IMPROVED IMPRINT AND MERGE FOR CONFORMAL MESHING

David R. White[1] and Sunil Saigal[2]

[1]*Sandia National Labs[*] and Carnegie Mellon University, Pittsburgh, PA, U.S.A. drwhite@sandia.gov*
[2]*Carnegie Mellon University, Pittsburgh, PA, U.S.A. saigal@andrew.cmu.edu*

ABSTRACT

This paper presents an algorithm for imprinting and merging adjacent parts. Imprint and merge is often used to facilitate generation of conformal meshes between adjacent parts. The algorithm tolerantly intersects the discretized boundary edges of adjacent faces to calculate the imprint Boolean. An input tolerance is used during the process to minimize the effect imprinting has on the meshing process. The topology changes from the imprint are generated using virtual geometry so that tolerant topology can be used. Several examples demonstrate how the present approach may be utilized to improve the mesh quality of conformal meshes over multiple parts. The approach is shown to work robustly with misaligned and poorly defined parts.

Keywords: imprint, merge, conformal meshing, tolerant intersections, assembly clean up, R-Tree

1 INTRODUCTION

The finite element process uses a mesh or grid to approximate a physical model. The mesh is typically generated as discretization of a solid model. The solid models are usually generated by Computer Aided Design (CAD) packages.

Multiple parts or assemblies are often modeled in a process which requires mesh generation programs to build conformal meshes between the different parts. A conformal mesh is one where assembly parts have shared nodes and elements at adjacent interfaces. An example of such a mesh is shown in Figure 1, where there are several parts that have adjoining interfaces in the assembly. At each of the part interfaces, the elements and nodes are shared between the parts making the entire mesh conformal.

A geometry or CAD package is often used to aid conformal meshing. Geometry or CAD packages have tools to *imprint* topology across adjacent faces and curves. The result of imprinting is that adjacent parts will have coincident or mirrored topology that can be used to ensure a conformal mesh. Some meshing packages additionally *merge* the coincident topology to make adjacent parts share the same topology at these locations. Merging allows the topology to maintain the conformity of the mesh. Some geometry packages facilitate conformal meshing by allowing parts to be built with "cellular topology" where adjacent parts already have been imprinted and merged. This is often difficult since the parts are typically built individually and would require an imprint and merge to become cellular.

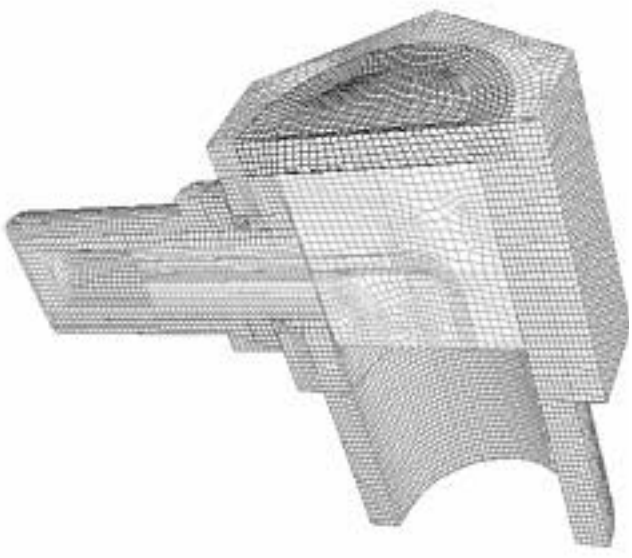

Figure 1 Conformal Mesh

Loose tolerances or user error from CAD packages often make conformal mesh generation tedious [1]. Adjacent parts can either overlap or have unintentional gaps. These overlaps

[*] Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy under contract DE-AC04-94-AL85000

and gaps make mesh generation difficult and often require rebuilding part of or the entire solid model for meshing purposes. Currently, none of the geometry clean-up techniques [2,3,4,5] fix problems that result from part interactions. An example of such a case is shown in Figure 2, where the shape of the parts prevent them from properly matching with each other, even though they would so match for modeling purposes. The zoomed regions in this figure show one of the four sliver surfaces that are created from the misalignment and the imprint process.

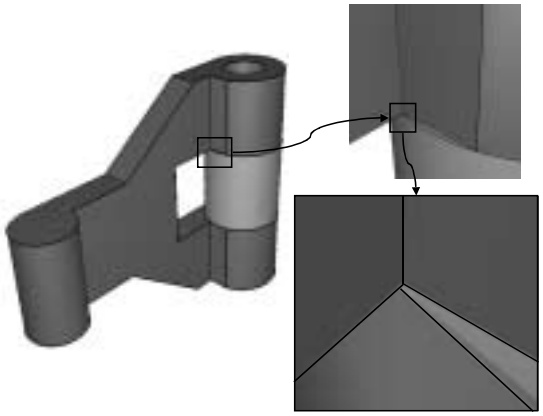

Figure 2 Misaligned Parts (Courtesy of SNL)

This paper presents new imprint and merge techniques that are tolerant to "dirty" assembly models and that improve the minimum quality of the mesh that can be created for the models. Additionally, by combining imprinting and merging and applying a spatial search tree, imprint and merge efficiency is improved from $O(N^2)$ to O(N) in practice, where N is the total number of faces in the solid model.

1.1 Conformal Meshing

Conformal meshing can be achieved in a variety of ways including:

- *Mesh Merging*
- *Imprint and Mesh Mirror*
- *Imprint and Merge*

The first approach, *mesh merging*, can be achieved either manually or automatically. The manual approach requires the user to select regions or even nodes and elements that should be merged or united. The user is also required to ensure that similar meshes are generated in regions where conformity is desired. In this approach, elements on the boundary between interacting parts are selected. The mesh is then modified by edge swaps and node insertions until the overlapping regions are conformal. Both approaches have their limitations. Manual merging is tedious and does not scale well for complex assemblies. Automatic merging is advantageous for unattended mesh generation, but the results often have unintentional void regions where the mesh does not properly conform well to the geometry due to the lack of constraining topology. Automatic merging works for tetrahedral mesh generation only. Additionally, it is more difficult to apply boundary conditions such as heat sources and flow rates because the mesh may not conform to the original model.

The last two conformal meshing approaches use the geometric operation of imprinting. Imprinting can be viewed as a geometric Boolean operation between two solid parts or faces. The imprint Boolean first calculates the intersection graph between the two objects. The intersection graph consists of edges that define how the two objects intersect each other. The graph is then split into parts that affect each object. The pieces of the graph are used to split the boundaries of the object. This will create coincident topology where the two objects intersect.

An example of imprinting is shown in Figure 3 where a brick part and a cylindrical part have adjacent surfaces. Figure 3 (a) shows the parts separated before the imprint operation. Figure 3 (b) shows the two parts together as they are in the model, and (c) shows how the brick part is modified topologically through imprinting. With the modifications from the imprint operation, the brick part and the cylindrical part can now be meshed conformably.

Figure 3 Imprinting

The *imprint and mesh mirror* approach of conformal meshing uses the previously discussed imprint Boolean to first make the adjacent parts topologically equivalent. After imprinting, the approach then ties the coincident geometry together so that when one topological entity (face, edge or vertex) is meshed on one part, the mesh is mirrored to the topology of the adjacent. The mirrored mesh is then tied back to the original mesh so that if a conformal mesh is required, only the original nodes and elements are referenced.

Imprint and merge is similar to the *mirror* approach except that rather than mirroring the meshes, the coincident topology is merged or shared. Merging requires modifying the topology of the parts. Merging makes both adjacent parts reference the same topology so that when meshing occurs on merged entities, both parts reference the same mesh and the global mesh between the two parts is always conformal. Imprint and merge can be done simultaneously or separately. If a third-party geometry package is used for imprinting, the merge step must be done after all imprinting is complete, making it necessary to recompute the coincident faces in the model. If the imprinting is controlled locally, imprint and merge can be done at the same time. Both forms of imprinting and merging or mesh merging have their advantages and disadvantages. *Imprint and merge,* however, is a cleaner and less confusing interface to the user and is therefore preferred by most meshing packages.

1.2 Imprinting

Mesh generation programs generally obtain their imprints either in an external geometry package or through face-face intersections. An external geometry package provides the entire imprint operation and is self-contained. When using face-face intersections, meshing packages must generate the imprint after receiving the intersection graph of two faces.

1.2.1 External Geometry Package

For meshing packages, the most common source of the imprint Boolean is the underlying geometry engine. This engine can be created in-house or as a third-party package. The engine performs CAD modeling and interpretation for the meshing package and usually contains some form of the imprint Boolean at the part or surface level. The authors are most familiar with the ACIS 3D modeling toolkit [6]. Imprinting in this system is done at the part level. Imprints performed in a third-party package are only as accurate as the geometry package (1e-6 for ACIS) and provide little control of the result from the imprint. Most packages allow the tolerance to be changed; however, often the models become unstable and unpredictable after performing operations with a larger tolerance [7].

1.2.2 Face-Face Intersections

A second option for meshing packages is to perform face-face intersections to generate the imprint Boolean. These can be performed either through a third-party package or a home grown geometry system such as a facet-based approach [8, 9]. With the result of the intersection graph from the face-face intersection, the meshing package can selectively modify the graph and apply it to the imprint targets. This requires either modifying the geometric model itself or using "virtual geometry" [5, 10] layered on top of the model to represent topology changes. Three roadblocks can discourage taking the intersection graph from face-face intersections and creating an imprint. They are:

i. Geometry package must be able to support radically tolerant topology.

ii. Virtual geometry prevents further part modifications through a third-party package.

iii. *Tolerant* surface-surface intersections are unreliable and difficult to compute.

Roadblock "*i*" occurs if the topology modifications from the imprint are large enough. In this case, the geometry engine may cease to function properly for the modified parts. For instance, to fix the interactions of the parts shown in Figure 2, topology must be moved a distance of .1 from the original location. The effect of this adjustment on the geometry engine may be terminal when additional intersections or operations are performed on the part. This roadblock can be overcome by using a system like virtual geometry rather than a CAD engine.

Roadblock "*ii*" can occur if virtual geometry is used to modify the parts. While virtual geometry handles the tolerance issues from the first roadblock, a second problem occurs when trying to modify the same part with additional imprints from other parts. By definition, the virtual geometry modifications do not interact with third-party geometry engines. If the geometry engine is homegrown or facet based, the third roadblock may come into play.

Roadblock "*iii*" is that the intersections required for imprinting need to be calculated with a geometric tolerance that can often be large with respect to features within the faces. With mesh generation, the geometric representation only needs to be good enough for the given mesh resolution. For some parts, that can be several orders of magnitude different from the tolerance at which the part is defined geometrically. Additionally, tolerant intersections can be difficult to compute. For example, if the intersection is performed by faceted representations, polyhedron may be required to represent the facets tolerantly. For more exact surface representations the results of changing the tolerance can be unpredictable.

Based on the previous discussions of conformal meshing and imprinting the next section describes the new imprint and merge algorithm.

2 BOUNDARY IMPRINTING

The main topic of this paper involves a new method for imprinting. The algorithm uses the boundary edges of the faces rather than actual face-face intersections. This method avoids all three of the roadblocks from the previously discussed face-face intersection imprint option (section 1.2.2), and the problems with using an external geometry package (section 1.2.1). Boundary imprinting uses virtual geometry to support radically tolerant topology. Curve intersections are easy to implement in meshing packages that rely on third-party geometry engines. They can be used with virtual geometry without preventing further modifications. Finally, tolerant boundary intersections can be computed reliably and easily.

There is some functionality lost when solely utilizing boundary intersections. With boundary or curve intersections, the only imprints created will be those between faces that have edge or vertex topology near the intersection locations. The boundary imprint will not affect intersecting faces when the intersection occurs only interior to the faces and away from the edges of the faces. Three cases that will not imprint in boundary imprinting have been identified and are shown in Figure 4. Extreme overlap is shown in Figure 4 (a), where the overlap is beyond the tolerance of the imprint operation. Figure 4 (b) shows two spheres where the faces intersect internally but no lower order boundaries intersect. Case (c) shows a situation where the two faces intersect at both the boundaries and through a degenerate line intersection that occurs between the inner portions of the faces. While these situations may be significant for other imprint uses, they are not important for conformal meshing. All of these cases can be handled by real face-face intersection based imprinting.

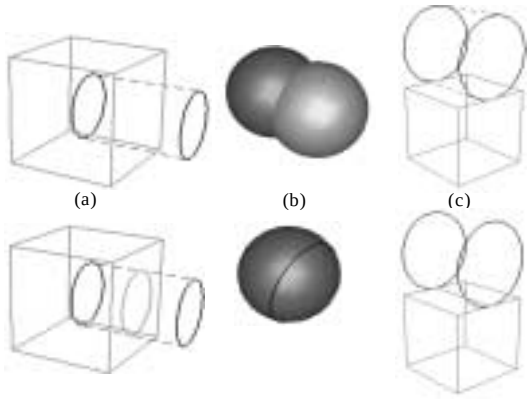

Figure 4 Boundary Imprint Cases Not Handled

Boundary imprinting is an alternative method for imprinting two adjacent faces. The term "boundary" refers to the boundaries of the faces, or in other words, the edges. The goal of boundary imprinting is to tolerantly and robustly achieve the results of an imprint Boolean between two faces. This section discusses how that goal is achieved. The new imprint algorithm consists of the following five steps:

1. *Discretize the Boundary*
2. *Intersect Line Segments*
3. *Build Partial Intersection Graph*
4. *Construct New Edges*
5. *Split Old Edges and Faces and Merge New Faces*

For steps 4 and 5 the algorithm relies on the underlying geometry engine to create edge topology and geometry from line segments and to tolerantly connect edges. Additionally, step 5 requires the geometry engine to be able to split or *partition* a face given a set of edges that splits it. The authors used the CUBIT Mesh Generation Toolkit [11, 12] for this work, where all of these prerequisites are available through the Common Geometry Module or CGM [13]. Within the CGM, a virtual geometry capability is available that allows local topological modifications and is able to generate curve geometry based on segmented data. Kraftcheck describes the virtual geometry capability [10].

In addition to two faces, the desired tolerance must be supplied to the boundary imprint algorithm. This absolute tolerance is used to determine whether entities during the imprint are touching. Choosing the tolerance is important. The tolerance determines which features in the imprint are resolved and which are ignored. For this reason, the algorithm assumes that the user will supply the tolerance. Examples in section 4.1 demonstrate how the tolerance affects the imprint results.

2.1 Discretize the Boundary

The first step in the boundary imprint algorithm is to approximate the boundaries of the two input faces. Rather than calculating the exact intersections between the boundary edges, which may be different for different geometry representations, the boundary is approximated by line segments and points. This step can be accomplished by either using the graphics faceting or by creating a new set of segments similar to edge meshing.

The discretization needs to ensure that the segments are not smaller than the input tolerance. If they are smaller than the tolerance, instability will be introduced in the intersection step of the algorithm. A discretization of approximately 2.5 times larger than the input tolerance was seen to ensure stability during the intersection step.

The line segments and points that are created during this step are used to store information about the intersection in step 2, which will be used later in steps 3, 4 and 5. The following three data values are stored on the segment points:

- *Original Topology Owner* (Vertex or Edge)
- *Point Type Classification*
- *Matching Point*

The *original topology owner* value is set to point to the edge or vertex to which the point approximates.

The *point type classification* value is an enumerated integer value that can have the following values:

- ON_BOUNDARY_1
- ON_BOUNDARY_2
- VERTEX_ON_BOUNDARY_1
- VERTEX_ON_BOUNDARY_2
- CREATE_NEW_VERTEX
- ON_BOTH_BOUNDARIES
- VERTEX_ON_BOTH_BOUNDARIES

Originally, this value is set according to its original topology owner. For example, if the point is on the first face, and it represents a point on one of the boundary edges, the *point type classification* value of the point would be set to ON_BOUNDARY_1.

The *matching point* value is initially left unset and does not get set until the point is determined to be spatially equivalent to a point on the other face.

The line segments also carry the original topology owner information. The line segment is additionally used as a doubly linked list, and therefore carries pointers to the next and previous line segments on the boundaries. A linked list structure is used rather than an array based list because of the number of insertions and deletions that occur during the intersection process. The linked lists allow local modification at a performance cost of constant time O(C). The list information is important to track since it will be modified during step 2 and used in each of the following steps. The head of each segment loop on each boundary is then stored in an array. Once the point and segment data has been obtained, the next step is intersecting the segment loops of the two faces.

2.2 Intersect Line Segments

Each line segment is intersected with the segments closest to it from the other face. This can be done efficiently by using an R-Tree data structure, which is discussed in section 3. Alternatively, the intersections can be done using a brute force approach which has O(N^2) running time where N is the total number of line segments.

Since the line segments are three-dimensional, the segments may not actually intersect. The input tolerance is used to place a "hotdog" shaped buffer zone around each segment. Overlaps in the "hotdog" space between segments constitute an intersection. Overall, there are 27 ways the segments can interact. These 27 cases are shown in Figures 5, 6, 7, 8, 9 and 10.

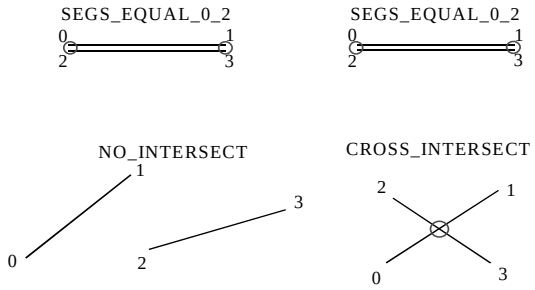

Figure 5 Equal, No Intersect and Cross Intersect Cases

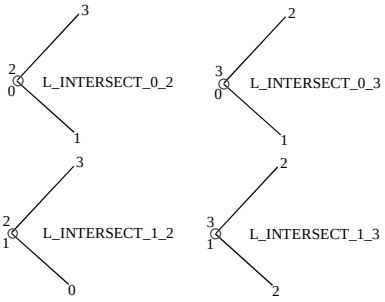

Figure 6 L Shaped Intersection Cases

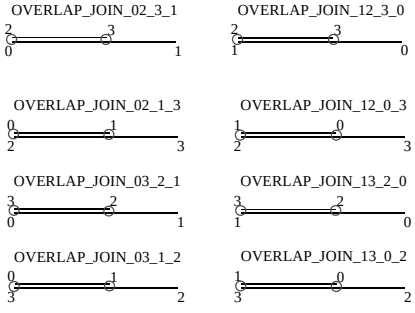

Figure 7 Joined and Overlapping Cases

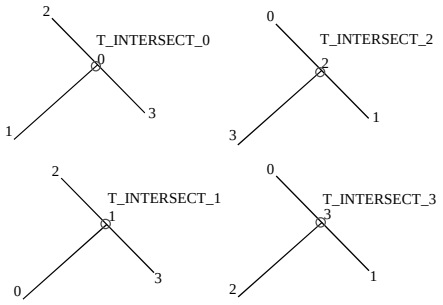

Figure 8 T Shaped Intersection Cases

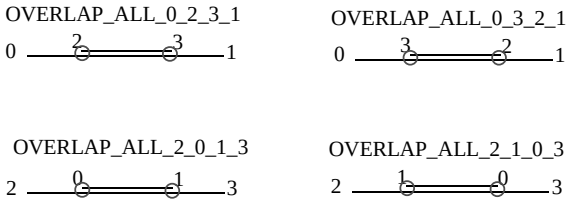

Figure 9 Total Overlap Intersection Cases

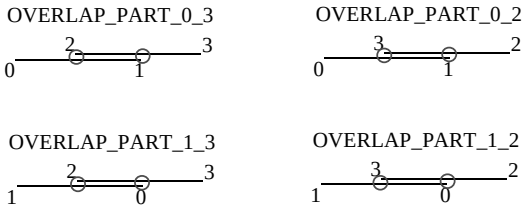

Figure 10 Partly Overlapping Intersection Case

To determine which intersection case is occurring, the end points are first checked to see if they are spatially within the input tolerance. If any of the points are equal, the intersection type can be one of three cases SEGS_EQUAL (Figure 5), L_INTERSECT (Figure 6) or OVERLAP_JOIN (Figure 7). If this situation occurs, points that are spatially equivalent are marked as matching points. If one of the points was originally owned by a vertex, the point not owned by a vertex has its point classification value set to CREATE_NEW_VERTEX. If both their owners were originally vertices, they are both marked as VERTEX_ON_BOTH_BOUNDARIES. If vertices owned neither of the two, then it cannot be determined at this time if the boundaries are actually intersecting or overlapping. The point types are then marked as ON_BOTH_BOUNDARIES.

If just two of the points are equivalent, then the intersect case must be an OVERLAP_JOIN or a L_INTERSECT. The points that are not equivalent are then tested to see if they are

geometrically "on" the other line segment. This is calculated by first computing the closest point from that point to the line segment. The closest point computation is described in [14] and is also described here as follows.

Two line segments, $\{P_0,P_1\}$ and $\{P_2,P_3\}$ are shown in Figure 11. It is to be determined if the point P_3 lies "on" the segment $\{P_0, P_1\}$. The closest point, Q, to P_3 on the segment $\{P_0, P_1\}$ is found as:

$$Q = P_0 + b * (P_1 - P_0) \tag{1}$$

where:

$$b = \left(\frac{(P_3 - P_0) \bullet (P_1 - P_0)}{|P_1 - P_0|} \right) \tag{2}$$

Because these equations apply to lines, *b* is restricted to lie within the domain (0,1). If *b* is outside this domain the following conditions are applied:

$$\begin{aligned} b \le 0, &\longrightarrow Q = P_0 \\ b \ge 0, &\longrightarrow Q = P_1 \end{aligned} \tag{3}$$

The point (P_3) is considered to lie on the segment $\{P_0, P_1\}$ if the new point (Q) is within tolerance to it. If the point P_0 or P_1 lies within tolerance of point P_2, then the OVERLAP_JOIN case results and the new point Q is used to split the segment. The new point, Q, and point P_3 are marked as either ON_BOTH_BOUNDARIES, or CREATE_NEW_VERTEX depending on the topology owner of P_3. If the points P_3 and Q are not within tolerance of each other, the L_INTERSECT case results and the points that are equivalent are marked as ON_BOTH_BOUNDARIES or CREATE_NEW_VERTEX, depending on the topology owners of both points.

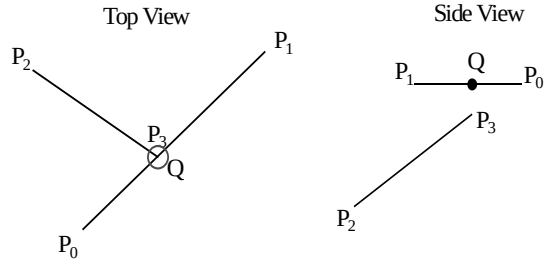

Figure 11 Closest Point on Segment

If none of the segment end points are spatially equivalent, then all the points are tested to see if they lie "on" the segment. This is computed through equations (1), (2) and (3) and shown in Figure 11. If any of the points are on the other segment, the result is T_INTERSECT (Figure 8), OVERLAP_PART (Figure 10) or OVERLAP_ALL (Figure 9). These cases are distinguished by simply knowing which points intersect with which segments. For the T_INTERSECT case, the point (P_3) and the new point (Q) are both set to have point classification types of CREATE_NEW_VERTEX. This is known because the T_INTERSECT case can only occur if the boundaries are truly intersecting and not overlapping. For OVERLAP_PART and OVERLAP_ALL, the point classification types cannot be fully resolved similar to previously discussed cases. For any point (Q) that is found, the line segment on which it lies ($\{P_0, P1\}$) is split into two new segments. The linked list data is adjusted accordingly.

Finally, if none of the points are on the other test segment, and if none of the points are equivalent, the lines are tested to be crossing as shown in Figure 5. The intersection is calculated by determining the closest point on each line segment. Finding the intersection between two segments is described in [14,15] and is now restated here in terms of tolerant intersections.

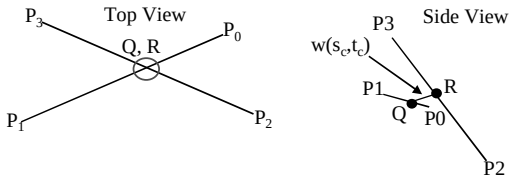

Figure 12 Closest Points Between Two Segments

For the two segments $\{P_0, P_1\}$ and $\{P_2, P_3\}$ shown in Figure 12, compute the closest points, Q and R, between them. The parametric equations (4) and (5) for each line segment are given as:

$$\begin{aligned} L_1(s) &= P_0 + (P_1 - P_0) * s \\ L_2(t) &= P_2 + (P_3 - P_2) * t \end{aligned} \tag{4}$$

The distance between two points, Q and R, on these lines is given as:

$$w(s_c, t_c) = L_2(t_c) - L_1(s_c) \tag{5}$$

Minimizing this equation and solving for the variables s_c and t_c (the parametric locations of the points Q and R) yields the following equations that can be used to solve for Q and R as:

$$s_c = \frac{b * e - c * d}{a * c - b^2} \text{ and } t_c = \frac{a * e - b * d}{a * c - b^2} \tag{6}$$

where:

$a=|P_1-P_0|$, $b=(P_1-P_0)\bullet(P_3-P_2)$, $c=|P_3-P_2|$, $d=(P_1-P_2)\bullet(P_0-P_2)$

and $e=(P_3-P_2)\bullet(P_0-P_2)$

The parametric values s_c and t_c are restricted to the interval (0,1) in the same way as b in Equation (3).

If the two closest points, Q and R, are within tolerance then the segments are intersecting. The two closest points are then used to split each segment into two new segments. The new points are marked as CREATE_NEW_VERTEX point classification types. If the new points are not within tolerance, then the segments are not intersecting and the next two segments are tested. The intersection process continues until all the segments have been tested.

After the intersection process is complete, the loops of segments are traversed sequentially to resolve the point classification type values for the points with point classification values of ON_BOTH_BOUNDARIES. Once this is finished, the segment loops are ready to be sorted into the partial intersection graph used for splitting the faces.

2.3 Build Partial Intersection Graph

The full intersection graph can now be calculated from the modified segment loops. The full intersection graph is not needed here because some of the information required from the full graph was already calculated during the intersection step. At this point, for each face, the only needed portions of the graph are those that do not overlap the boundary and are geometrically "on" the face. This part of the intersection graph will be used to split the faces. The portion of the graph that overlaps the boundary of the face is usually needed to imprint the vertices across the faces. This has actually already occurred and is stored in the point classification data on the imprint points.

The partial intersection graph for face 1 is obtained by traversing the loops segments of face 2. If a point is found to have a point classification value of CREATE_NEW_VERTEX, then this point may be the start or end of a needed portion of the intersection graph. Whether or not it is the start of a needed portion of the graph is determined by looking at the points prior to and next to it. If the next or previous points are not overlapping face 1 and are within face 1 geometrically, then that portion of the segment's loop is part of the needed intersection graph for face 1. The portion of segments added to the graph will be between two points that have point classification values of CREATE_NEW_VERTEX. This process is continued until all the segment loops of face 2 have been traversed. The process is repeated for face 2 where the loops of face 1 are traversed over face 2. Special cases where the segment loops do not intersect but are on the face are also handled at this point. These special loops are added entirely to the partial intersection graph. After this process, the partial intersection graph for each face can be used to construct new model edges for the faces.

2.4 Construct New Edges

The partial intersection graphs are used to create segmented geometric curves. These curves approximate portions of edges from opposite faces. The segmented curves are assumed to approximate the boundaries with sufficient accuracy for mesh generation. This also assumes that the input tolerance is smaller than the mesh size. The edges represented by segmented curves are created in the geometry engine. Again, for the present implementation this was done using the virtual geometry engine to maintain independence of the underlying modeling kernel. Once the new edges representing the partial intersection graph have been created, the two faces can be split accordingly.

2.5 Split Old Edges and Faces and Merge New Faces

With the new edges defining the intersection graph and the points from the segment loops that define vertex imprints, the actual imprint can occur. The boundary edges are first split or partitioned according to points on the boundary that are marked CREATE_NEW_VERTEX. Once this occurs, the partial intersection graph is merged with the boundary of the face. The partition utility of the virtual geometry engine is then used to split the face according to the graph. This is repeated for both faces, and the boundary imprint operation is complete. The following section describes the imprint process with an example to better clarify the algorithm. After the imprint is complete, new faces that are now completely coincidental, both geometrically and topologically, are merged or consolidated.

2.6 Boundary Imprint Example

A simple example will now demonstrate the imprint process. This example will consider two square faces that overlap. The faces and their trivial discretization are shown in Figure 13. This figure also shows how the point classification type is set for the segment points.

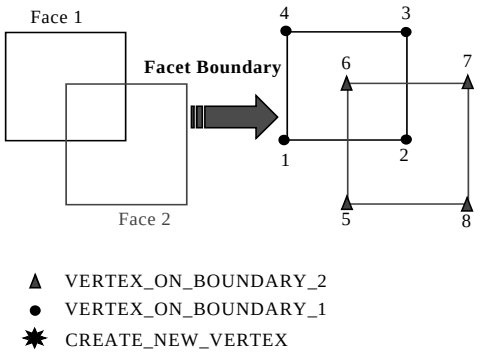

Figure 13 Example: Discretizing the Boundary

Following discretization, the segments are intersected. This process is shown in Figure 14. Here, four new vertices, 9, 10, 11 and 12 are created through the CROSS_INTERSECTION case where segments {2,3} and {6,7} and {1,2} and {5,6} intersect. Nodes 9, 10, 11 and 12 are marked as CREATE_NEW_VERTEX point classification types.

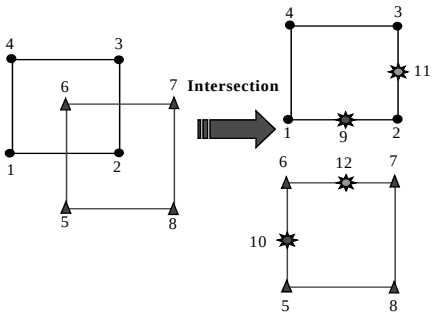

Figure 14 Example: Intersection

Following the intersection step, the partial intersection graph is constructed. The example will follow this process for face 1; the process for face 2 is similar. The partial intersection graph is obtained in Figure 15. For face 1, the partial intersection graph consists of segments {12,6} and {6,10} from the boundary of face 2.

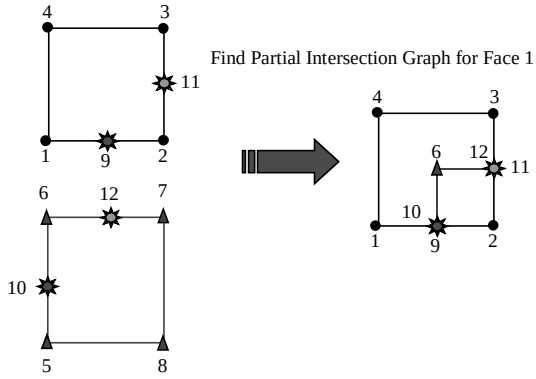

Figure 15 Example: Partial Intersection Graph

Once the partial intersection graph is found, new edges can be created. For this example, the edges are simple; they are each one line segment. For more realistic examples there can be many segments for each new edge. Once the edges are created, they are used to partition face 1 into two new faces. This process is shown in Figure 16.

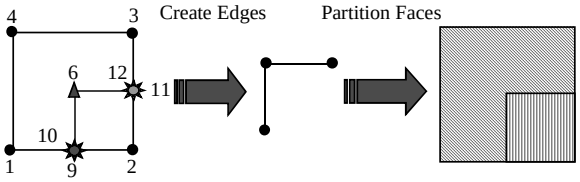

Figure 16 Example: Edge Creation and Partitioning

This example shows how the new boundary imprint algorithm works for two simple faces. For more complex faces the boundary imprint algorithm calculates tolerant, real, 3D face boundary intersections. Tolerant intersection calculations are used to allow loosely overlapping surfaces to conform. When large assemblies and complex surfaces are present, the described algorithm uses a spatial tree to improve performance. This is now discussed in the following section.

3 R-TREES IN BOUNDARY IMPRINTING

Two performance issues arise in performing boundary imprints. The first is in the imprint process itself. During the intersection step of the imprint process, the line segments must be intersected against other close segments from a different face. The trivial method is to intersect every segment on face 1 with every segment on face 2. This method yields an O(M*L) run time performance where M is the number of segments in face 1 and L is the number of segments in face 2.

The second problem occurs when there are many faces to be imprinted in the model. Here the trivial method will produce a run-time performance of O(N^2) where N is the total number of faces in the model. For both of these cases, the time results can be tedious to users.

An R-Tree developed by Guttman [16] is used to overcome these problems. The authors found the R-Tree in practice to have a run-time performance of O(N) for the second problem and O(H) for the first, where H is the average number of segments in both faces. The R-Tree is a height-balanced, multi-dimensional, dynamic, spatial access tree. Unlike other spatial trees like octrees or kd-trees, the R-Tree supports multi-dimensional data like three-dimensional faces and segments. The R-Tree approximates this data with close fit bounding boxes.

The R-Tree works by inserting each entity into the tree. The entity is placed in a leaf node with sufficient space. The leaf nodes have a number of data nodes between size *m* and *M*. For this implementation *m* was chosen to be two, and *M* to be five, though these values may not be optimal. As the entity is inserted, the bounding boxes of the leaf node and parent node are updated to contain the new entity. If there are no spaces left to insert new data, the leaf node with the closest bounding box is split in half. The splitting portion of the R-Tree is the most significant part of the algorithm because it is this phase that ensures the balanced nature of the tree. Guttman proposed several approaches for node splitting and determines the quadratic split approach to be optimal. The quadratic split is used here. The quadratic split minimizes the waste of the bounding boxes by the two new covering nodes. Other trees such as the R*-Trees [17] may achieve a higher optimality, though such approach was not attempted here. Node deletion is performed in a similar way to insertion. Searching the tree is similar to searching any binary search tree [18]. The one caveat to the R-Tree, and the reason only O(N) is achieved in practice and why O(N^2) can occur, is that during the search, *all* children that overlap with the search space must be traversed.

The memory required for the R-Tree is $O(\log_m N)$, where *m* is the minimum size of the children (two for our case) and N is the number of entries.

In practice the R-Tree performs very well. In Figure 17 a graph is shown comparing the trivial method versus the R-Tree for face merging, which is similar to the face-face imprinting problem.

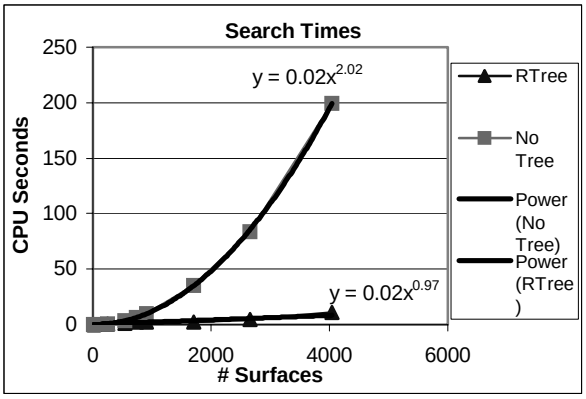

Figure 17 R-Tree Performance

The R-Tree effectively decreases the run time for the new imprint and merge algorithm. The algorithm is still slightly slower than commercial geometry packages that provide imprinting; however, this is probably due to the relative newness and non-optimized aspects of the algorithm. The improved results significantly outweigh this drawback. Additionally, an R*-Tree which claims a 50% speed improvement over the R-Tree could be implemented if this problem remains an issue.

4 CONCLUSION

Boundary imprint and merge can be used to ease the often-tedious chore of conformal meshing. The tolerant process of boundary imprinting improves the solid model for meshing where parts are not aligned properly. Several examples of this are now shown as part of the results of this work.

4.1 Results

The purpose of imprinting and merging during the meshing process is to obtain a conformal mesh. Often, this process is difficult due to poorly aligned parts and CAD operator error. The following examples demonstrate how boundary imprinting improves this process.

The first example, shown in Figure 18, shows the results of imprinting the two parts in Figure 2 with both boundary imprinting and with the ACIS geometry engine. The results are then meshed conformally. Figure 18 indicates how imprinting affects the quality of the mesh. With the ACIS imprint a ledge is maintained where the parts are misaligned while the tolerant boundary imprinting removes that ledge from the model.

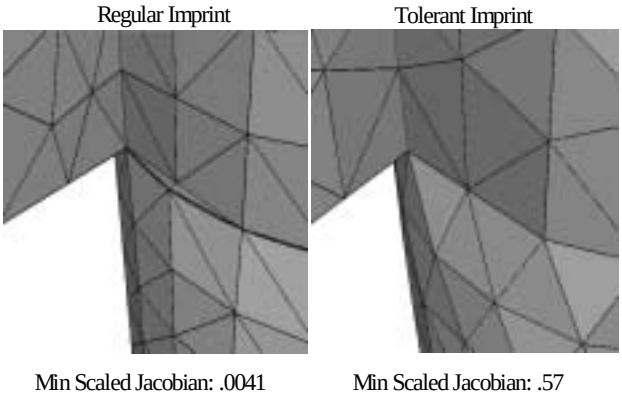

Figure 18 Example 1 of Tolerant Imprinting vs. Normal Imprinting

The next example, shown in Figure 19, more explicitly shows how the face topology is modified differently for tolerant imprinting. Here, two cubes of different sizes are placed next to each other. The top face of the smaller cube is slightly (0.05 units) below the top face of the larger cube. The parts are imprinted with the ACIS geometry engine and the tolerant boundary imprint algorithm with an input tolerance of 0.5. The new algorithm produces topology such that the top face of the large surface is connected through an edge to the top face of the smaller face. The ACIS imprint produces a small gap between the two top faces. This example demonstrates how imprinting affects slightly misaligned parts. Despite the misalignment, the tolerant imprint algorithm produces a clean connection between the two parts.

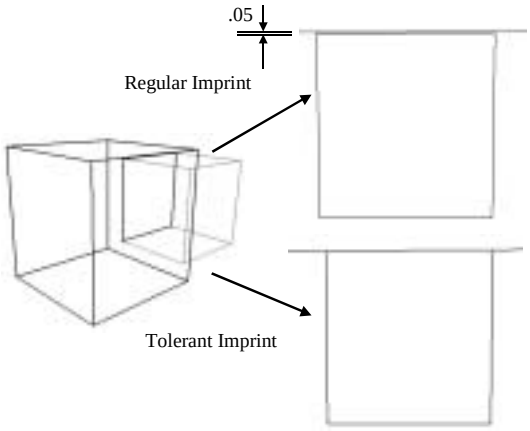

Figure 19 Example 2 of Tolerant Imprinting vs. Normal Imprinting

The parts were mesh with hexahedral elements with a size of 1.0 in the CUBIT Mesh Generation Toolkit. A zoom in of the top faces is shown in Figure 20. The parts that are imprinted with the ACIS imprint cannot be meshed with acceptable quality without reducing the mesh size. Figure 20 also shows how the mesh conforms to the tolerant shape.

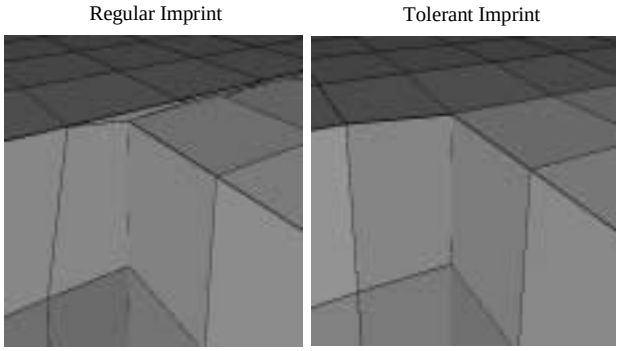

Figure 20 Meshes of Example 2

The next example shows how the boundary imprint tool can be used to aid meshing of parts that meet tangentially. This example is seen in Figure 21. This example shows two parts, a brick and a cylinder, stacked on top of each other. A square face of the brick and circular face from the cylinder are adjacent. The regular imprinting that is done with ACIS produces four faces that contain edges that meet tangentially. Tolerant boundary imprinting also produces similar four faces, but the topology and geometry of the edges that bound the faces is modified such that no edge meets tangentially. Again, the quality of the resulting mesh is shown to emphasize the improvement of boundary imprinting.

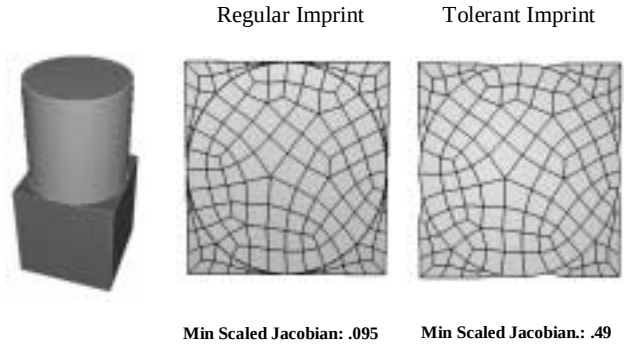

Figure 21 Example 3 of Tolerant Imprinting vs. Normal Imprinting

The final example demonstrates the robustness of the imprint boundary algorithm. The example has two faces from parts in an assembly used at Sandia National Laboratories. The faces, shown in Figure 22, are roughly on top of each other, but the definition of the faces makes them poorly aligned, as shown in the "zoom-in" pictures of Figure 22.

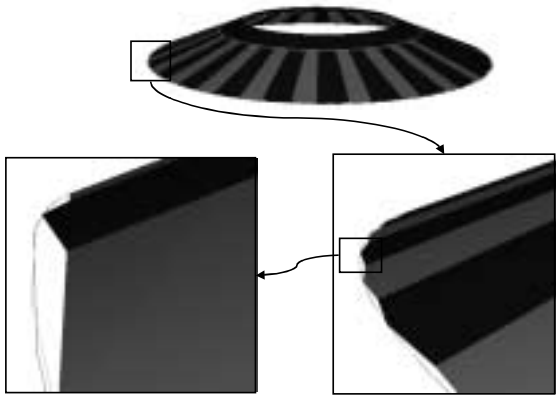

Figure 22 Poorly Aligned Surfaces (Courtesy of SNL)

Figure 23 shows the results of the imprinting process using the ACIS geometry engine for imprinting, a facet-based intersection algorithm from Los Alamos [19], and boundary imprinting. Both the ACIS and the facet-based imprints fail because of the ill-aligned/defined faces. The boundary algorithm successfully imprints the two faces.

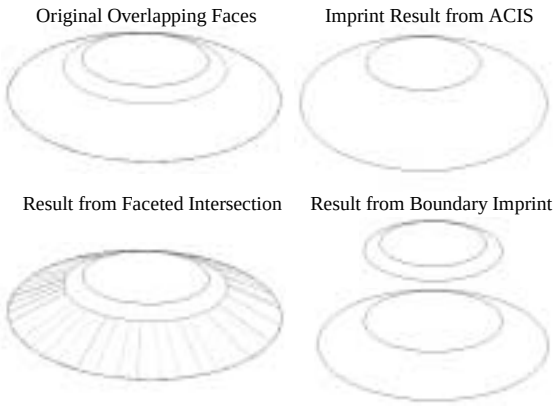

Figure 23 Results of Imprinting Poorly Overlapping Faces

The boundary imprint algorithm works well to clean up misaligned and poorly defined adjacent parts for conformal meshing. The algorithm takes discretized segments of the boundary edges and intersects them to produce an intersection graph to split overlapping faces. The calculations are performed tolerantly to ensure that overlaps are properly maintained and that only the intended face splitting occurs. Utilizing the R-Tree to find close faces and segments enhances the efficiency of the boundary imprint algorithm.

4.2 Future Research

The major contribution of the boundary imprint algorithm is the ability to simply calculate tolerant intersections for three-dimensional faces. The boundary imprint algorithm does not do this exactly because only the edges are taken into consideration. Future research could include the following options:

- Calculate automatic tolerance
- Improve facet-based intersections to calculate tolerant intersections
- Improve exact intersections to calculate tolerant intersections

If the intersection graph can be calculated tolerantly in either of the two previous methods, results similar to the boundary imprinting code could be shown. Additionally, using a virtual geometry capability to perform the actual imprint of the intersection graph would allow the graph to be cleaned for meshing purposes. This is a major benefit to implementing the imprint operation within the meshing package itself. Any future research should continue to allow this modification from within the meshing package, where the problems with mesh generation are best known.

5 REFERENCES

[1] White, D. R., R. W. Leland, S. Saigal, and S. J. Owen, "The Meshing Complexity of a Solid: An Introduction", *Proceedings, 10th International Meshing Roundtable,* Sandia National Laboratories, pp.373-384, October 7-10 2001.

[2] Tautges, T. J., "Automatic Detail Reduction for Mesh Generation Applications", *Proceedings, 10th International Meshing Roundtable*, Sandia National Laboratories, pp.407-418, October 7-10 2001.

[3] Mezentsev, A., "Methods and Algorithms of Automated CAD Repair For Incremental Surface Meshing", *Proceedings, 8th International Meshing Roundtable,* Sandia National Laboratories, pp. 299-309, October 1999.

[4] Mobley, A. V., M. P. Carroll, and S. A. Canann, "An Object Oriented Approach to Geometry Defeaturing for Finite Element Meshing", *7th International Meshing Roundtable,* Sandia National Laboratories, pp. 547-563, October 1998.

[5] Sheffer, A., T. Blacker, J. Clements, and M. Bercovier, "Virtual Topology Operators for Meshing", *Proceedings 6th International Meshing Roundtable,* Sandia National Laboratories, pp. 49-66, October 1997.

[6] http://www.spatial.com, May 2002

[7] Jackson, D. J., "Boundary representation modeling with local tolerances", *Proc. of the Third Symposium on Solid Modeling and Applications*, 1995, Salt Lake City, 1995, pp 247-253.

[8] Laidlaw, D. H., W. B. Trumbore, and J. F. Hughes, "Constructive Solid Geometry for Polyhedral Objects", Computer Graphics, 20, 4, 161-170 (1986).

[9] Hubbard, P. M., "Constructive Solid Geometry for Triangulated Polyhedra", Brown University Technical Report No. CS-90-07, 1990.

[10] Kraftcheck, J., "Virtual Geometry: A Mechanism for Modification of CAD Model Topology For Improved Meshability", Published Master Thesis, University of Wisconsin at Madison, December 2000.

[11] Blacker, T.D., "*CUBIT Mesh Generation Environment Users Manual Vol. 1*", SAND94-1100, Sandia National Laboratories, Albuquerque, NM, 1994.

[12] CUBIT Mesh Generation Tool Suite: Automatic Unstructured Hex, Tet Quad and Tri Meshing and Solid Model Geometry Preparation. Web Site: http://endo.sandia.gov/cubit (May 2002).

[13] Tautges, T. J., "The Common Geometry Module (CGM): A Generic, Extensible Geometry Interface", *Proceedings, 9th International Meshing Roundtable*, Sandia National Laboratories, pp.337-348, October 2000.

[14] Bowyer, A. and J. Woodwark, "A programmer's geometry", *Butterworths*, 1983, pp. 47-53.

[15] O'Rourke, J., *Computational Geometry in C*, Cambridge University Press, 2nd Edition, 1998, pp. 220-226.

[16] Guttman, A., "R-Trees: A Dynamic Index Structure for Spatial Searching", *Proceedings, SIGMOD Conference*, Boston, June 1984, pp. 47-57.

[17] Beckmann, N., H.-P. Kriegel, R. Schneider, and B. Seeger, "The R*-tree: An efficient and robust access method for points and rectangles", *Proceedings ACM SIG-MOD International Conference on Management of Data*, 1990, pp. 322-331.

[18] Weiss, M. A., *Data Structures & Algorithm Analysis in JAVA*, Addison-Wesley, 1999, pp. 109-112.

[19] Fowler, J., http://www-xdiv.lanl.gov/x8/oso/