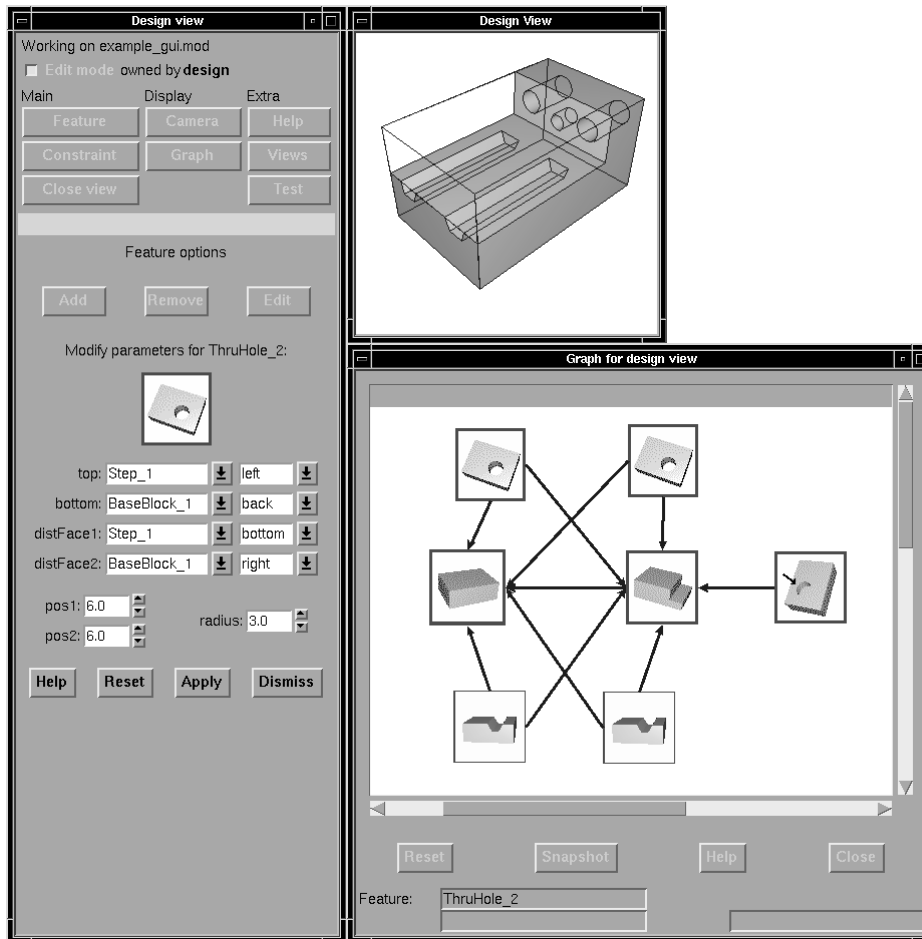


Figure 4 - Example feature model created in SPIFF

of a base block, a rectangular step, two blind slots, two through holes and a blind hole.

By means of the graphical user interface of SPIFF the above model can be edited, for instance by modifying one of its features or adding a new one. Figure 5 shows a variety of interaction situations which have been derived in this way. In (a) the blind hole boundary has no subtractive cell faces, and is thus closed inside the model; in (b) the through slot is inserted such that its additive boundary becomes split into three disconnected subsets; in (c) the same through slot is inserted with an excessively large depth, causing the base block disconnection; in (d) a decrease in the rectangular step depth

causes unclearance of the blind and through holes; in (e) the two blind slots see their effective length reduced due to the step insertion over their entrance faces; and in (f) one of the blind slots overlaps with the other when displaced, resulting in the geometry of a unique, larger blind slot.

It should be recalled that the actual detection of such interactions depends on the explicit presence of the respective interaction constraint in the affected feature instances.

6 CONCLUSIONS AND FUTURE WORK

Validity maintenance of feature models is not complete without proper management of feature interactions. This poses

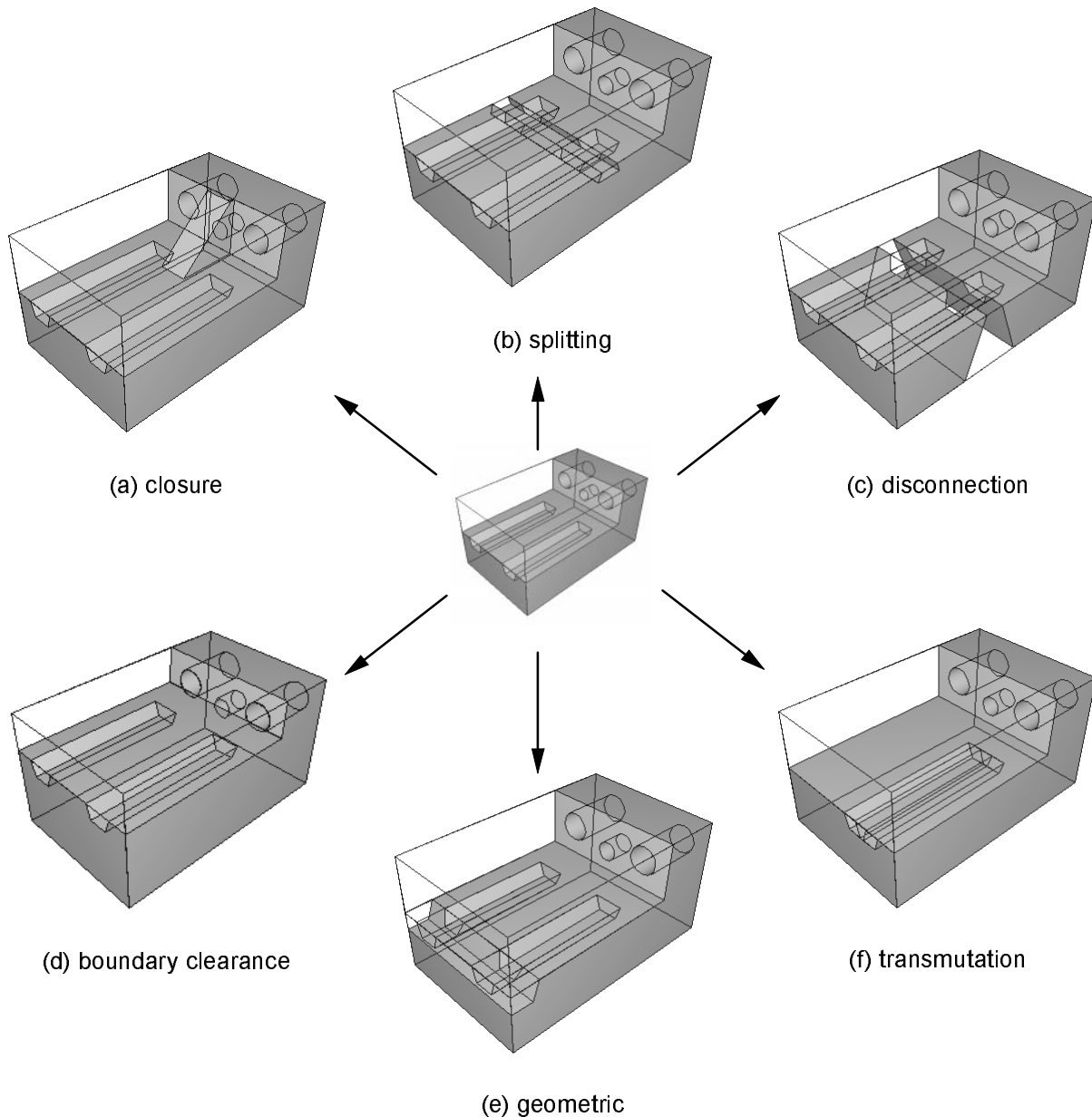


Figure 5 - Some interaction situations caused by editing the model of Figure 4

strong requirements at various levels of a feature-based modeling system, in particular at the:

- specification level of feature classes;
- geometric representation level of feature models;
- operational level of the modeler.

The approach described in this paper has the following advantages. It:

- permits fine tuning of validity specification (both at the generic feature definition level and at any modeling stage) for all feature instances in the model;
- ensures feature validity after each modeling operation;
- detects and classifies each kind of interaction occurring in the model.

Future research includes the generation of possible reaction methods to interactions detected, and the development of mechanisms for automatic recovery of model validity.

ACKNOWLEDGMENTS

Rafael Bidarra's work is supported by the Praxis XXI Program of the Portuguese Organization for Scientific and Technological Research (JNICT).

REFERENCES

- Bidarra, R. and Bronsvort, 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*, pp. 99-108.
- Bidarra, R., de Kraker, K.J. and Bronsvort, W.F. (1997) "Representation and management of feature information in a cellular model". Submitted for publication.
- Dohmen, M., de Kraker, K.J. and Bronsvort, W.F. (1996) "Feature validation in a multiple-view modeling system". In: McCarthy, J.M., editor, *CD-ROM Proceedings of ASME 1996 Computers in Engineering Conference*.
- de Kraker, K.J., Dohmen, M. and Bronsvort, W.F. (1995) "Multiple-way feature conversion to support concurrent engineering". In: Hoffmann, C. and Rossignac, J., editors, *Proceedings of the Third ACM/IEEE Symposium on Solid Modeling and Applications*, pp. 105-114.
- de Kraker, K.J., Dohmen, M. and Bronsvort, W.F. (1997a) "Multiple-way feature conversion - opening a view". In: Pratt, M., Sriram, R.D. and Wozny, M.J., editors, *Product Modeling for Computer Integrated Design and Manufacture*, Chapman & Hall, London, pp. 203-212.
- de Kraker, K.J., Dohmen, M. and Bronsvort, W.F. (1997b) "Maintaining multiple views in feature modeling". In: Hoffmann, C. and Bronsvort, W.F., editors, *Proceedings of the Fourth ACM/IEEE Symposium on Solid Modeling and Applications*, pp. 123-130.

- Kramer, G.A. (1992) "Solving geometric constraints systems: a case study in kinematics". The MIT Press, Cambridge, MA, USA.
- Mandorli, F., Cugini, U., Otto, H.E. and Kimura, F. (1995) "Reflective control of attributed entities in feature-based CAD systems using a CARW system manager"; In: Tomiyama, T., Mäntylä, M. and Finger, S., editors, *Preprints of the IFIP WG5.2 Workshop on Knowledge Intensive CAD-1*, Espoo, Finland, pp. 217-244, September 1995.
- Noort, A. (1997) "Solving over-constrained geometric models". Master's Thesis, Delft University of Technology, The Netherlands.
- Regli, B. and Pratt, M. (1996) "What are feature interactions?". In: McCarthy, J.M., editor, *CD-ROM Proceedings of ASME 1996 Computers in Engineering Conference*.
- Sannella, M. (1992) "The SkyBlue constraint solver", Technical Report 92-07-02, University of Washington, USA.
- Vieira, A.S. (1995) "Consistency management in feature-based parametric design". In: Gadh, R., editor, *Proceedings of the ASME 1995 Design Engineering Technical Conferences*, Vol. 2, Boston, MA, pp. 977-987.