

VALIDITY MAINTENANCE OF MODELS WITH INTERACTING FEATURES

Rafael Bidarra

Willem F. Bronsvort

Faculty of Information Technology and Systems
Delft University of Technology
Zuidplantsoen 4, NL-2628 BZ Delft, The Netherlands
Email: (Bidarra/Bronsvort)[@cs.tudelft.nl](mailto:)

Abstract

Current feature-based modelling 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 modelling process. An essential 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 modelling 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 modelling 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 modelling 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 modelling steps.

Without effective validity maintenance, or in case the validation constraints specified are insufficient, the modeller 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.

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 modelling 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.

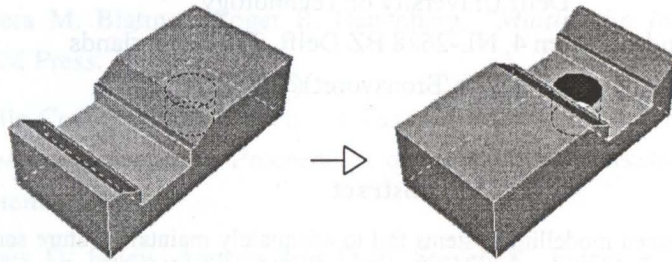


Figure 1 - Blind hole transmutation due to slot displacement

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 modelling 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). Within the scope of this research, we understand as feature interactions those modifications of the shape aspects represented by a feature that affect its functional meaning.

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 modeller operation issued by the user in order to *detect feature interactions*;

- d) reporting to the user the *causes* of any invalid situation, together with a detailed analysis of its consequences;
- e) providing the user with a reasonable choice of *reaction mechanisms* to overcome invalid situations.

The remainder of the paper is organized as follows. First, we give an overview of the current status of the feature validity specification and maintenance scheme (Section 2) and summarize how it is implemented within the SPIFF¹ modelling system, a prototype multiple-view feature-based modeller developed at Delft University of Technology (Section 3). Next, we focus on the interaction management issues of this approach (Section 4) and describe the interaction detection algorithms in detail (Section 5). We then illustrate their operation with an example model (Section 6). Finally, some conclusions are drawn on the present approach, pointing out some further developments in this research (Section 7).

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 (Bidarra et al. 97), 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:

- *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;
- *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 should belong to the model boundary;
- *interaction constraints* specify whether a given interaction type should be disallowed for a feature instance.

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

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

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).

The basic idea of our approach is that after a modelling 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. However, analysis of all constraint violations is performed in order to provide the user of the system with proper explanations of their causes. Such explanations typically include references to the feature elements or parameters involved in the invalidity situation, and possibly conflicting constraints (Noort et al. 97). 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.

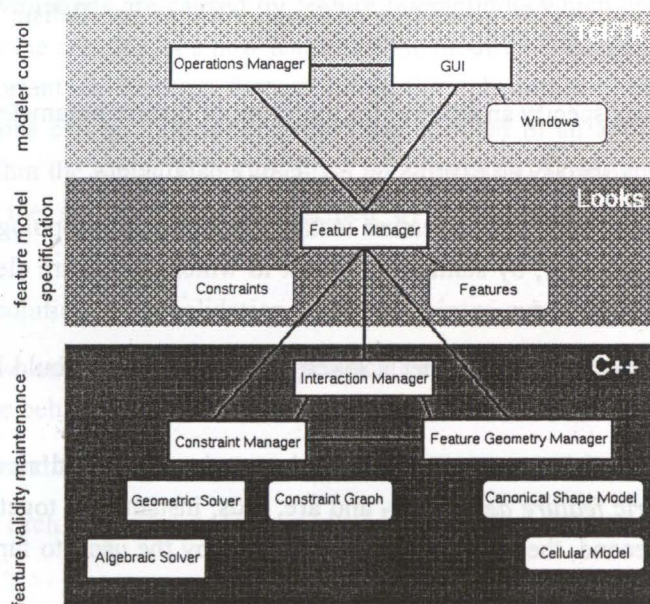


Figure 2 - Architecture of the SPIFF system

3. System Architecture

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.

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.

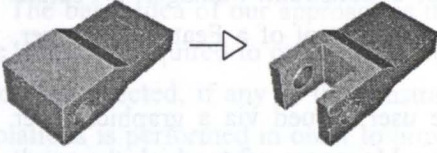
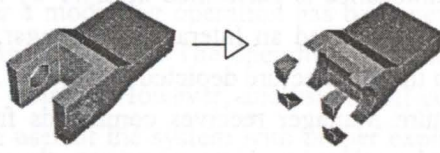
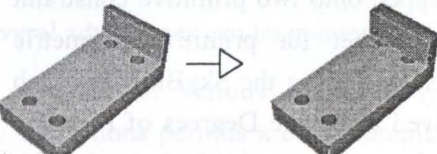
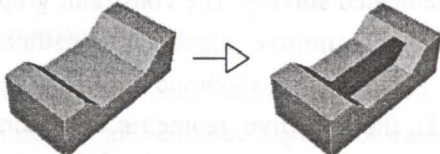
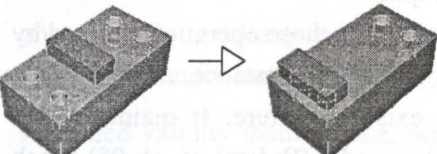
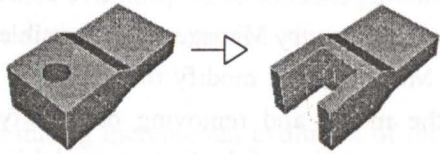
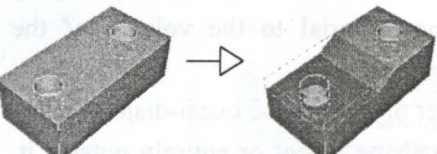
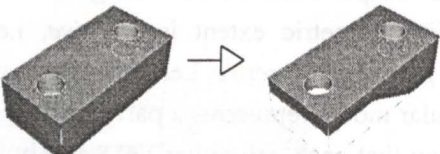
The Constraint Manager maintains all constraints in a constraint graph, and solves them by calling dedicated solvers. The constraint graph is mapped onto two primitive constraint graphs, one for primitive algebraic constraints and another for primitive geometric constraints. The primitive algebraic constraint 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, primitive constraint graph 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 is responsible for performing those operations, issued by the Feature Manager, that modify the geometry of the model, for instance, adding a new feature to the model and removing or modifying an existing feature. It maintains the geometric representation of the feature model in a cellular model (Bidarra et al. 98). 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. Features also assign to their shapes the *nature* attribute, specifying whether this volumetric extent is *additive*, i.e. adding material to the volume of the represented part, or *subtractive*, i.e. removing material from it.

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 functionally meaningful subsets, the *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 shapes 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 feature semantics.

In the next section we describe the functionality of the Interaction Manager.

Table 1 - Interaction classes handled in SPIFF

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

4. The interaction Manager

The Interaction Manager performs the last stage of the validation process, after each modelling operation: determining whether any disallowed feature interaction occurs, and taking appropriate action. A variety of interaction classes that may affect feature semantics has been identified (Bidarra and Bronsvort 96). A summary of these interaction types is presented in Table 1.

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

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

The *feature interaction scope* (FIS) of a modelling operation on a feature f is determined by identifying all feature instances in the model that may potentially be affected by it. For this, 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 modelling operation, the feature interaction scope will consist of different combinations of overlapping and dependency sets, see (Bidarra et al. 97) for a detailed description.

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

Each constraint violation is recorded by the Interaction Manager. Eventually, the set of constraint violations is analyzed, in order to identify their causes, which are then reported to the user.

5. 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.

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 the methods are described in detail in (Bidarra et al. 98); for completeness, a summary of them is given in Table 2.

Table 2 - Summary of methods used in the detection algorithms

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

5.1 Splitting interaction

Splitting interactions can be described in terms of the nature of feature boundaries. They occur to a feature shape whenever the cellular decomposition of its boundary is such that the subset of its 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)
        return true
return false
    
```

additional data returned

- the split subsets of additive faces

The `accessible(e1, e2)` method of a `set` of entities returns `true` iff (a) for the two specified elements, e₁ and e₂, either e₁=e₂ or e₁.adjacent(e₂) holds; or (b) there is a third element e₃ in the `set` such that e₁.adjacent(e₃) and `set.accessible(e3, e2)`.

5.2 Disconnection interaction

Disconnection interactions are analogous to splitting 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*

5.3 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*

5.4 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 owner list 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 si in list
    if list.after(si, s) and si.nature = additive
      return true
return false
```

additional data returned

- *the additive feature shape causing the interaction*

5.5 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 si in closedShapes
  for each cell face cf in si.boundary(subtractive)
    if not exists cf.partner
      return false
    else
      closedShapes.add(cf.partner.ownerlist.last.shape)
return true
```

additional data returned

- *the set of closed feature shapes*

5.6 Absorption interaction

Absorption interactions are again 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 owner list 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*

5.7 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 cfi in cf.cell.boundary
      if cfi.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)*

5.8 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 no semantic constraints are 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. 97).

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

6. 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 with the model in Figure 3, which consists of a base block with six subtractive features: one rectangular step, two blind slots, two through holes and one blind hole.

By means of the graphical user interface of SPIFF, this model can be edited, for instance by modifying one of its features or adding a new one. After each operation, model validation analysis signals all interaction constraint violations detected. The user can then inspect each of them, getting feedback on its causes and consequences; see Figure 4 for a variety of interaction situations which have been derived in this way. For each of them, we assume that the respective interaction constraint has been specified and is, thus, present in the affected feature instances.

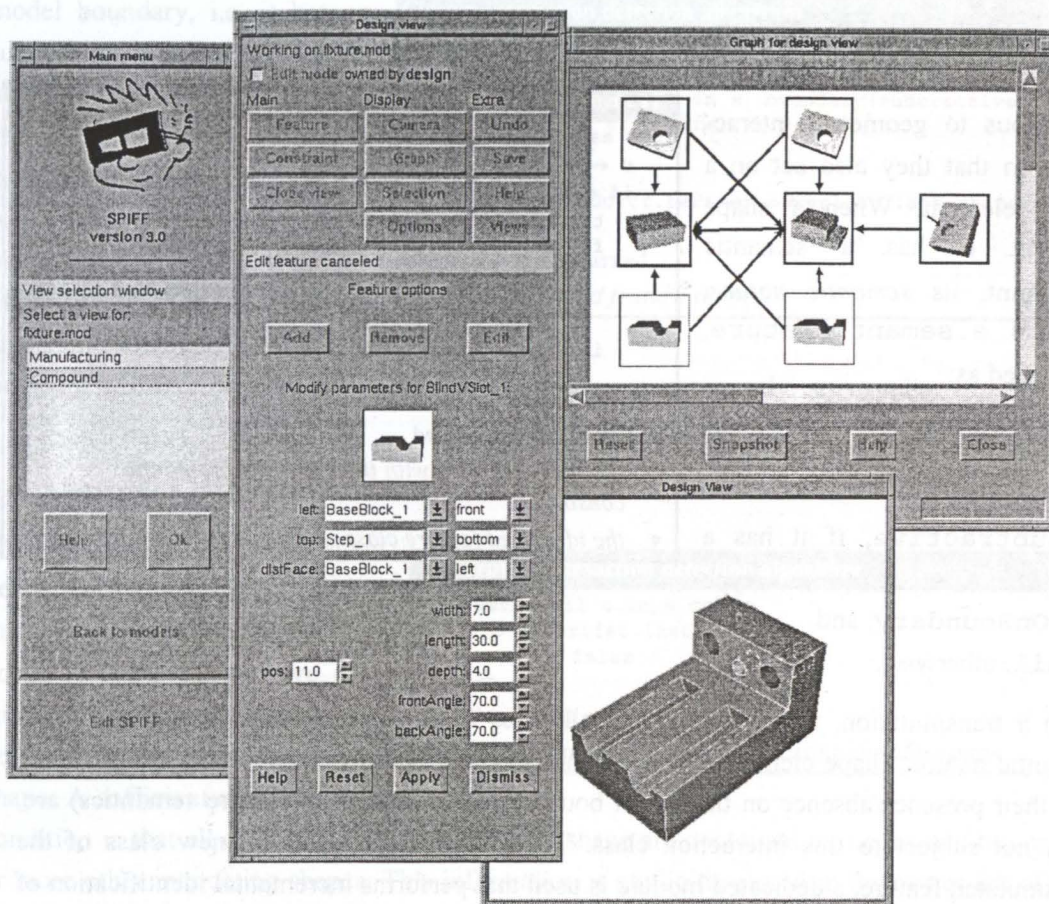
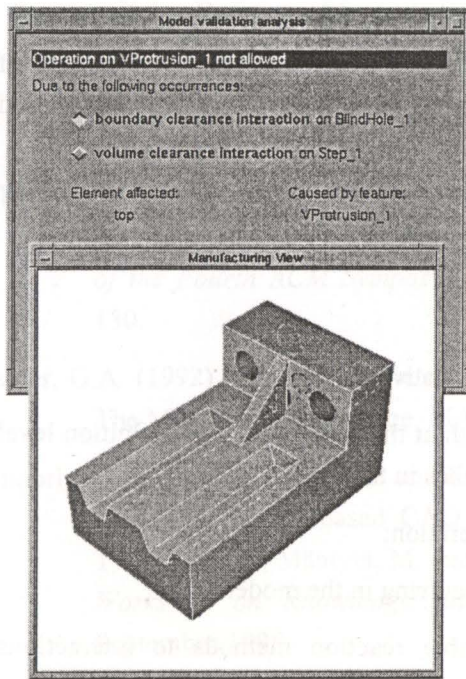


Figure 3 - Example feature model created in SPIFF

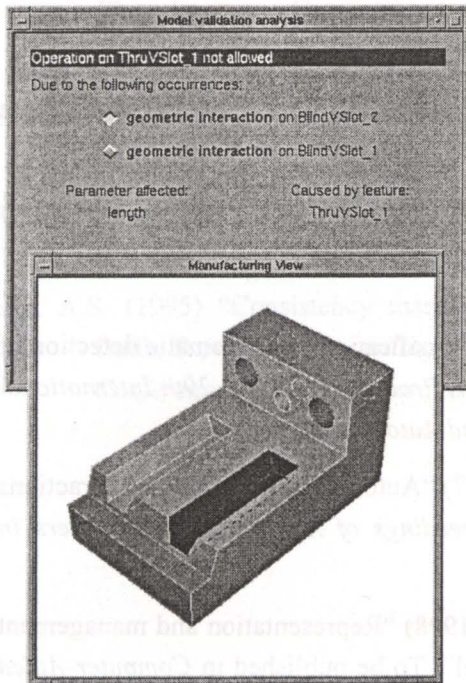
In Figure 4.(a) the insertion of a rib generates a clearance interaction on the blind hole. In Figure 4.(b) one of the through holes is displaced, so that its side face breaks across the block. In Figure 4.(c) the two blind slots see their effective length reduced due to the step insertion over their entrance faces. In Figure 4.(d) a through slot with an excessively large depth is inserted, causing the base block disconnection interaction.



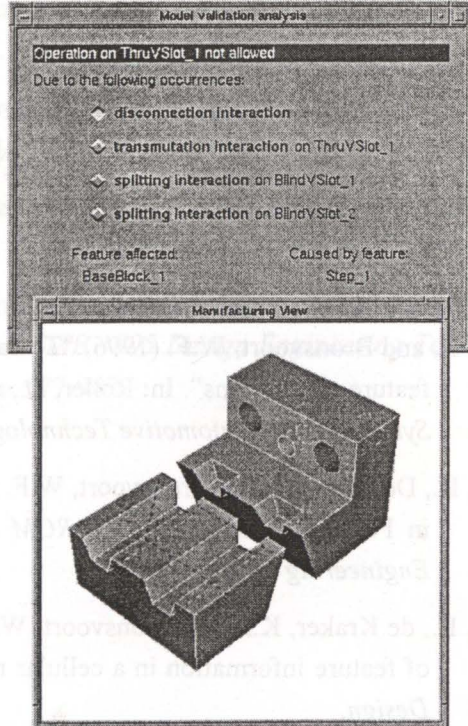
(a) boundary clearance



(b) semantic



(c) geometric



(d) disconnection

Figure 4 - Some interaction situations caused by editing the model of Figure 3

7. Conclusions and Future Work

Validity maintenance of feature models is not complete without proper management of feature interactions. This poses strong requirements at various levels of a feature-based modelling system, in particular at the:

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

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 modelling stage) for all feature instances in the model;
- ensures feature validity after each modelling 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 Foundation for Scientific and Technological Research (FCT).

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., Dohmen, M. and Bronsvort, W.F. (1997) "Automatic Detection of Interactions in Feature Models". In: *CD-ROM Proceedings of ASME 1997 Computers in Engineering Conference*.
- Bidarra, R., de Kraker, K.J. and Bronsvort, W.F. (1998) "Representation and management of feature information in a cellular model". To be published in *Computer-Aided Design*.
- 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. (1997) "Maintaining multiple views in feature modeling". In: Hoffmann, C. and Bronsvort, W.F., editors, *Proceedings of the Fourth ACM 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., Dohmen, M. and Bronsvort, W.F. (1997) "Solving over- and underconstrained geometric models". In: Brüderlin, B. And Roller, R., editors, *Proceedings of Workshop on Geometric Constraint Solving and Applications*, Ilmenau, Germany, pp. 8-22, September 1997.
- 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.