Local Topological Beautification of Reverse Engineered Models

C. H. Gao F. C. Langbein A. D. Marshall R. R. Martin

Department of Computer Science, Cardiff University, PO Box 916, 5 The Parade, Cardiff, CF24 3XF, UK

Abstract

Boundary representation models reconstructed from 3D range data suffer from various inaccuracies caused by noise in the data and by numerical errors in the model building software. The quality of such models can be improved in a *beautification* step, where geometric regularities need to be detected and imposed on the model, and defects requiring topological change need to be corrected. This paper considers changes to the topology such as the removal of short edges, small faces and sliver faces, filling of holes in the surface of the model (arising due to missing data), adjusting pinched faces, etc. A practical algorithm for detecting and correcting such problems is presented. Analysis of the algorithm and experimental results show that the algorithm is able to quickly provide the desired changes. Most of the time required for topological beautification is spent on adjusting the geometry to agree with the new topology.

Key words: Beautification; Healing; Topological Modification; Reverse Engineering; Geometric Modelling.

1 Introduction

Reverse engineering the shape of a 3D object is the process of reconstructing a geometric model of an object from measured data [18]. The general procedure consists of measuring surface points on an object, usually with a 3D laser scanner, merging multiple views into a single registered data set, segmenting the point set, fitting surfaces to each point subset, and stitching these into a solid model. Our goal is to create a system that, for simple engineering objects, reconstructs a boundary representation (B-rep) model from a physical object, with a minimum of human interaction. It should be usable both by naive users and engineers. In particular, the generated model should have all the intentional *geometric regularities* present in the original, ideal, design of the object, to ensure that the model has maximum utility for manufacturing, redesign, etc. The model should also have the expected *topology*—for example, if we reverse engineer a four-sided pyramid, we expect all four sloping faces to meet at a single vertex.

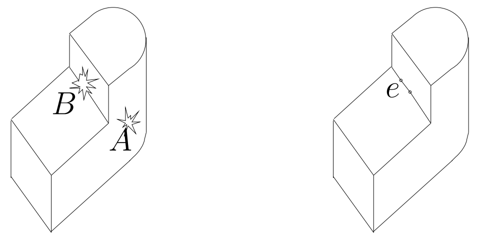
In reverse engineering, numerical errors occur in the reconstruction algorithms, and noise is present in the measured data. Improving the sensing techniques and the reconstruction methods can reduce errors, but some will always remain. Additional errors may be present due to wear of the object before scanning, and the particular manufacturing method used to make the object (e.g. if the object was cast in a mould, draught angles may have been added). Note that we wish to recover a geometric model of the *ideal* object as conceived by the designer. However, reverse engineering often fits each face individually, and treats it independently of the other faces in the model, losing regularities present in the original design. We propose to improve the reconstructed B-rep model by adjusting it in a separate *beautification* post-processing step. This paper in particular considers the problem of detecting and making any necessary *topological* (and consequent geometric) adjustments to the model; our earlier work considered *geometric* beautification without topological change [6,7,8,9,10,13,14].

For example, if a four-sided pyramid is reverse engineered, and each sloping face is fitted to data points independently, any three of these faces will intersect in a point, but it is extremely unlikely all four as fitted will pass through a single point (see Figure 28 later). Thus, the initial geometric model created will have either a very short edge or a very small face instead of a vertex at its apex, depending on exactly how the software produces a B-rep model by intersecting and stitching the individual faces. The topology of such a model needs to be adjusted, in conjunction with the geometry of the sloping faces, to produce a new model in which all four sloping faces pass through the same point.

1.1 The Topological Beautification Problem

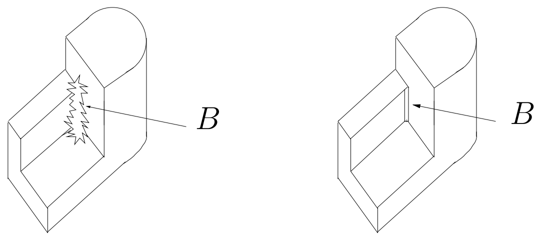
In this paper, we specifically address topological beautification of ‘conventional’ reverse engineered models bounded by planar, spherical, cylindrical, conical, and toroidal faces. However, the ideas presented are likely to be applicable to models containing free-form surfaces, too, even if we have not specifically considered such cases. In particular we consider a specific set of problems listed below, where beautification requires adjusting the topology of the model besides the geometry. In the following we refer to these as *topological problems*. Note that this does not mean that the initial topology is invalid, but rather that changes to the topology are needed to resolve such problems.

All such problems depend upon a notion of “small”, e.g. we intend to remove only small, spurious faces. The tolerances employed to decide about this are discussed in Section 3.1.



(a) Before modification (b) After modification

Figure 1. Repairing a face gap and an edge gap

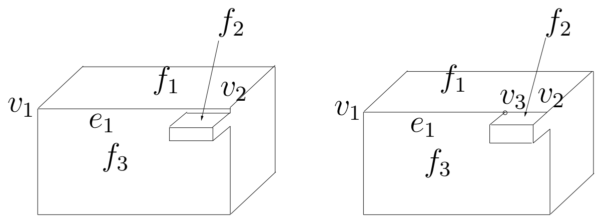


(a) Before modification (b) After modification

Figure 2. Repairing a complex multiple face gap

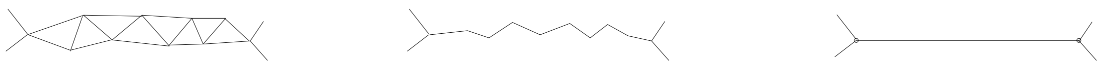
The problems (and their resolutions) listed below have been identified as the ones which can arise from earlier model building processes. They do not represent a list of all possible topological problems, but are problems which are likely to arise during reverse engineering of models. The detailed cases which have to be considered to remove such problems are described in Sections 3.2 and 4.

- **Removing gaps in a single face:** A loop of half-edges may exist in the interior of a face, with nothing on the other side of the loop. Such a case may arise, for example, where the scanner did not collect any data from within a deep concavity in the face. Here the loop of half-edges should be removed, extending the face (see hole A in Figure 1).
- **Removing gaps crossing an edge:** A loop of half-edges may span two faces, with nothing on the other side of the loop. The edge between the faces is divided into two pieces by the gap. The gap should be removed, the existing faces extended, and the two edge pieces joined (see hole B in Figure 1).
- **Removing gaps spanning multiple faces:** A loop of half-edges may span multiple faces, with nothing on the other side of the loop. Existing faces and edges must be extended to fill the gap, and new vertices and edges must be added as necessary (see gap B in Figure 2).
- **Adjusting pinched faces:** If a face narrows to a very thin part it is *pinched*. Other parts of the model should be adjusted to remove the thinning, resulting in a change in connectivity of the face; a face may be split into two faces (see Figure 3).
- **Removing chains of small faces:** Faces should meet in an edge, but instead a chain of small faces may separate them. The chain of small faces should be



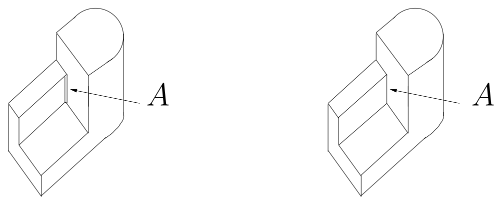
(a) Before modification (b) After modification

Figure 3. Repairing a *spiking* pinched face



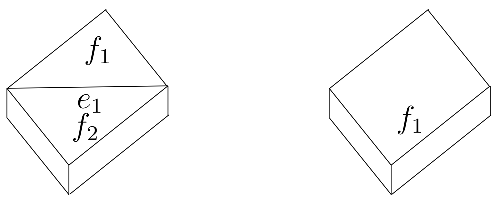
(a) A chain of small faces (b) becomes a chain of short edges (c) and is replaced by an edge

Figure 4. Removing a chain of small faces



(a) Before modification (b) After modification

Figure 5. Replacing a sliver face with an edge

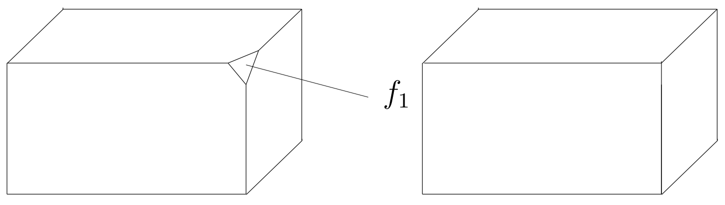


(a) Before modification (b) After modification

Figure 6. Merging adjacent faces with the same geometry

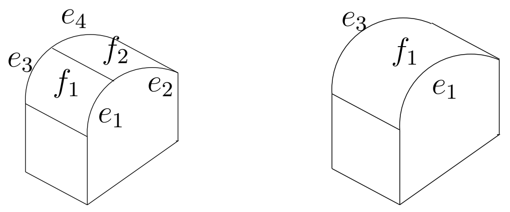
replaced by an edge (see Figure 4, where the first step is to reduce a chain of small faces to a chain of short edges).

- **Removing sliver faces:** Two faces should meet in an edge, but instead a long very thin face (a *sliver* face) may separate them. The sliver face should be replaced by an edge (see Figure 5).
- **Removing chains of short edges:** Several consecutive short edges may need to be replaced by a single long edge (again see Figure 4). This is in particular a problem which may result from repairing some of the other problems listed.
- **Merging adjacent faces with the same geometry:** Two adjacent faces may share the same geometry across a contiguous edge sequence. Edges and vertices as appropriate should be removed, and the faces merged (see Figure 6).
- **Removing isolated small faces:** Several edges should meet in a single vertex, but instead they meet at several distinct vertices, joined by multiple short edges



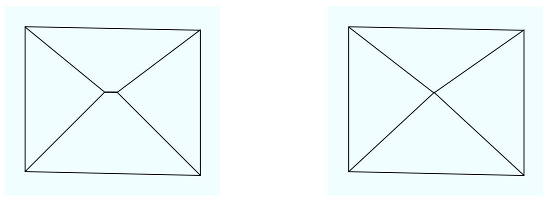
(a) Before modification (b) After modification

Figure 7. Removing a small face



(a) Before modification (b) After modification

Figure 8. Merging faces and edges with the same geometry



(a) Before modification (b) After modification

Figure 9. Removing a short edge

which surround a small face. The small face should be replaced by a vertex connected to the existing edges (see Figure 7).

- **Merging edges:** See Figure 8: once faces f_1 and f_2 have been merged, edges e_1 and e_2 can also be merged. Essentially this means that we have to merge adjacent edges with the same geometry. However, each edge should be the complete, connected intersection of two adjacent faces. Hence, we can simply merge each edge pair connected by a vertex which is attached to no more than two edges (taking care when handling special cases involving closed curves, where the modeller may require a vertex).
- **Removing isolated short edges:** Several edges should meet at a single vertex, but instead they meet at several distinct vertices, joined by one or more short edges. These short edges should be replaced by a single vertex (see Figure 9).

Beautification is a final step in producing a solid model from the scanned point set. We assume that a valid model has already been produced by prior model creation steps (although it may not have a closed shell, if gaps exist in the scanned point set). The first step of beautification consists of identifying and correcting the above topological problems. This process may involve the local addition or removal of faces, edges and vertices, and other modifications to the existing topological data-structure to ensure that a correct, valid model results—for example, edges may need to be disconnected from an existing vertex, and connected to a new vertex. In addition, constraints must be generated and imposed on the geometry supported by the topological elements to ensure that in the final model, the geometry and topology are consistent (e.g. to ensure that a given vertex lies on a given surface). This step can be combined with geometric beautification where desired regularities which only require adjustments to the geometry of the object are imposed on the model.

Detecting geometric regularities is discussed in previous work [8,9,10,13,14]. We describe in [6,7] how to select appropriate geometric regularities and impose them on the model using geometric constraints. The methods presented in this paper change the topology *prior* to these geometric beautification steps. One of the main sets of geometric constraints used for geometric beautification ensures that the model's geometry is consistent with its topology. As topological beautification updates the topology before geometric beautification, these constraints ensure that the geometry is consistent with the updated topology. It may be the case that a topology suggested by topological beautification cannot be realised geometrically. This will become obvious at the constraint solving stage and the reason for the inconsistency can be reported via the set of inconsistent constraints and the geometric objects involved. In such cases, either small faces, etc. can be reinserted to create a valid model, or it can be reported back to the topological beautification step in order to try a different topological structure. Note that as we are only considering small changes to the topology for typical reverse engineered models, such issues are unlikely to occur. In this paper we solely describe the methods to detect and adjust the above topological problems. In summary, the steps needed for topological beautification are:

- **Detecting topological problems:** small faces, sliver faces and short edges are identified; gaps in the model (in a single or multiple faces) arising from missing scan data are identified; etc.
- **Adjusting the topological structure:** isolated small faces and short edges are replaced by a single vertex, and surrounding topology is adjusted to meet it; existing faces are extended to cover gaps left by missing data, by removing the edges and loops bounding gaps; etc.
- **Constraining the geometry:** geometric constraints are generated to ensure that in the new structure, the faces, edges and vertices have the desired connectivity and contact. This usually also involves generating additional constraints for geometric beautification.
- **Regenerating the geometry:** the above geometric constraints are solved to produce the new geometry for the final model.

1.2 Healing

Topological beautification as outlined above has some similarities to, but also some differences from, CAD model healing [11,16,17]. Healing is a process that tries to correct inconsistencies and invalidities in B-rep models. A major application for healing arises when a model written out by one CAD system is to be read into another CAD system. If the recipient system works to tighter tolerances or uses different a representation from the sending system, a model which the sender considers to be valid may well be topologically inconsistent according to the recipient—e.g. edges may not lie on the surfaces which they bound, etc.

Healing, like topological beautification, aims to improve the topology of the model, but the differences are that (i) it starts with *invalid* models, not valid ones, and (ii) the main aim of the changes is to ensure a valid model is the result. For example, healing may have to cope with such problems as duplicated geometry, physically impossible geometry, incorrectly oriented surfaces, faces with no defined geometry, self-intersecting edges, edges which should meet in one vertex but which end at two topologically distinct but geometrically coincident vertices, end vertices which do not lie on the edges they bound, faces whose boundary is not a closed loop, and incomplete topology even though all individual faces are present. Note that such problems of validity are *not* expected to occur in beautification.

1.3 Outline

In the rest of the paper, we first review previous relevant work. We then specify in detail the topological changes considered by our algorithm and outline our algorithmic approach to topological beautification. In more detail, we then describe how we handle the particular problems listed above. The performance of our approach is then analysed, and finally, we provide the results of some experiments.

2 Previous Work

In our previous work on beautification, we have shown how to find and beautify approximate geometric regularities [7,8,9,10] and approximately congruent features [4] in initial B-rep models produced by reverse engineering; this prior work assumed that the models already had the desired topology. This work is also *directly* relevant here, because as well as modifying the topology, the existing geometry has to be constrained to fit the new topology. As already noted, we use the methods described in [7] to produce new geometry meeting these constraints.

Much work has been done on healing, although many of the methods are embedded in commercial systems, and have not been openly published. However, for example, one commercial system [2] uses an approach of (i) re-intersecting higher-dimensional geometry to get lower-dimensional geometry, (ii) adjusting the shapes and positions of higher-dimensional geometry to meet other lower-dimensional geometry, and (iii) an algorithm to orchestrate (i) and (ii) in the correct order. Deciding automatically which faces originally defined as blends may be useful information to guide the process. Converting simple faces to analytic surfaces instead of NURBS also helps.

In a similar vein, Park and Chung present a topology reconstruction algorithm which starts from a set of unorganised trimmed surfaces [15]. Essentially, any existing topology is discarded, and rebuilt. They note that the presence of undesirable elements like short edges, sliver faces, and so on can cause problems, and correct these as part of the rebuilding process; sometimes user-assistance is needed. Their approach is based on using vertices which are at the same location to within a small tolerance to deduce the topology. As the initial topology (if any) is discarded, their algorithms take more than linear time in model complexity, because geometric sorting must be done.

Butlin and Stops [1] were some of the first to discuss the healing problem, noting some of the problems in models to be "slivers, crossovers, minute edge lengths, stray points 'on the moon', . . . , patchworks of faces [with] unnecessary elements". FEGS CADfix software resolves such problems so that CAD models can be imported into finite element meshing packages. This software solves many of the problems noted under the description of healing above, as well as other issues described in their paper. They classify problems as "geometric sanity" problems, such as edges of a face not lying in that face, and "topological insanity", such as neighbouring faces not connected to the same edge. The approach taken by CADfix is to automatically detect problems whenever possible, but to leave it to the user to specify what particular steps should be taken to remedy them. Specific algorithms are not discussed.

Petersson and Chand have also developed tools for the preparation of CAD geometries for mesh generation [16], using a similar approach. Geometries are read from IGES files and then maintained in a boundary representation. Gross errors in the geometry are identified and removed automatically, while a user interface is provided for manipulating the geometry (such as correcting invalid trimming curves or removing unwanted details). Modifying the geometry by adding or deleting surfaces, and sectioning it by arbitrary planes (e.g. symmetry planes) is also supported. These tools are used to produce robust and accurate geometric models for initial mesh generation, but, as in CADfix's approach, they require some user assistance.

Mezentsev and Woehler [11] also consider mesh generation, and perform separate steps to first ensure a correct watertight model (i.e. having a topologically valid shell), and then to alter the model in ways which assist meshing algorithms. Again, they divide the former problem into geometric and topological errors, and suggest a cyclic process of verification, automatic repair and manual repair, until a satisfactory model results. The algorithms given are fairly straightforward, but the paper does contain a useful appendix showing some of the problems which need to be repaired.

Much work has also been done on simplifying and processing triangulations, and some of these ideas are relevant to the present problem. For example, Dey et al. present a method for preserving the topology of simplicial complexes while applying edge contractions [3]. Guskov and Wood present a topological noise removal method [5] which processes a triangular mesh and identifies features such as small tunnels. They then identify the non-separating cuts needed to cut and seal the mesh, thus reducing the genus and the topological complexity of the mesh. However, we do not expect such problems to be present in the B-rep models we are processing.

3 Principles of the Topological Beautification Algorithm

Our algorithm aims to improve the model by repairing local topological problems of the types listed in Section 1. After detecting topological problems and changing the topology, our overall beautification algorithm takes the corrected topology, generates geometric constraints from it, combines them with other constraints arising from geometric beautification, and solves the constraint system [7] to deduce the geometry for the new model. Topological modification is thus just one component of an *overall* beautification system, which also imposes geometric regularities found in the model.

In this section we discuss the tolerances needed to detect the topological problems in the model, and the sequence of operations required to change the topology of the model. We also present an outline of our algorithm, which we illustrate with an example.

3.1 Tolerances

Our algorithm starts with a topologically valid geometric model, which may have some holes (i.e. may not be a closed shell), but is otherwise correct. In order to decide whether and where the problems listed above are present, we need a tolerance indicating when a face is small or pinched, or when an edge is short, etc. In our algorithm we use a single tolerance for such purposes, which essentially separates intentional “features” of the model from artifacts generated by the reverse

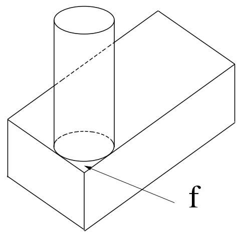


Figure 10. A necessary small face

engineering process.

For simplicity, we assume here that the tolerance is provided *either* by the user based on the size of errors expected in the model (for example, from a knowledge of the scanner and reverse engineering algorithm characteristics), *or* by some method which analyses the raw model. For instance, we can use methods like those for detecting approximate symmetries of point sets formed by the vertices in the model [12,13] to produce a transitive clustering in which all distances between points in different clusters are larger than distances between points in the same cluster. Characteristic lengths determined by the clustering can be used to deduce a suitable tolerance value. Similarly, we could use tolerance levels at which congruent features of the model are detected [4].

When choosing a tolerance, we should be careful that small but significant parts of the model are not deleted. For example, in Figure 10, face f may be small, but is necessary, and we should not attempt to remove it. Thus, the length tolerance should be larger than the size of any small face or short edge which is to be deleted, but smaller than any part of the model which is to be retained. For simplicity, we assume here that a single global tolerance value is used satisfying this requirement, but we note that if, for example, different regions of the object were scanned at different resolutions, or that large features exist for which the scanner could not capture high quality data, a more sophisticated approach with an adaptive tolerance might be needed. Nevertheless, at least in a local sense it still has to be possible to distinguish between spurious and intended features by a tolerance, otherwise automated decisions based on tolerances alone are not possible. Normally, we expect there to be a large size difference between the largest erroneous feature and the smallest intentional feature, so this is a reasonable limitation.

In the rest of the paper, we implicitly assume that in decisions like “a face is small”, or “edges have the same geometry”, the tolerance is used in an appropriate way.

3.2 Order of topological beautification operations

In a raw reverse engineered model, multiple topological problems of different types often coexist. To efficiently solve these problems, we need to detect and modify the problems in the right order. In particular, we wish to avoid having to use a loop which considers different types of problem repeatedly. This *can* be done if we repair some types of problem earlier than others. For example, closing a gap spanning multiple faces may result in the generation of a sliver face to fill the gap. But removing a sliver face cannot produce a gap. Thus, we detect and close gaps before removing sliver faces.

In the following we make the basic assumption that all the problems listed in Section 1.1 are local or isolated. This means that they can be corrected without having to consider the adjacent topological structure beyond the one specified (see below). For instance, when removing a small face we assume that we do not have any small faces adjacent to it which must be considered for removal of the small face. However, note that we consider chains of small faces at an earlier stage; for those chains we assume that there are no areas of connected small faces (e.g. a large surface made of small face patches) which would contain many ambiguous chains of small faces. This means that there are certain models which cannot be repaired by our methods. However, rather than considering all possible cases we present an algorithm which can repair common cases. For models with many non-local problems, the most appropriate course of action is probably to collect more accurate data (or improve the earlier reverse engineering stages).

Table 1 lists all the cases in which solving a topological problem of a given type may create a further topological problem of a different type. The columns are topological problems to be solved in order, and the rows are new problems which may arise from solving them, *assuming the ordering given*. In the following we discuss the topological operations for repairing each of the problems in turn, justifying the entries in this Table, by briefly listing the types of changes involved, and what sort of other problems may be introduced by the change. Details of the actual operations are also given in Section 4.

(1) **Face gaps.** A face gap is a sequence of edges connected by vertices where exactly two edges meet. Each of the edges is only on the boundary of one face. Removing this isolated sequence of edges cannot introduce any new face gaps. The resulting face (without gap) may be pinched, small or a sliver face, but the face with the gap already had this property, so it is not introduced by the topological change (it may now be simpler to detect, though). As no additional faces or edges are introduced, no other problems are created. Fixing face gaps also decreases the number of edges in the model (see A in Figure 1), speeding up subsequent processing.
(2) **Edge gaps.** An edge gap is a loop of connected edges where each vertex except for two lies on exactly two edges of the loop. Each of the edges is only on the boundary of one face and there are exactly two faces involved. The two special vertices lie on additional edges. An edge gap is removed by replacing the loop by a single edge between the two special vertices. This does not create new gaps (in particular no face gaps), nor introduce new problems relating

Modification order	1	2	3	4	5	6	7	8	9	10	11
Fixing → can cause ↓	Face gaps	Edge gaps	Multiface gaps	Pinched faces	Small face chains	Sliver faces	Short edge chains	Adjacent faces	Small faces	Merging edges	Short edges
Face gaps	No	No	No	No	No	No	No	No	No	No	No
Edge gaps	No	No	No	No	No	No	No	No	No	No	No
Multiface gaps	No	No	No	No	No	No	No	No	No	No	No
Pinched faces	No	No	Yes	No	No	No	No	No	No	No	No
Small face chains	No	Yes	Yes	Yes	No	No	No	No	No	No	No
Sliver faces	No	No	Yes	Yes	No	No	No	No	No	No	No
Short edge chains	No	Yes	Yes	Yes	Yes	Yes	No	No	No	No	No
Adjacent faces	No	Yes	Yes	Yes	Yes	Yes	Yes	No	No	No	No
Small faces	No	No	Yes	Yes	No	No	No	Yes	No	No	No
Merging edges	No	Yes	No	Yes	Yes	No	Yes	Yes	Yes	No	No
Short edges	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No	Yes	No

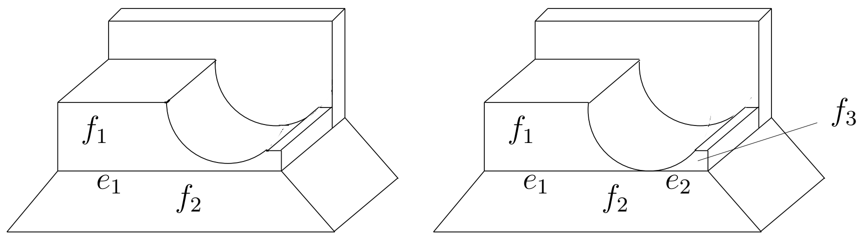
Table 1
Sequence of topological repair

to faces (a small face will remain small independently of the introduction of the new edge, etc.). However, the newly introduced edge may be short, and thus can also be part of a chain of short edges (e.g. consider removing gap B in Figure 1); it may also connect faces with the same geometry or connect small faces building a face chain.

(3) **Multiple face gaps.** A multiple face gap is a loop of connected edges where each of the edges lies on the boundary of only one face. Each of the involved vertices may lie on an arbitrary number of additional edges. Such gaps are not removed by the previous steps. We have to distinguish three cases, depending on the number of vertices the *additional* edges intersect in (within tolerance):
- **One vertex:** The edge loop is replaced by a single vertex which connects the additional edges. This vertex may now connect short edges leading to a chain of short edges.
- **Two vertices:** The edge loop is replaced by a single edge between the two intersection vertices. This edge may be short and may also create a chain of short edges. This edge may also connect two faces with the same geometry or connect two small faces creating a chain of small faces.
- **More vertices:** An additional face is added to close the gap. This face may be pinched, small or a sliver face and it may have the same geometry as an adjacent face.

Note that this step removes all remaining gaps, and no new gaps are introduced. Also note that none of the following steps are able to introduce new gaps as in all cases the connectivity between the elements may be changed, but not eliminated.

(4) **Pinched faces.** Topologically the boundary of a pinched face contains two sequences of edges whose geometry is in some sense closer than the tolerance. It is repaired by combining the edge sequences. The two sequences may be joined at a single vertex or both edge sequences may be replaced by a single



(a) Before modification (b) After modification

Figure 11. Fixing a pinched face creates a small face

edge sequence. There may or may not be a loop which contains both edge sequences, and combining the edges thus may or may not split a face into two faces. This means that either a new vertex or a new edge sequence is introduced. The new vertex may result in the creation of a short edge or a chain of short edges. The new edge sequence may result in adjacent faces having the same geometry, or a chain of small faces, or a chain of small edges. Furthermore, if the two close edge sequences are part of the same loop, then a new face is generated. This may be a sliver face, be adjacent to a face with the same geometry, or a small face. For instance, see the small face f_3 in Figure 11(b), the short edge e_2 in Figure 11(b), adjacent faces f_1, f_2 with the same geometry in Figure 3(b), and adjacent edges e_1, e_2 in Figure 3(b). However, repairing pinched faces cannot introduce any gaps. Thus, we fix pinched faces after removing gaps, but before fixing other problems.

(5) **Chains of small faces.** Chains of small faces should be processed as a whole, rather than face by face. A chain of small faces is a sequence of small faces pairwise connected by edges. No face in this sequence is connected to more than two other small faces in the sequence and the sequence has at least two faces. To repair this, each face is replaced by a vertex and the vertices are connected in the same sequence as the faces. Additional edges connected to vertices on the small faces are connected to the new vertex closest to their connection with the old vertex. This results in a chain of edges, as shown in Figure 4. It is very likely that this chain of edges is a chain of short edges, processed below. But it may also lead to isolated short edges and faces with the same geometry connected by the newly introduced edges. We also need to remove chains of small faces before single small faces to simplify detection. Thus, chains of small faces are handled at this early stage.

(6) **Sliver faces.** Topologically a sliver face is an ordinary face of the model, but its geometry is long and thin. It is replaced by one or more edges depending on the number of adjacent faces. We assume that there is no sliver face adjacent to another sliver face to avoid cases where we have many long thin faces which may really represent a different face geometry (e.g. a cylinder replaced by many adjacent long thin planes, due to inadequate data or earlier reconstruction algorithms). The introduction of new edges can lead to adjacent faces with the same geometry. The new edges may be short, and lead to a chain of short edges. Note that as a sliver face is long (if it were not, it would

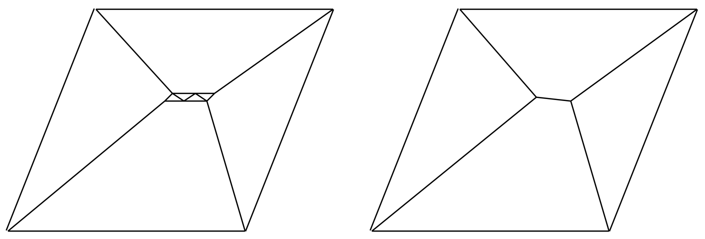

(a) Before modification (b) Final result

Figure 12. Replacing a chain of small faces

be a small face), the new edge cannot lead to a new chain of small faces (it may be part of such a chain for ambiguous cases of sliver / small faces, but then it would have been removed earlier).

(7) **Chains of short edges.** Chains of short edges are sequences of connected edges, where each vertex between two edges of the sequence connects exactly two edges. The vertices at the start and the end of the sequence can lie on any number of other edges not part of the sequence. A chain of short edges is replaced by a single edge between the start and the end vertex. If there is no start and end vertex we can simply replace it by an edge representing a closed curve. Chains of short edges should be processed as a whole, rather than edge by edge, as shown in Figures 4 and 12. This means we can later replace short edges by a single vertex and do not have to consider any other cases. Replacing the chain by a single edge can introduce a short edge, or adjacent edges with the same geometry.

(8) **Adjacent faces with same geometry.** Adjacent faces with approximately the same geometry are merged by removing all edges between them. This may create adjacent edges with the same geometry. It may also result in new small faces and edges. Thus, we fix adjacent faces with the same geometry at this stage.

(9) **Small faces.** Topologically a small face is an ordinary face of the object, but with a small geometry. It cannot be adjacent to any other small faces (such cases have already been handled by small face chains). Here small geometry essentially means that *all* distances between the vertices of the face are smaller than a tolerance. Hence, all edges connected to the face intersect in a single vertex within this tolerance and we can replace the small face by a single vertex. (Note that small faces which may have to be replaced by edge sequences are considered to be sliver faces—see above). Introducing a new vertex may require us to merge edges.

(10) **Merging edges.** Some of the previous steps may create isolated vertices which only connect two edges. These vertices have to be removed resulting in a single intersection edge between two faces. As this can result in short edges (merging two short edges may result in a new short edge), this is done before removing short edges.

(11) **Short edges.** First consider a short edge not connected to any other short edges. In this case we simply replace the short edge by a vertex, which cannot

introduce any other problems. Now consider short edges which are connected to other short edges. As we have already processed chains of short edges, these other short edges have to be between different face pairs. If the vertices of all connected short edges are within tolerance, they can be replaced by a single vertex connecting all edges adjacent to the group of short edges. In this case all short edges are removed and we have a single vertex which cannot introduce any new problems. However, if the vertices are further apart, we do not change anything. There is no obvious way in which to replace such a sequence. If it has not been processed earlier in the context of chains of small faces, etc., this indicates that the data or methods used to construct the raw model need to be improved (as discussed under adjacent sliver faces above).

Items above the diagonal in Table 1 are all "No". Thus if problems are fixed in the sequence given, no repair later in the sequence can cause a problem of a type already fixed earlier in the sequence. Note carefully the logic. Given *this* particular ordering, certain types of problem are known not be present at each stage, having been fixed earlier. Thus, certain potential complex interactions between multiple types of topological problem need not be considered, as they cannot arise, simplifying the analysis. Table 1 shows just one self-consistent ordering in which problems of the various types can be solved sequentially. Other orderings may also be possible. We thus carry out topological beautification of the various problems, as explained in further detail below, in this order.

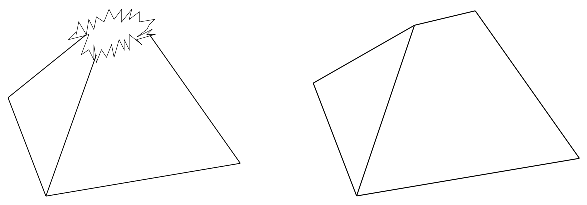
3.3 Algorithm Outline

In the following we give a brief overview of our topological beautification algorithm. The details are provided in Section 4. The input to our algorithm for detecting and modifying topological problems is a (reverse-engineered) B-rep model, together with a tolerance value. The output is a B-rep model with modified topology. The tolerance is used as described in Section 3.1. Pseudocode for the main algorithm is given in Figure 13; the problems are detected and corrected in the order justified in Section 3.2.

We start by detecting and removing from the model the gaps of various types (lines 2 to 6). This is done by detecting edges which are part of gap boundaries using the `find_halfedges` method. To find the three different gap types the edges in the list are clustered into lists of connected edges using the `find_gaps` method. If there is no edge adjacent to a cluster, the cluster forms a face gap. If there are only two edges connected to the cluster, we have an edge gap. Any other cluster has to represent a multiple face gap. Three lists, one for each gap type, are returned by `find_gaps` (for details see Section 4.1). Then `remove_facegaps` removes a gap in a face, such as A in Figure 1; `remove_edgegaps` removes gaps lying across an edge as shown for B in Figure 1; `remove_multiplefacegaps` re-

```
00: INPUT: body, tolerance
01: OUTPUT: modified body

02: halfedges_list = find_halfedges(body, tolerance)
03: (facegap_list, edgegap_list, multiplefacegap_list) = find_gaps (halfedges_list)
04: remove_facegaps (body, facegap_list)
05: remove_edgegaps (body, edgegap_list)
06: remove_multiplefacegaps (body, multiplefacegap_list)

07: WHILE ((face = find_pinchedface (body, tolerance)))
08:    modify_pinchedface (body, face)

09: smallfaces_list = find_smallfaces (body, tolerance)
10: facechains_list = find_facechains (smallfaces_list)
11: FOREACH facechain IN facehains_list
12:    remove_facechain (body, facechain)

13: WHILE ((face = find_sliverface (body, tolerance)))
14:    remove_sliverface(body, face)

15: shortedges_list = find_shortedges (body, tolerance)
16: edgechains_list = find_edgechains (shortedges_list)
17: FOREACH edgechain IN edgechains_list
18:    remove_edgechain (body, edgechains)

19: WHILE ((twofaces = find_adjacentfaces (body, tolerance)))
20:    merge_adjacentfaces (body, twofaces)

21: WHILE ((face = find_smallface (body, tolerance)))
22:    remove_smallface (body, face)

23: WHILE ((vertex = find_adjacentedges (body, tolerance)))
24:    merge_adjacentedges (body, vertex)

25: WHILE ((edge = find_shortedge(body, tolerance)))
26:    remove_shortedge (body, edge)

27: RETURN body
```

Figure 13. The main topological beautification algorithm



(a) Before modification (b) After modification

Figure 14. Repairing a simple multiple face gap

moves gaps which span several faces, as seen in Figures 2 and 14.

Secondly, we detect pinched faces (lines 07 to 08). The `find_pinchedface` method detects a pinched face in the body and the `modify_pinchedface` method modifies the model accordingly. We repeatedly look for, and remove pinched faces, until no more remain in the model.

The next step detects and removes chains of small faces by detecting small faces with `find_smallfaces`, combining them into face chains using `find_facechains`, and finally removing each chain with `remove_chainsmallfaces` (lines 09 to 12).

Afterwards, we detect and remove any sliver faces with the `find_sliverfaces` and `remove_sliverface` methods respectively, until no more sliver faces can be found (lines 13 to 14). An example is face B shown in Figure 5.

The methods `find_chainshortedges` and `remove_chainshortedges` are used to detect and remove chains of short edges analogously to chains of small faces (lines 15 to 18).

We next use `find_adjacentfaces` and `merge_adjacentfaces` to detect and merge any adjacent faces with the same geometry (lines 19 to 20). The former method compares each face with its adjacent faces in the body and decides if any two such faces have the same geometry. If so, the two faces are merged into a single face by `merge_adjacentfaces`, as shown in the example in Figure 8. Note that when faces (such as f_1 and f_2 in Figure 8) are merged by this method, we do not immediately consider merging adjacent edges (like e_1 and e_2 in this example): these will be processed by the edge merging process later. A marking process is used to avoid reconsidering the same face repeatedly [4].

Next, we repeatedly detect and remove small faces using `find_smallface` and `remove_smallface` methods until no more small faces can be found in the model (lines 21 to 22). A simple example is shown in Figure 7. Note that we have already detected small faces in line 09 when finding chains of small faces. Small faces which did not form part of a face chain can be cached for the small face removal step.

We now merge adjacent edges connected by a vertex which only lies on these two edges (see Figure 8). This is done by running the `find_adjacentedges` and `merge_adjacentedges` methods (lines 23 to 24). The former method checks the number of edges at each vertex and reports those which only lie on two edges. The merging method has to take care not to remove all vertices for a closed loop edge, in case the modeller requires such vertices.

Finally, we detect and remove any short edges using `find_shortedge` and then `remove_shortedge` repeatedly (lines 25 to 26).

Some of the loops are shown as `WHILE` loops above, rather than `FOR` loops. This allows for possibilities such as the merging of two faces with the same geometry producing a new face which also has the same geometry as further faces. In fact, to avoid reconsidering already processed items (e.g. faces) repeatedly in `WHILE` loops, we use a slight variation of the simplified algorithm given, which builds queues of items to be processed, and adds any relevant newly created items to the ends of the queues (e.g. for removing small faces). The queues can then be processed in order in linear time.

3.4 Example

An example is now given to illustrate the overall algorithm. A model before beautification is shown in Figure 15(a). In this model, cylindrical boss 1 is close to the edge of face f, causing it to be pinched. Problems 2, 3, 4 and 5 are gaps of various kinds. Problem 6 is a chain of several small faces. 7 is another small face, while the faces adjacent to edge 8 have the same geometry.

As described earlier, we first detect gaps in the body by finding edges which lie on only one face and putting them into a list. We then cluster the edges in this list to give four clusters which surround gaps 2, 3, 4 and 5 respectively, as shown in Figure 15(a). There are no edges adjacent to cluster 2 so it is put into the face gap list. There are two edges connected to cluster 3, so it is put into the edge gap list. The other clusters, 4 and 5, are put into the multiple face gap list. Each gap in the face gap list is removed by extending the face. Each edge gap is removed by extending the faces and connecting the edges meeting the gap. For each member of the multiple face gap list, we extend faces adjacent to the gap and intersect them as appropriate, possibly adding a new face as well as new edges and vertices, as explained in Section 4. For gap 4, such a new face is created. This new face is long and thin (a sliver face). For gap 5, the faces intersect in a single point, so we simply insert a new vertex, as shown in Figure 15(b).

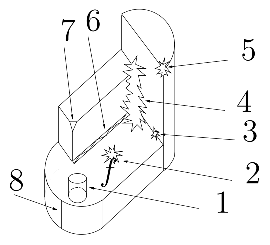
Next, we detect and remove the chains of small faces, 6, as shown in Figure 15(d). Sliver faces are now processed. The only sliver face found is the sliver face arising from gap 4. We replace it by an edge as shown in Figure 15(e). The chain of short edges arising from the chain of small faces 6 is next replaced by a single edge: see Figure 15(f). We now seek adjacent faces with the same geometry. Adjacent faces to edge 8 are found to have the same geometry within tolerance and are merged, as shown in Figure 15(g). Remaining isolated small faces, in this case face 7, are now detected and removed. See Figure 15(h).

Afterwards, we merge edge pairs connected by a vertex only lying on the two involved edges. Edges which belong to the new face produced by merging the edges on either side of edge 8 are found and are merged, as shown in Figure 15(i).

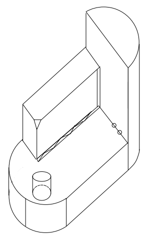
Finally we detect and modify short edges. No short edges need fixing in this case. The updated topology is now returned.

4 Algorithm Details

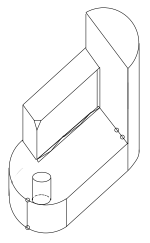
In this Section, we give further details of the methods called by the main algorithm. The discussion essentially follows the sequence they are used by the main al-



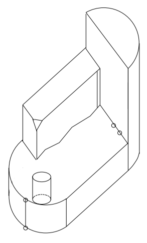
(a) Initial model



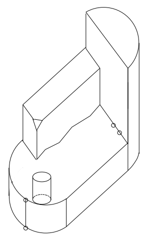
(b) After removing gaps



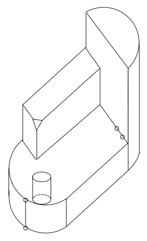
(c) After fixing pinched faces



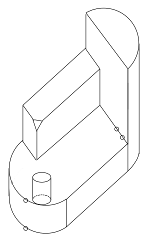
(d) After fixing chain of small faces



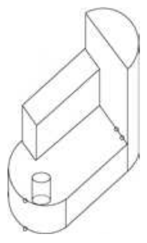
(e)After removing sliver face



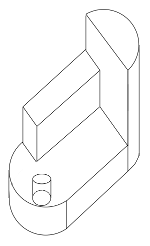
(f) After fixing chain of short edges



(g) After merging faces



(h) After removing small faces



(i) After merging edges

Figure 15. Modifying a complex model

gorithm.

4.1 Removing Gaps

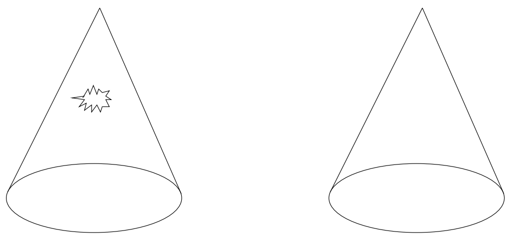
In the initial B-rep model, each "proper" edge is a boundary element of two faces. We refer to an edge which only lies on one boundary as a half-edge (slightly abusing the normal terminology). These edges can easily be detected in a B-rep data structure as edges associated with a single loop where this association is represented by a coedge. Gaps are surrounded by edges which only have one associated coedge, and belong to one loop. All edges having a single coedge are collected into a list by `find_halfedges` which is used as the input to the `find_gaps` method.

The main task of `find_gaps` is to determine groups of connected half-edges which are not connected to any other half-edges outside the group. This can be done by initially creating a table associating the involved vertices with the half-edges they lie on. Then we start with the first vertex in the table and mark it as used. We add the associated half-edges to a new group and from the new half-edges we collect the second vertices. If these vertices are in the vertex table and are not already marked as used, we mark them as used in the table and add their half-edges to the group (if they are not already in it). This is continued until no more vertices like this are detected. Then we form the next group with the next vertex in the table not marked as used until all vertices are marked as used.

This results in groups of connected vertices which now have to be classified according to the gap type they bound. A group with a single half-edge cannot arise assuming the input is a valid model. Thus, we first count the number of faces adjacent to the gap by counting the number of distinct loops to which the involved coedges belong. Depending on whether there are one, two or more faces involved we get the following three cases respectively:

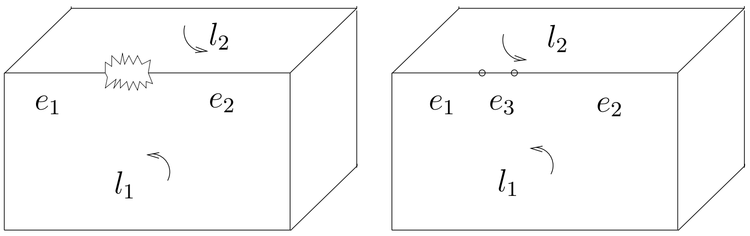
Face Gaps: Here the one-sided edge loop lies entirely within one face. This corresponds to a face with a gap or hole in it (see Figure 16). A typical example causing such a problem arises when there is a deep cylindrical hole or pocket in some face, inside which the scanner has been unable to collect data.
We do not attempt to recognise and insert a hole, but merely repair the model to a consistent state by deleting the gap, which can easily be done by removing the complete loop and all its associated edges and vertices from the model. The existing geometry of the face then naturally covers the gap.

Edge Gaps: Here the one-sided edge loop lies across an edge, i.e. across a pair of faces, as shown in Figure 17. This kind of problem is most likely to occur for concave edges, where internal reflections may cause scanning to fail.
As in the previous case, the one-sided hole loop is removed, but a new edge (like e_3 in Figure 17) is also inserted to join the two edges meeting the loop. The



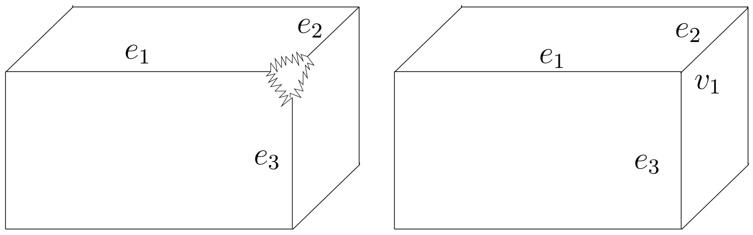
(a) Before modification (b) After modification

Figure 16. Repairing a gap in a single face



(a) Before modification (b) After modification

Figure 17. Repairing an edge gap



(a) Before modification (b) After modification

Figure 18. Repairing a multiple face gap

geometry of the inserted edge is immediately found by intersecting the faces on either side of it.

Multiple Face Gaps: Here the one sided edge loop lies across several faces (see Figures 18 and 19). Such problems typically arise at concave corners of the object.

To remove these gaps, firstly, the intersection edges of the faces adjacent to the gap are found by intersecting neighbouring faces pairwise. We then compute the intersection points, where they exist, of these intersection edges. If all intersection edges meet at a single point (within tolerance), we replace this gap by a vertex (see Figure 19(a)). If the intersection edges intersect in two points, as shown in Figure 19(b), we replace the gap with two vertices and a new edge. If the intersection edges intersect in three or more points, as in Figures 19(c) and (d), we add a new face as well as new vertices and edges. The geometry of the new face is constrained to interpolate the new vertices.

So far we have ignored the case of adjacent gaps where two gaps share a single

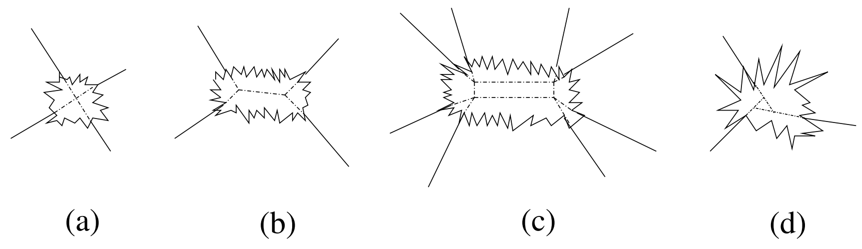


Figure 19. Cases of multiple face gap modification

vertex. These would be combined to form a single group by the above method. In order to handle such cases we need to process the groups of connected edges further. A simple tree-growing algorithm can be used to detect the fundamental cycles in the groups. Each fundamental cycle represents a gap. This is relatively inexpensive as the groups are small for models with localised topological problems and it only has to be done for groups with at least one vertex having more than two half-edges.

4.2 Modifying Pinched Faces

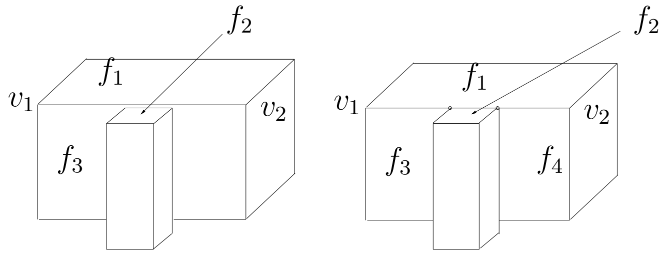
In general, detecting and correctly handling all cases of pinching is problematic. Here, we only attempt to identify and fix in a straightforward way a range of frequently occurring simple cases. Further discussion is given in Section 7.

To identify pinched faces, we compute the (minimum) distance between each pair of non-consecutive edges of the face. If the distance is smaller than the tolerance, the face is potentially a pinched face. However, it could also be a sliver face, or a small face. These cases must be detected separately, using the methods given later, and removed from consideration. (The information that such faces are small or pinched is cached, however, to save time later.)

To remove the pinching, we identify and treat subcases differently. We handle each occurrence of pinching independently—multiple occurrences of pinching in the same face are resolved sequentially. To fix the pinching, two pieces of information are needed: where the pinching occurs, and the kinds of edges adjacent to the pinching.

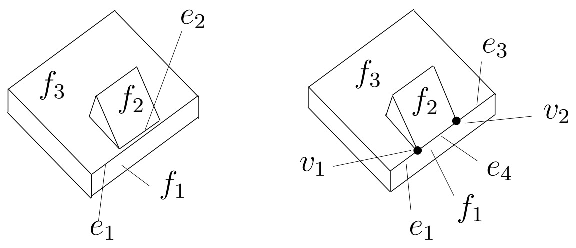
In terms of location of pinching, we identify three cases which we have named as follows:

- **Necking.** The pinching almost cuts the pinched face into two large pieces. See the faces marked f_3 in Figure 20(b).
- **Spiking.** The pinching is adjacent to a short edge (or several making a short chain) in the outer boundary of the face, as shown in Figure 3(a): the pinching causes a “spike” to stick out of the main portion of the face.



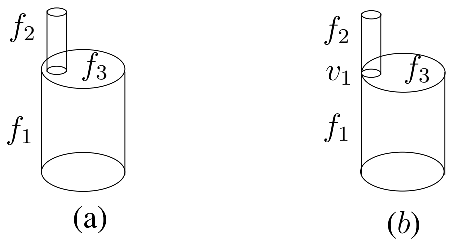
(a) Before modification (b) After modification

Figure 20. A *necking* pinched face with *straight* pinching



(a) Before modification (b) After modification

Figure 21. A *bossing* pinched face



(a) Before modification (b) After modification

Figure 22. A *bossing* pinched face with *complex* pinching

- **Bossing.** The pinched face has an outer loop and an inner loop adjacent to the place where pinching occurs. For example, face f_3 has an outer loop, which is adjacent to f_1, and an inner loop, adjacent to f_2, in Figures 21(a) and 22(a).

These cases can easily be distinguished. If edges on either side of the pinching belong to different edge loops, we have *bossing*, as shown in Figure 21 (the inner loop could actually surround a pocket instead of a boss). Otherwise, the edges on either side of the pinching belong to the same outer edge loop (see Figures 3 and 20). Ignoring the edges taking part in the pinching, the other edges in the loop form two connected sequences at either end of the pinching. We measure the lengths of these sequences (as the sum of the respective edge lengths). If both sequences are long, we have a *necking* case (see Figure 20). If one sequence is short, we have *spiking* (see Figure 3). (Both cannot be short, as this would imply a sliver face).

Next, we also identify two possible configurations of edges adjacent to the pinching. These configurations can arise independently of the location of the pinching.

Note that several consecutive faces may exist on each side of a single area where the face is pinched. Furthermore, the other faces next to the pinching may be planar or curved.

- **Straight.** A single edge exists on either side of the pinched area, and both are straight lines (see Figure 20).
- **Complex.** A single edge exists on either side of the pinched area, and at least one is curved, or multiple edges are present on at least one side of the pinched area (see Figure 22).

To modify pinched faces, in each case we attempt to close the gap. The method of closing the gap depends on the configuration. For *straight* cases, the pinched part of the face is replaced by a new straight edge, and other topology is adjusted according to the location. An example is shown in Figure 21. For *complex* cases, the pinching is closed by finding the narrowest gap across the pinching, and making the appropriate pair of edges next to it, one from either side, meet in a new vertex (see Figure 22). Again, other topology is adjusted according to the location case. The other changes made to the topology are as follows:

- **Necking.** To correct necking, the single edge loop is split into two loops at the position where the pinching occurs. Depending on whether we have the straight or the complex case, either a single vertex or an edge is inserted. For example, see Figure 20. The original face f_3 is cut into two, one piece remaining f_3, and the other becoming a new face. Old edges of f_3 are put into face loops of f_3 and f_4, or removed, as needed. The new edge and its end vertices, or the new vertex, according to configuration, are incorporated into the face loops of the faces f_1 and f_2 on either side of the pinching.
- **Spiking.** Irrespective of whether we have a *straight* or *complex* case, we replace the pinched part by an edge. In the *straight* case, this is simple. The *complex* case is handled in an analogous way, where (after splitting certain edges if necessary) some existing edges are kept and others deleted to give the new loop for face f_3. We now proceed as for *necking*, except that face f_3 is not split, and there is no f_4 to consider; however, we also have to adjust the face loops of the end face or faces (see Figure 3).
- **Bossing.** We have two edge loops, one inside the face of the other one. Both loops are split in a similar way to the necking case, but instead of splitting the face in two, the two loops are combined to give a single loop at the newly inserted vertex (or vertices). The loops of adjacent faces have to be adjusted accordingly. See Figure 22. Loops of faces f_1 and f_2 are adjusted as in the *necking* case. Appropriate edges from the inner and outer loops of f_3 are merged to give a new outer loop for f_3.

In the following we first describe how to detect and remove chains of small faces. Detecting and removing short edges is done in a very similar way so we describe both here. However, note that removing chains of short edges is done later *after* removing sliver faces.

Chains of small faces are shown in Figures 4(a) and 12(a). To detect such chains, we first detect all small faces in the model. This information is also cached for later use when removing isolated small faces (which do not belong to chains). To detect the chains in the set of small faces, we employ the `find_facechains` method which returns a list of chains.

This method works in similar way to `find_gaps` described in Section 4.1. In each chain, each element is connected to other elements of the chain, and each element has only two adjacent elements, except for the elements at the ends of the chain. Thus, if a member has more than two adjacent elements, the chain is divided here into multiple chains. Note that this means that we cannot have surface areas partitioned into small faces (which should be replaced by a single face). Again, we consider that such problems can only arise due to problems in data acquisition or earlier model building processes.

In order to find the chains, we first set up a table of edges connecting small faces. For each such edge we store the two small faces it connects. This table can easily be set up by traversing the list of unique edges connecting small faces. Starting at an arbitrary small face, we can then collect the small faces connected to it over its edges. If there are none, or more than two small faces connected to it, we do not have part of a chain. Otherwise we follow either one or two paths from the original face using the edge table until we reach a face with more than two neighbours. By keeping track of already processed faces, we avoid considering the same chains more than once. This also avoids infinite iteration in the case of closed loops.

After detection, each chain of small faces (with at least two elements) is partially repaired by replacing it by a chain of short edges, as shown in Figure 4(b). This is done by joining mid-points of edges shared by small faces in the chain.

In the same way, chains of short edges are detected by finding short edges using `find_shortedges` and combining these in to chains using `find_edgechains`. The latter works in a similar way to `find_facechains`, where short edges connected by vertices to one or two other short edges are detected. Each chain in the edge list is replaced by a single edge, as shown in Figure 4(c), joining the ends of the chain. Note that there can only be a single face on either side of the chain, because of the way a chain is defined. The geometry of the edge is set to the geometry of the intersection of the adjacent faces. See also Figure 12(b).

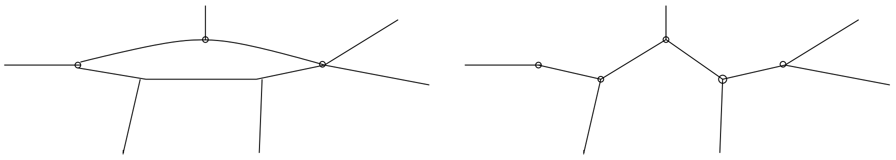

(a) Before modification (b) After modification

Figure 23. Replacing a sliver face by edges

4.4 Removing Sliver Faces

A sliver face is a long thin face. To decide if a face is a sliver face, we compute its area, and the diagonal length of its bounding box. If the ratio of these two numbers is smaller than the length tolerance, the face is considered to be a sliver face.

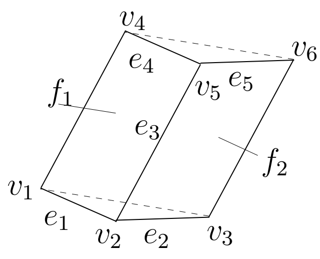
Each "long side" of the sliver face can be bounded by one or more edges, as shown in Figure 23. Each sliver face is replaced by an edge sequence. The vertices used in the edge sequence are those vertices of the sliver face which are also connected to some other edge. These vertices are joined by a chain of edges, as shown in Figure 23. The two closest vertices are first connected by an edge. The remaining vertices are joined one by one to the ends of the growing edge sequence in such a way that the next vertex added is always the one which is closest to one end or the other of the sequence. We finally update the face loops adjacent to the edge sequence.

4.5 Merging adjacent faces and edges

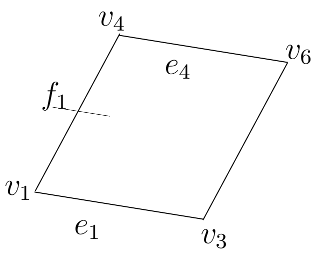
Two adjacent faces with the same geometry (to within a tolerance) can arise in a reverse engineered model, as shown in Figure 6(a). The edge(s) between them should be removed and the two faces should be merged, as shown in Figure 6(b).

To detect such faces, the method considers each edge of the model. If the two faces on either side of the edge are of the same type and have the same surface parameters (e.g. have the same normal vector and distance from the origin if planar) then the two adjacent faces are merged. The method removes all common edges between the two faces. Vertices which only belong to a common edge and other edges of the faces to be merged are also removed (see v_2 and v_5 in Figure 24(a)), and the adjacent edges are merged (such as e_1 and e_2 in Figure 24(a)). If there is more than one common edge between adjacent faces with the same geometry, all such edges are removed, as shown in Figure 25.

Adjacent edges which are connected by a vertex only lying on two edges may also exist. If just two edges meet at any given vertex, the vertex may be removed and the edges may be merged. All vertices in the model are considered in turn to find

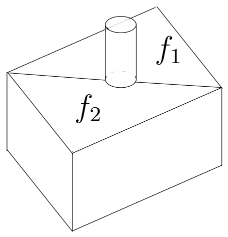


(a) Before modification

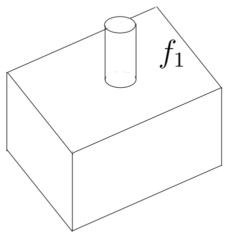


(b) After modification

Figure 24. Merging two faces



(a) Before modification



(b) After modification

Figure 25. More than one common edge between two faces with the same geometry

such cases. After removing the vertex, the face loops of the faces on either side of the new edge are updated.

4.6 Detecting and removing small faces and short edges

The aim of this part of the processing is to discard small faces. Small faces are detected by computing the area of each face. If the area of the face is smaller than the square of the length tolerance, it is a small face. We remove a small face by replacing it with a new vertex at the centroid of its vertices. Edges adjacent to the small face are connected to this vertex. Edge loops of faces adjacent to the small face are also updated. Note that small faces have already been determined earlier and cached for this step.

Similarly short edges are detected easily as edges whose length is smaller than a given tolerance. Each short edge is replaced by a vertex at its mid point, and the surrounding topology is updated. Short edges were also detected and cached for processing here by earlier steps.

5 Algorithm Analysis

Our algorithm runs the steps in the sequence explained in Section 4 on the initial B-rep model. To simplify the discussion, we loosely use n to interchangeably denote the number of faces, edges, or vertices; doing so is justified by Euler's formula which is a linear relation between these quantities. Furthermore, while it is possible to construct mathematical objects where a few faces have many vertices, and most faces have just a small number, such objects are uncommon in engineering practice, so we also assume that the number of vertices and edges for each face is no more than a small constant m, and that each face has no more than m neighbours. We note that the number of gaps, small faces, short edges, and so on, must all be less than n. We analyse each stage as follows.

Finding all edges belonging to gaps takes time $O(n)$. The `find_gaps` method first constructs a table linking the vertices with the half-edges they lie on. This can be done in $O(n)$ time. Then, using this table, groups of connected half-edges are determined. Each edge belongs to only one such group. So an edge is considered once as a seed (and ignored if it has already been added to another group) and once to build up the sequence for the group. Thus, gap finding takes time $O(n)$. Processing each gap requires the edges in the gap to be removed, and adjacent topology to be updated. In the worst case, faces have to be intersected pairwise, taking time $O(nm^2)$.

Pinched face detection considers each face in turn, and makes a comparison of distances between each pair of edges of that face, taking time $O(nm^2)$. Each pinched face is modified in time $O(m)$. Thus the pinched face methods take time $O(nm^2)$.

Assuming the area of each face can be computed in time $O(m)$, the time required to detect small faces is $O(nm)$. Chains of small faces are detected using a similar approach to the `find_gaps` method. Again using a table, each face is at most considered twice, so the time taken is $O(n)$. Each face in a chain is replaced by an edge, and the adjacent faces to the chain must be updated, which also takes time $O(nm)$.

Assuming $O(m)$ time computation for area and bounding box for a face, sliver faces are detected in time $O(nm)$, as each face must be processed. They are also modified by the `remove_sliverface` method in time $O(nm)$, giving an overall time of $O(nm)$.

Chains of short edges can be detected in time $O(n)$, as the length of each edge can be found in constant time. Chains are replaced by a single edge, and the loops of adjacent faces are also updated, also taking time $O(n)$.

Finding adjacent faces with the same geometry is done by comparing each face with neighbouring faces, taking time $O(nm)$. Merging adjacent faces with similar geometry takes time $O(n)$.

Finding small faces takes $O(n)$ time; this is already done during the detection of chains of small faces. When removing small faces, there may be m faces to adjust around each problem. Hence, removing small faces can take time $O(nm)$. Similarly, finding and fixing short edges can take time $O(nm)$.

Merging edges which are connected by a vertex only lying on two edges requires $O(n)$ time for detection and $O(n)$ time to repair.

Strictly, further justification is needed that when new topology is created, it does not increase the size of the problem for later stages in some way. This is not difficult to see informally, as in no case is a problem resolved by adding more than a *local* amount of new topology, and we have already assumed m to be a small constant. This is also further justified by being able to fix the problems in sequence as discussed in Section 3.2.

Each of the steps takes linear time with respect to n, given our assumptions above, and so the algorithm for adjusting the topology is expected to take linear time for realistic engineering objects.

However, in addition, a suitable new geometry must be generated which is consistent with that topology. This is done as described in our earlier work [7]. Regenerating the geometry takes much longer than finding and correcting the topological problems. For example, for Object 8 described in our experiments later, the topological problem detection and repair process takes under 2 seconds (on a 450MHz Pentium III computer running Linux); running our geometric reconstruction algorithm (on a 700MHz Pentium III computer) to detect geometric regularities in this model and choose an overall set of constraints coming from both topological and geometric beautification takes 50 seconds. Numerical constraint solving and generating the final geometry for the new model takes 148 seconds.

6 Experiments

This section describes a number of practical tests which demonstrate the capabilities and speed of our topological beautification algorithm.

6.1 Test objects

Various test objects were used to verify that our topological beautification algorithm produces the expected results. Objects 1 to 7 are simple models exhibiting only a single problem. Objects 8 to 10 are complex models which contain more than one problem.

Objects 1 to 4 are derived from cubes. Object 1 has a face gap in the top face, as shown in Figure 26(a). Object 2 has a multiple face gap, as shown in Figure 26(b). Object 3 has an edge gap on the top right edge, as shown in Figure 26(c). Object 4

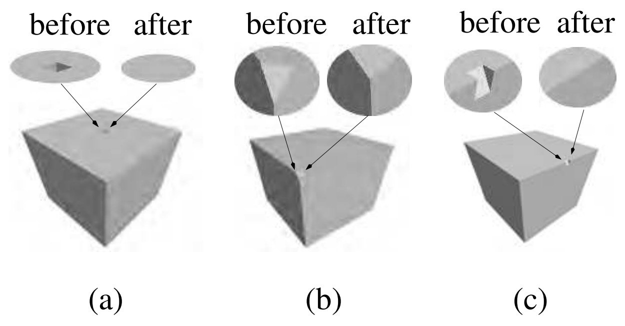


Figure 26. Objects 1, 2 and 3 before and after modification

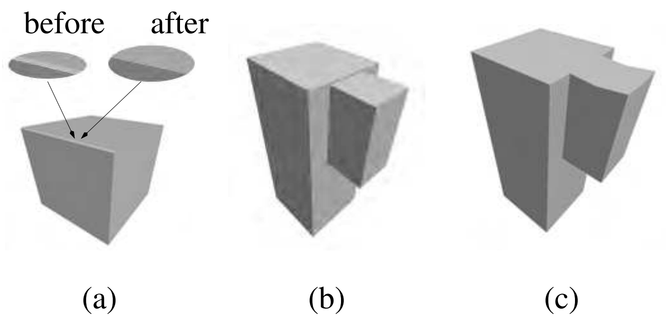


Figure 27. Objects 4 and 5 before and after modification

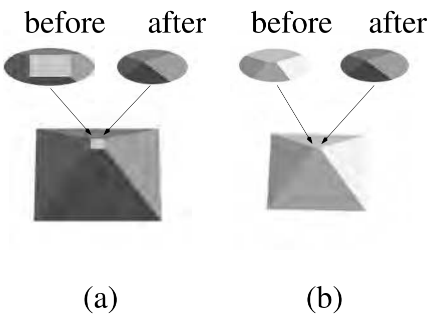


Figure 28. Objects 6 and 7 before and after modification

is a cube with a sliver face on the top left side, as shown in Figure 27(a). Object 5 illustrates a pinched face, and is composed of two blocks whose top two faces have almost the same geometry: see Figure 27(b); results of modifying it are shown in Figure 27(c). Objects 6 and 7 are based on four-sided pyramids. Object 6 has a small face in place of the apex, as shown in Figure 28(a). Object 7 has a short edge in place of the apex, as shown in Figure 28(b).

Object 8 is a plate with blocks and pyramids on top (see Figure 29(a)). The boss at the top left has an edge gap on its top right edge. The left side face of this boss and the left side face of the plate produce a pinched face. The middle boss at the back of the plate has a small sliver face on its top front side. The middle boss at the front of the plate has a vertex gap on the top left front side. On the right of the plate, there are two pyramids. The front pyramid has a small face in place of its apex and the other pyramid has a short edge in place of its apex. Between the two pyramids, the top face of the plate has a small cylindrical gap.

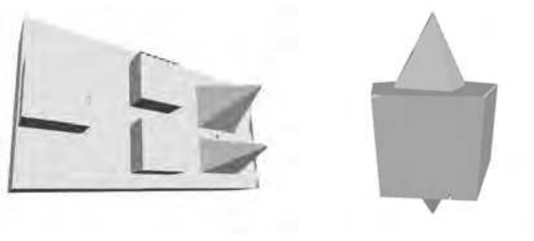


Figure 29. Objects 8 and 9 before modification

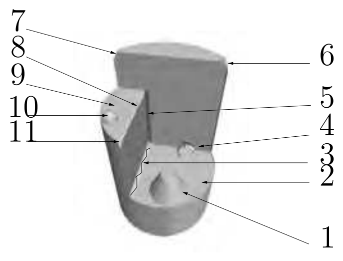


Figure 30. Object 10 before modification

Object 9 is prism with a pyramid on top and bottom (see Figure 29(b)). The top pyramid has a short edge at its apex and the bottom pyramid has a small face at its apex. On the top left of the prism, there is a vertex gap. On the top right side of the prism, there is small sliver face. On the bottom front side, there is an edge gap.

Object 10 is a complex model, shown in Figure 30, which contains pinched faces (2 and 9), a chain of small faces (3), an edge gap (4), two face gaps (5 and 7), two small faces (6 and 11) and a sliver face (8). In the model, 1 is a conical boss and 10 is a spherical boss. Note that sliver face 8 is adjacent to multiple face gap 5 and small face 11.

6.2 Topological Beautification Results

Table 2 shows the results of running the algorithm on these objects. Columns 2 and 3 are the number of faces and edges of each model before topological beautification. Columns 4 and 5 are the number of faces and edges of each model after beautification. Column 6 is the time taken to run the algorithm, and includes topological problem detection and topological repair time only. Our algorithm was implemented on a 450MHz Pentium III computer, using Linux and ACIS as the core modeller. Each of the test objects was successfully modified as expected.

Test object	Before modification		After modification		Time (seconds)
	Number of faces	Number of edges	Number of faces	Number of edges	
1	6	16	6	12	0.89
2	6	15	6	12	0.95
3	6	19	6	12	0.87
4	7	15	6	12	0.90
5	11	24	10	24	0.95
6	6	12	5	8	0.85
7	5	9	5	8	0.78
8	31	83	28	64	1.61
9	17	51	15	31	1.22
10	21	40	10	15	1.59

Table 2
Topological modification results for Objects 1 to 10

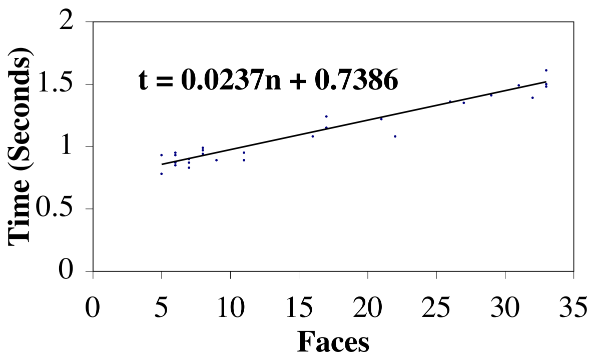


Figure 31. Algorithm performance

6.3 Running time

To evaluate the performance of our algorithm, we ran it on a larger variety of objects, including those shown earlier and others. Running times were averaged over several runs for each test object. The times taken and results of modification are summarised in Table 3. A linear analysis was performed to find the best fit to the timing data of the form $t = a \times n + b$, where t was the time taken, n was the number of faces and a, b were constants to be fitted. The results gave an empirical performance for the algorithm of time $t = 0.0237 \times n + 0.7386$. A plot of this fit is shown in Figure 31; good agreement is obtained with the predicted linear performance for the topological beautification algorithm.

7 Discussion

Some steps of the algorithm we present are computationally intensive, and were used with the aim of demonstrating an overall working system in a research context, rather than making each component individually as fast as possible. One such case is the need to calculate the area and bounding box of a face to decide if it is a sliver face. Undoubtedly some components could be tuned or replaced with more efficient methods. Nevertheless, our methods are already of an acceptable speed, taking just a few seconds, especially when placed in the context of an overall reverse engineering system where data acquisition alone can take many hours.

Our algorithm does not handle *all* possible cases of topological beautification. Firstly, it is only intended to handle those cases which we believe will arise in raw B-rep models coming from our approach to model building in reverse engineering. Secondly, some of our algorithms make assumptions about the kinds of defect expected, and it is certainly possible that cases could arise which break these assumptions. While it is not too difficult to generate such examples by hand with a little thought, it is not clear whether such cases can occur in real reverse engineering. A much more substantial research and development project would be needed to rigorously analyse and demonstrate which cases could and could not occur, and to ensure that a beautification algorithm handles all of the possible cases correctly. However, we believe that our algorithm is of practical use, and can automatically correct many problems, while acknowledging that user assistance might be needed in certain cases. This is in line with the experience of research into development of related algorithms for CAD model healing.

Three specific problems are noted which our algorithm cannot currently handle, and for which further research is needed:

- **Sliver faces.** We have assumed all sliver faces are unbranched. However, it is possible to have branched sliver faces such as the one shown in Figure 32(a).
- **Networks of connected faces and edges.** When detecting chains of faces and edges, we always assume that there is only a single chain, and not a more complicated branched or closely connected area of small elements. This is justified by observing that if an original natural face has been split into many small faces, then the original data is too noisy to reconstruct the model properly and should be improved. However, at least in some cases it may be possible to detect these defects and correct them in a beautification step. Note that this may also need us to consider the global structure (e.g. a network of small faces along a large number of natural edges of the object).
- **Unrealisable topology.** As the new topology is constructed, geometric constraints are generated to enforce that topology. It may be possible that constraints describing the topology cannot be realised due to the nature of the geometry, especially if the user has chosen an inappropriate tolerance value. For example,

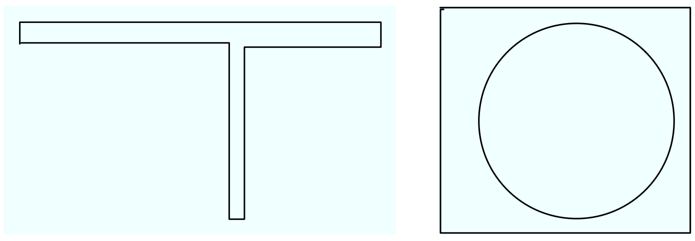

(a) A branched sliver face (b) Multiple pinching

Figure 32. Problem cases

consider the cylinder atop a block shown from above in Figure 32(b). If we were to try to fix all instances of pinching here, the new vertices produced would impose four positional constraints on the circular cross-section of the cylinder—but a given circle is completely determined by three points.

More consideration is also needed of how robust some of the ideas are when applied to extreme or unusual cases. The method given to detect a sliver face clearly makes sense for nearly planar, approximately straight sliver faces, but could give questionable results for long thin helical surfaces, for example. Overall, there is a trade-off to be made between speed, robustness, and generality of methods which would need to be carefully addressed in a commercial implementation of topological beautification.

Further investigation is also needed to determine to what extent the proposed approach is applicable to reverse engineered objects having free-form surfaces, and whether other specific kinds of topological beautification may be needed in such cases.

8 Conclusions

We have presented an efficient topological beautification algorithm for use in reverse engineering. It can detect and modify many topological defects of reverse engineered models including gaps, short edges, small faces, pinched faces, and sliver faces. Our experiments show that the algorithm produces the expected corrections. The algorithm takes approximately linear time to detect and correct topological errors in realistic engineering objects. Modifying the accompanying geometry, as described in [6,7], takes much longer. We note that in this paper we assume that an appropriate tolerance is provided by the user or some other mechanism such as a transitive clustering of involved positions. In many cases this tolerance can be found easily, but more research is required to devise an adaptive method for detecting appropriate local tolerances automatically.

9 Acknowledgements

The authors wish to thank Tamás Várady of the Hungarian Academy of Sciences and CADMUS Ltd. for providing reverse engineering software and for many helpful discussions. This research was funded by EPSRC grant GR/M78267.

References

[1] G. Butlin, C. Stops. CAD data repair. In: *Proc. 5th Int. Meshing Roundtable, Sandia National Laboratories*, 7–12, 1996.

[2] P. Campbell-Preston. *Personal communication*, 1998.

[3] T. K. Dey, H. Edelsbrunner, S. Guha, D. Nekhayev. Topology preserving edge contraction. *Publications de l' Institut Mathematique (Beograd)*, 60(80):23–45, 1999.

[4] C. H. Gao, F. C. Langbein, A. D. Marshall, R. R. Martin. Approximate congruence detection of model features for reverse engineering. In: M.-S. Kim (ed), *Proc. Shape Modeling International*, IEEE Computer Society, Los Alamos, CA, 69–77, 2003.

[5] I. Guskov, Z. J. Wood. Topological noise removal. In: *Proc. Graphics Interface*, 19–26, 2001.

[6] F. C. Langbein. *Beautification of Reverse Engineered Geometric Models*, PhD Thesis. Department of Computer Science, Cardiff University, June 2003. `http://www.langbein.org/research/BoRG/thesis.html`.

[7] F. C. Langbein, A. D. Marshall, R. R. Martin. Choosing consistent constraints for beautification of reverse engineered geometric models. *Computer-Aided Design*, 36(3):261–278, 2004.

[8] F. C. Langbein, B. I. Mills, A. D. Marshall, R. R. Martin. Approximate geometric regularities. *Int. J. Shape Modeling*, 7(2):129–162, 2001.

[9] F. C. Langbein, B. I. Mills, A. D. Marshall, R. R. Martin. Finding approximate shape regularities in reverse engineered solid models bounded by simple surfaces. In: D. C. Anderson, K. Lee (eds.), *Proc. 6th ACM Symp. Solid Modeling and Applications*, ACM Press, New York, 206–215, 2001.

[10] F. C. Langbein, B. I. Mills, A. D. Marshall, R. R. Martin. Recognizing geometric patterns for beautification of reconstructed solid models. In: *Proc. Int. Conf. Shape Modelling and Applications*, IEEE Computer Society Press, Los Alamitos, CA, 10–19, 2001.

[11] A. A. Mezentsev, T. Woehler. Methods and algorithms of automated CAD repair for incremental surface meshing. In: *Proc. 8th International Meshing Roundtable*, South Lake Tahoe, 299–309, 1999.

[12] B. I. Mills, F. C. Langbein. Determination of approximate symmetry in geometric models — an exact approach. Submitted to *Computational Geometry and Applications*, 2002.

[13] B. I. Mills, F. C. Langbein, A. D. Marshall, R. R. Martin. Approximate symmetry detection for reverse engineering. In: D. C. Anderson, K. Lee (eds), *Proc. 6th ACM Symposium on Solid Modelling and Applications*, ACM Press, New York, 241–248, 2001.

[14] B. I. Mills, F. C. Langbein, A. D. Marshall, R. R. Martin. *Estimate of frequencies of geometric regularities for use in reverse engineering of simple mechanical components*. Technical Report GVG 2001–1, Geometry and Vision Group, Department of Computer Science, Cardiff University, 2001. `http://ralph.cs.cf.ac.uk/papers/Geometry/survey.pdf`.

[15] J. C. Park, Y. C. Chung. A tolerant approach to reconstruct topology from unorganized trimmed surfaces. *Computer-Aided Design*, 35(9):807–812, 2003.

[16] N. A. Petersson, K. K. Chand. Detecting translation errors in CAD surfaces and preparing geometries for mesh generation. In: *Proc. 10th Int. Meshing Roundtable*, Sandia National Laboratories, 63–371, 2001.

[17] A. E. Uva, G. Monno. A new method for the repair of CAD data with discontinuities. In: *Proc. Convegno Italo-Spagnolo — Progettazione e Fattibilita dei Prodotti Industriali*, June, 1998.

[18] T. Várady, R. R. Martin. Reverse Engineering. In: G. Farin, J. Hoschek, M. S. Kim (eds), *Handbook of Computer Aided Geometric Design*, Elsevier Science, Ch. 26, 2002.

Before modification		After modification		Time
Number of faces	Number of edges	Number of faces	Number of edges	(seconds)
5	9	5	8	0.78
5	9	5	8	0.93
6	12	5	8	0.85
6	15	6	12	0.95
6	16	6	12	0.87
6	19	6	12	0.93
7	15	6	12	0.87
7	15	6	12	0.83
7	15	6	12	0.90
8	16	7	15	0.94
8	28	7	15	0.97
8	28	7	15	0.99
9	21	6	12	0.89
11	24	10	24	0.95
11	24	6	12	0.89
16	32	15	31	1.08
17	49	15	31	1.24
17	49	15	31	1.15
17	51	15	51	1.22
21	40	10	15	1.59
22	57	6	12	1.08
26	71	28	64	1.36
27	69	18	40	1.35
29	80	23	52	1.41
31	81	23	52	1.49
31	83	28	64	1.48
32	81	28	64	1.39
33	84	28	64	1.50
33	84	28	64	1.61

Table 3
Test results for further objects