A Software Tool for 3D Meshing of VLSI Interconnect Structures

K.J. van der Kolk and N.P. van der Meijs
Delft University of Technology,
EEMCS, Circuits and Systems group,
Mekelweg 4, 2628 CD Delft, The Netherlands
Email: {keesjan,nick}@cas.et.tudelft.nl

***Abstract*— Due to decreasing dimensions and increasing signal-frequencies, the study of on-chip parasitic effects has become of basic importance. In order to accurately predict the behavior of a VLSI design, it is nowadays necessary to use tools capable of performing detailed field-calculations. A primary requirement for such computations is that the domain is decomposed into a mesh. This paper describes the design and implementation of a software tool capable of generating fully three-dimensional meshes from VLSI layout information. The main features of the tool are that the basic elements are tetrahedral, the generated elements satisfy a predetermined quality condition, and the meshes exhibit good grading.**

I. Introduction

As chip dimensions decrease and signal-frequencies rise, the parasitic coupling between on-chip interconnect becomes more and more important. Therefore, designers of VLSI circuits become increasingly dependent on automated tools to inspect if what they designed will actually have the desired physical behavior. Since several decades, many publications have appeared on the topic of parasitic coupling and its efficient computation (see, e.g., [1]). To actually incorporate such techniques into software, one needs to build appropriate data-structures describing the geometric details of the interconnect. A so-called boundary representation (b-rep) may not be sufficient, and what is often needed is a two- or three-dimensional mesh of the interconnect and possibly the surrounding structures, including the substrate.

In this paper, we describe a software tool we developed for converting arbitrary VLSI layouts into a three-dimensional tetrahedral quality mesh. This implies that only tetrahedra (or 3-simplices) are used as the building block of our meshes. Further, each tetrahedron is guaranteed to satisfy some quality condition.

This paper is structured as follows. First, we make a brief reference to the mesh-generation literature on which our work is based (unfortunately the theory is too extensive to replicate here). Then, we discuss the implementation-details of our mesh-generator, where it will become clear that, although the underlying theory of Delaunay-based mesh generation is gaining maturity, robust implementation is still far from trivial. Finally, we describe the implementation of a front-end which renders our mesh-generator suitable for handling VLSI structures.

II. Delaunay Refinement

Our mesh-generator is based on techniques from the theory of Delaunay refinement, that evolved in recent years. Limitation of space restrains us from publishing a full account of this theory, and we refer to [2], [3], and [4] for expositions of the matter.

Figure 1 shows the Delaunay refinement algorithm in the form of pseudo-code. The functions SPLIT_1, SPLIT_2, and SPLIT_3 perform the operations of segment, subfacet and tetrahedron splitting, respectively. The function $\gamma(t)$ gives the circumradius-to-shortest-edge ratio of a tetrahedron t. A tetrahedron t with $\gamma(t) > B$ has an unfavorable circumradius-to-shortest-edge ratio and is called *skinny* (such tetrahedra will be eliminated).

When one chooses $B > 2$, and all inter-edge and dihedral angles in the input domain are greater than, or equal to $\pi/2$, then the algorithm is guaranteed to terminate ([3]).

III. Geometric Predicates and Robustness

Geometric predicates form the necessary link between the geometric information and topological information present in the mesh. Geometric and topological interpretations should always be mutually consistent: for example, a point which is geometrically in the interior of a triangle (in two dimensions) must not simultaneously be considered to lie on its outside on the basis of topological inspection.

In our implementation, only two geometric predicates are needed. One predicate, Orient3d, determines the relative orientation of four points (in R^3). Another predicate, InSphere, determines, given four points (in R^3) p_1, p_2, p_3, p_4, whether a given fifth point p_5 lies inside or outside the circumscribed sphere of the other four points. Both predicates can be computed by the evaluation of a determinant. See [4] for details on these predicates. Note that the two-dimensional variants of these predicates are not needed in our implementation (for example, all operations on subfacets are performed using three-dimensional predicates for robustness, see Section IX).

Without additional care, both predicates can run into degenerate situations. In the case of Orient3d, a degeneracy occurs if the four given points lie exactly in a plane. In the case of InSphere, a degeneracy occurs if the five given points lie exactly on a sphere. For both predicates, the underlying determinant evaluates to zero in case of a degeneracy. Because

```
01:    while True:
02:        (STEP 1) if some subsegment s is encroached:
03:            SPLIT_1 s
04:        else (STEP 2) if some subfacet u is encroached:
05:            let c be the circumcenter of u
06:            if c encroaches upon a subsegment s:
07:                SPLIT_1 s
08:            else:
09:                SPLIT_2 u
10:        else (STEP 3) if there exists a tetrahedron t with γ(t) > B:
11:            let c be the circumcenter of t
12:            if c encroaches upon some subsegment s:
13:                SPLIT_1 s
14:            else if c encroaches upon some subfacet u:
15:                SPLIT_2 u
16:            else:
17:                SPLIT_3 t
18:        else:
19:            break
```

Fig. 1. Pseudo-code for the three-dimensional mesh-refinement algorithm.

degeneracies would greatly complicate the implementation of the algorithm, we adopted the symbolic perturbation scheme by Edelsbrunner and Mücke, named Simulation of Simplicity (SoS) [5]. Essentially, this scheme perturbs the points symbolically, such that the determinants underlying the predicates can never evaluate to zero.

The SoS method is in principle quite inefficient, since it relies on exact arithmetic, and therefore we apply it only in case of actual degeneracies. For the conventional evaluation of predicates, we use Shewchuk's adaptive floating point library [6]. In order to decrease the probability of degeneracies, we also apply a slight physical perturbation to the points which are inserted into the mesh. As a consequence, degeneracies are, in practice, quite rare, so that the efficiency of our SoS implementation is not very important. In fact, the optimizations proposed in [5] are only marginally relevant in our case.

IV. Delaunay Triangulations and Tetrahedrizations

In a three-dimensional Delaunay refinement scheme, the mesh starts as a Delaunay tetrahedrization (DT) [7] of the vertices in the input, which is supposed to be a piecewise linear complex (PLC). (We use the abbreviation "DT" both for "Delaunay tetrahedrization" and "Delaunay triangulation," but we add the prefix "2d" or "3d" if confusion may arise.) Eventually, i.e., in the final mesh, all facets should appear as a union of (triangular) faces of tetrahedra, since facets typically form the domain-boundaries, with different materials on both sides. We say then that the mesh *conforms* to the domain-boundaries. However, in the initial DT, this is not the case (in general), and tetrahedra may 'pierce' through facets. Therefore, points are added to the DT until all segments and facets of the input PLC are represented in the DT as the union of tetrahedron-edges and tetrahedron-faces respectively (according to Figure 1).

The incremental insertion of a point to the DT is implemented using the Bowyer-Watson scheme [8], [9]. Essentially, when a point p is added to a DT, we first find the so-called Bowyer-Watson polyhedron, which is the union of tetrahedra containing p in their circumscribed sphere. Starting from a tetrahedron t containing p (t is clearly in the polyhedron), the full polyhedron is computable by a simple breadth-first search (an elegant proof of this fact can be given along the lines of the proof of the "acyclicity lemma" given in [3]). After collecting the tetrahedra which form the polyhedron, we remove them, and we add a new tetrahedron between p and each of the triangles at the boundary of the formed cavity. The resulting complex is guaranteed to be a consistent Delaunay tetrahedrization, and it is unique due to the exclusion of degeneracies. Note that this point-insertion scheme indeed requires exclusively the two geometric predicates described in the previous section.

The algorithm manages one large 3d DT for the overall mesh, and one 2d DT for every facet, so that the current set of subfacets is always known. In 2d, point-insertion is handled similarly as in 3d. However, for efficiency, we remove the exterior subfacets from the triangulations at STEP 2 of the algorithm (see Figure 1), since at this step, all subsegments are guaranteed to be in the mesh. The triangulation is therefore a kind of constrained Delaunay triangulation (CDT) [3], where the boundary edges are the constraining ones. (Note that in

three dimensions, a concept similar to that of a CDT does not exist [4], and therefore we leave external tetrahedra in place.)

The Bowyer-Watson algorithm (in 2d) can be shown to work also in the case of constrained boundary edges. However, the insertion-polygon does not necessarily contain all triangles which have the insertion point p in their circumcenter. Basically, the insertion-polygon should be computed from a breadth-first search starting at the triangle t containing p, and the search should stop at the boundary of the domain. After the insertion-polygon is found, the insertion procedure continues by removing the corresponding triangles, and connecting the new point p to the edges of the polygon (in a similar fashion to the 3d case).

V. Elementary Data-structures

In this section we describe the elementary data-structures used in the implementation. The data-structures are chosen such that all operations can be performed by "local" inspection or modification of the mesh, which is of course important for efficiency.

For every node in the mesh, a unique node-object is created which stores the respective x, y, z coordinates. Other objects, when referring to a particular node, contain a pointer to the corresponding node-object. Node-objects in the 3d DT and 2d DT's are shared. The address of a node-object is used as its "perturbative-index" in the SoS scheme [5] (in our implementation, the address of an object remains fixed after it has been created).

A subsegment-object contains two pointers to node-objects. During execution, a subsegment g is "attached" to a subfacet s if and only if s contains both nodes of g. Every subsegment-object has associated with it an (arbitrarily large) set of subfacets to which it is attached (these subfacets are called its "wings.") Furthermore, each subsegment g, when it lies in a facet f but is not attached to a subfacet of f records a pointer to a subfacet in f which is topologically "close" to g.

A subfacet-object contains three pointers to node-objects. Furthermore, it contains three pointers to its neighbors (a pointer is simply *null* if there is no neighbor at a certain edge of the subfacet). As a local optimization, each subfacet also stores the ordinal number of the abutting edge of each of its neighbors. During execution, a subfacet s is "attached" to a tetrahedron t if and only if t contains all nodes of s. Every subfacet-object records a set of (at most two) pointers to abutting tetrahedra. Every subfacet also records a topologically "close" tetrahedron, when not attached. Furthermore, a subfacet contains pointers to attached subsegments (detached subsegments are not recorded).

Each tetrahedron contains four pointers to node-objects, and potentially four pointers to its neighbors. As a local optimization, each tetrahedron also stores the ordinal of the abutting face of each of its neighbors, as well as the orientations of those faces. Furthermore, the tetrahedron stores up to six pointers to subsegments which are attached to it (during execution, a subsegment g will be attached to a tetrahedron t if and only if t contains both nodes of g).

Attaching and detaching subfacets to or from their subsegments is done when a point is inserted into the corresponding 2d DT. First, all attachments between subsegments and triangles in the Bowyer-Watson insertion-polygon are broken. Detached subsegments are stored in a (global) table, implemented as a hash-table. Then, each edge of each triangle created by the insertion is searched in the table, and if found, the corresponding subsegment is (re)attached. After the procedure, detached subsegments may remain in the table, and they will be eventually attached during insertion of some other point.

Attaching and detaching tetrahedra to or from their subfacets is done similarly; as is attaching and detaching tetrahedra to or from their subsegments.

VI. Encroached Elements

As Figure 1 shows, the algorithm needs to query the currently encroached subsegments and subfacets. Furthermore, it needs to query whether a given point c, which is not (yet) in the mesh, encroaches upon a subsegment or subfacet. Both operations are quite different. Recall that a subsegment s is encroached upon by some node c iff c lies in the diametral sphere of s (i.e., the smallest sphere containing s); similarly, a subfacet f is encroached upon by some node c iff c lies in the equatorial sphere of f (the smallest sphere containing f).

First consider the case in which we need to pick any encroached subsegment currently in the mesh (at STEP 1 of the algorithm). To implement this operation efficiently, we keep a list of "possibly encroached subsegments." Each time a new node is inserted into the 3d DT, we determine the subsegments which are possibly encroached by this node and insert them into the list (by pointer). Fortunately, we can simply add all subsegments to the list which become detached by the insertion (but note that consequent attaching will not automatically remove them from the list). Thus effectively, we consider only those subsegments which are attached to tetrahedra in the insertion-polyhedron. Now, when looking for *any* encroached subsegment, we take one subsegment from the list and see if it is still encroached (this test can be performed by inspecting only the tetrahedra which share both nodes of the subsegment); if it is not encroached, we discard it and move to the next element in the list, and so on. Note that the delay in testing whether a subsegment is actually encroached improves efficiency, since a subsegment can disappear before it is tested for encroachment (by a SPLIT_1 operation).

Consider the case in which we need to pick any encroached subfacet in the mesh (at STEP 2). This case is handled similarly to the previous case, i.e., we keep a list of "possibly encroached subfacets." Maintaining and querying this list is essentially similar.

Now, we shift our focus to the case in which we need to find a subsegment encroached by a given point c, as in lines `06` and `12`. In this case, we compute the Bowyer-Watson insertion-polyhedron of c (but note that c is not inserted). Again, the edges of the tetrahedra in the insertion-polyhedron may correspond to subsegments encroached by c, so any

attached subsegment is candidate. No other subsegments need to be considered; one can see this from the fact that at lines `06` and `12`, all subsegments are present in the 3d DT (and thus attached) since STEP 1 is complete, and a subsegment encroached by c must be attached to a tetrahedron having c in its circumsphere. Thus, in any case, we may safely neglect the list of "possibly encroached subsegments".

The case in which we need to find a subfacet encroached by a given point c, as in line `14`, is handled similarly to the previous case.

VII. Bookkeeping of Zones

For each tetrahedron, we keep the "zone" in which it resides; the zone of a tetrahedron is either *inside*, when the tetrahedron is in the domain to be meshed, or *outside*, when the tetrahedron is outside the domain. We need zone-information at line `10` of the algorithm, because we only need to split skinny tetrahedra which are actually inside the domain (doing otherwise would be wasteful).

Note that, because we need the zone-information at STEP 3, we are certain that all subsegments and subfacets are part of the mesh and thus the zone of tetrahedron is in fact properly defined (no tetrahedron can "pierce" through a facet so that it is in two different zones simultaneously). Therefore, when we reach STEP 3 for the first time, we mark each tetrahedron with its proper zone.

Of course, when points are subsequently inserted into the mesh, the well-definedness property of the zone-information is potentially lost. However, tetrahedra which are untouched by a point-insertion can clearly keep their zone-information. Also, when inserting a point into the 3d DT, and all tetrahedra in the Bowyer-Watson insertion-polyhedron are in the same zone, we can keep the zone information. However, when these tetrahedra are in a different zone, we mark the tetrahedra created by the insertion to have an "unknown" zone.

When querying for the zone of a tetrahedron t at STEP 3, we thus have to consider the case of ending up with an "unknown" zone. Again, we know that the zone we are querying is well-defined (since we query it at STEP 3). Also, other tetrahedra around t may carry correct zone-information. The approach is to walk from t towards a point at infinity. We will then eventually run into a tetrahedron u which has known zone-information, or we run into the boundary of the 3d DT. In either case, we can reconstruct the zone of t by monitoring the number of subfacets we crossed during the walk. Once we have found the zone of t, we recursively reconstruct the zone information of other tetrahedra around t (until we run into a boundary or a tetrahedron with known zone-information). This last step is necessary to ensure the efficiency of future queries. Note that, when following this approach, the number of tetrahedra visited during zone-reconstruction is actually at most the number of tetrahedra visited during point-insertion, and hence the efficiency of the method is not fundamentally altered by these zone-bookkeeping procedures.

VIII. Point-location

An operation which was not discussed yet, but is essential in the algorithm is that of point-location, i.e., finding the tetrahedron t in which a given point p lies (we consider here the 3d case).

Point-location can be performed by starting with some random tetrahedron u and "walking" through the DT towards the point p (this is called a linear walk [10]). In essence we may consider each face f of u, and in case the apex of the face (the node of u not in f) is on the opposite side of f w.r.t. p, we replace u by the tetrahedron abutting it at f. Eventually we will find a tetrahedron containing p (this can be seen from the acyclicity theorem in [3]).

Of course, to maintain efficiency, it is essential that the initial tetrahedron u is chosen to be (topologically) as close as possible to the tetrahedron t of interest, i.e., the number of steps issued by a linear walk should be kept minimal. One might consider to take u in each search to be equal to the tetrahedron t in the previous search. However, the insertions into the mesh (and therefore point-locations) happen in a quite non-local manner, as for example, missing subfacets may be repaired simultaneously at two completely opposite areas of the domain.

From the above, we see that it is important, that for any operation on the mesh, we keep track of an appropriate tetrahedron which is topologically close to the area of interest. Consider for example the case of inserting (into the 3d DT) the circumcenter c of an encroached subfacet f. We can find a tetrahedron close to f by considering the tetrahedra attached to f, but f might not always be attached to a tetrahedron. Therefore, we explicitly store with each subfacet a tetrahedron which is topologically "close" to it.

When pointing to a tetrahedron t which is "close" to some point or area of interest, we have to take into account that the tetrahedron may actually be removed when inserting new points into the mesh. Therefore, we try to re-use tetrahedron-objects as much as possible when emptying and re-populating the insertion-polyhedron during point-insertion. Following this approach, tetrahedron-objects remain close to their original geometric location, and this is experimentally verified.

Note that when an insertion leads to a decrease in tetrahedra (which is not the common case), we cannot reuse some of the tetrahedron-objects. In such situations we let those remaining objects point to objects which do end up in the mesh.

Figure 2 shows the point-location walk-lengths throughout the execution of the algorithm, when meshing the sample PLC of Figure 5. We observe that initially, walk-lengths are large, but this is simply because initially the "topologically close" tetrahedra are not known yet (it turns out that this has only a marginal effect on runtime). Figure 3 shows the same graph for a mesh which is about 10 times larger. We see about the same trend, although the outliers in the second graph are larger (which we can expect, since the topological distances between elements in the mesh are supposedly also larger). We note that for *both* meshes, point-location immediately finds

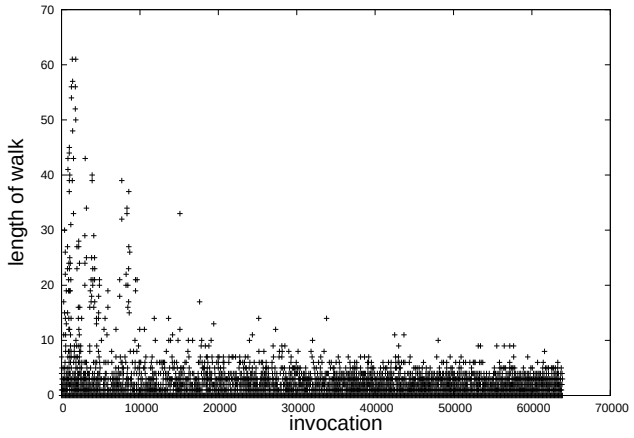

Fig. 2. Walk-lengths for three-dimensional point-location operations when refining a sample mesh.

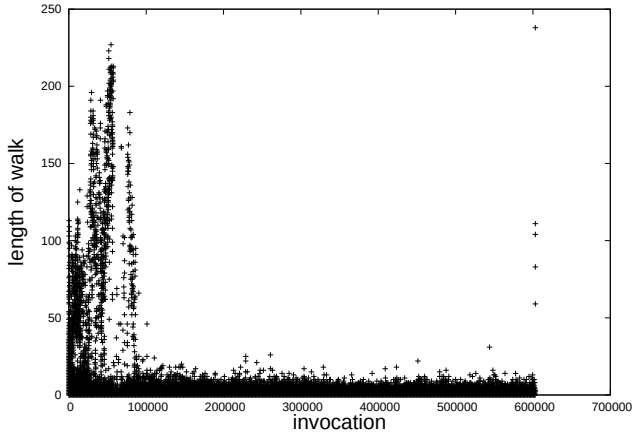

Fig. 3. Walk-lengths for three-dimensional point-location operations when refining a mesh of approx. 10 times the complexity of Figure 2.

the desired tetrahedron (i.e., 0 steps are required) in about 82% of the invocations, and less than 5 steps are required in about 98% of the invocations. Since tests with other (larger) meshes show the same trend, we conclude that point-location using our strategy for the initial guess can be considered a constant-time operation in practice.

Point-location is also necessary in two dimensions, i.e., one sometimes needs to find the subfacet s which contains a given point p (for example when p is to be inserted into the corresponding 2d DT). The 2d operation is almost completely similar to the three-dimensional variant described above. However recall from Section IV that we remove subfacets which lie outside the facet. This means that a linear walk may prematurely end at a subfacet q at the boundary of some concave part of the domain. Fortunately, we may assume that our initial guess is already topologically close to the subfacet of interest, and thus we can, without significant loss of efficiency, complete the query by performing a breadth-first search through the 2d DT, starting at q.

IX. Representation of Facets

As mentioned above, besides the 3d DT for the overall mesh, we keep a separate 2d DT for each of the facets. Note however, that it would be cumbersome to manipulate these DT's using two-dimensional primitives, since they are embedded in a 3d space. A proper implementation then would probably require explicit rotation and projection operations, which are of course undesired since they are prone to round-off errors.

Therefore, we manipulate each 2d DT by using three-dimensional predicates. In essence, for each facet f, we create an external node, which is located at a distance from the plane of the facet. This node is called the *apex* of the facet, and is denoted here by a_f. The distance from the plane of the facet is chosen such that the apex is unlikely to encroach upon any of the subfacets, for reasonably shaped subfacets.

Now, let us formulate the equivalent of a 2d-orientation test on the three nodes of a subfacet s and some point p (thus a decision on whether p is "inside" s or not). We denote the three nodes of s by n_1, n_2 and n_3 and assume they are ordered counterclockwise when viewed from a_f. Assume also that the tetrahedron (a_f, n_1, n_2, n_3) is "properly" oriented. Then we define that "p is inside s" when the tetrahedra (a_f, p, n_2, n_3), (a_f, n_1, p, n_3) and (a_f, n_1, n_2, p) are properly oriented (thus we use the ORIENT3D predicate three times in this case).

For the (2d) incircle-test, we use the INSPHERE predicate, by taking the a_f node into account. Thus given the three nodes of a subfacet s, named n_1, n_2, and n_3, and a test-point p, we test whether the point p lies inside the circumsphere of the four points a_f, n_1, n_2, and n_3.

With some basic assumptions, it is not difficult to see that both tests should work in practice. However, the nodes which are part of a facet may not lie exactly in a common plane, due to roundoff errors. We have not (yet) looked into this issue, and certainly, in order to find tests which work as expected under any circumstances, more work is necessary.

X. Front-End for Meshing VLSI Structures

Although our mesh-generator is suitable for an extended range of applications, we specifically designed it for the purpose of analyzing parasitics in VLSI interconnect structures. To be able to read descriptions of physical VLSI layout and mesh them, we added a front-end to the mesh-generator, which is described in this section.

The front-end consists of two stages. The task of the first stage is to read a GDSII file, which decribes the lateral geometry of the VLSI interconnect structures [11]. Using external technology data, the polygons in the GDSII file are extended into the third dimension, and a so-called "overlapping PLC" is created. An overlapping PLC is similar to a PLC, except that nodes, segments and facets may (partially) overlap (however, the interiors of any two facets may not overlap). The task of the second stage is to convert the overlapping PLC into a regular PLC, by removing overlapping features. The second stage also removes features which unnecessarily reduce the local feature

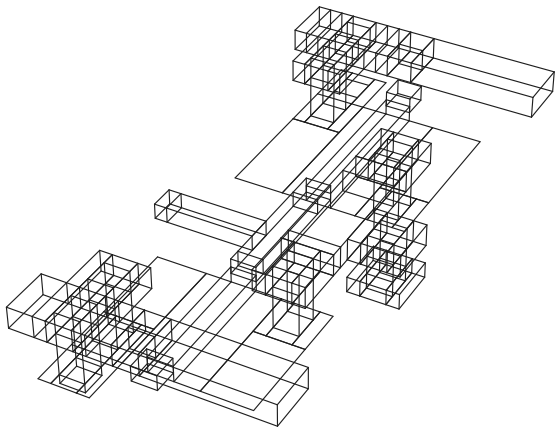

Fig. 4. Original input describing the layout of a cmos-inverter. In the input, faces, segments and nodes are (partially) overlapping.

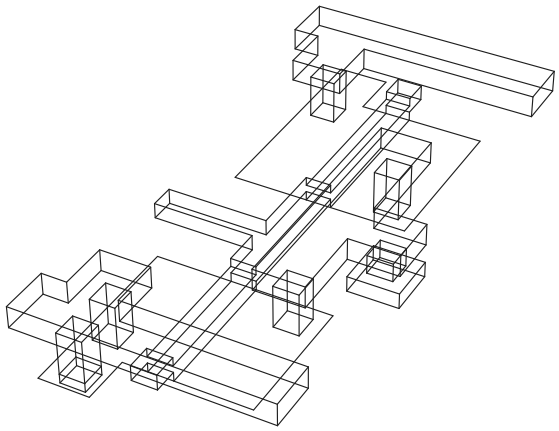

Fig. 5. The original input of Figure 4 converted to a proper piecewise linear complex.

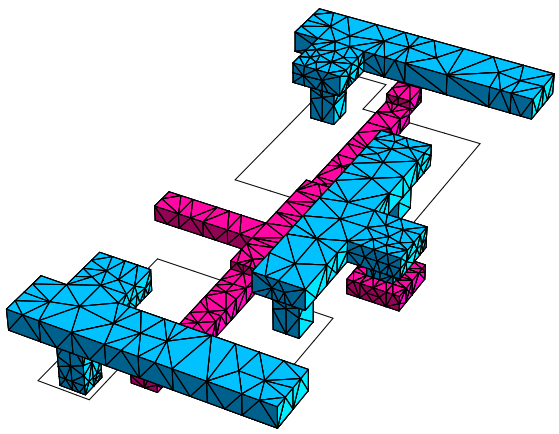

Fig. 6. Mesh for the piecewise linear complex of Figure 5. Note that the whole domain is contained in a rectangular box, which is fully meshed (some tetrahedra have been removed to reveal the contained structure).

size ([3]) in certain areas of the PLC, which is important in order to keep the resulting mesh as small as possible.

We implemented the first stage of the front-end by extending the SPACE layout-to-circuit-extractor [12]. SPACE is able to read GDSII data, and has its own format for technology data. The GDSII data basically describes a set of masks, where each mask is represented by a set of two-dimensional polygons. Internally, SPACE uses a scanline-algorithm to break the layout data into trapezoids, where each trapezoid describes a (two dimensional) region over which the layout consists of a fixed set of masks [13]. For every combination of masks, the technology description is used to generate a corresponding set of three-dimensional prisms, related to chunks of conductors. Every prism is basically the z-extended form of a trapezoid, and it is output as a set of six facets in the overlapping PLC (when one of the sides of the trapezoid has zero length, the trapezoid reduces to a triangle, and naturally, the corresponding prism will be output as five facets instead of six).

The above approach has the advantage that it is quite flexible: for every combination of masks, we can determine which conductors should exist, and at what z-coordinates they should lie. On the other hand, many overlapping facets will be generated, as well as many unnecessary nodes and segments. This deficiency is overcome by the next stage.

The second stage of the front-end, i.e., that which converts an overlapping PLC into a regular PLC, is logically divided into a number of steps, which we will briefly describe now.

- (STEP 1) For any two segments which intersect in their interior, a node is created at their point of intersection (if the interiors overlap at more than a single point, no action is taken).
- (STEP 2) Any node which lies at the interior of a segment will split that segment into two pieces.
- (STEP 3) Any duplicate segments are deleted.
- (STEP 4) If a chain of segments divides a facet into two or more separate regions, then the facet is split along this chain, and two or more facets result. This is done for all facets, and it is repeated until no more facets can be split.
- (STEP 5) For any pair of facets, if the two facets have an equal boundary, then either both facets are removed, or only one facet is removed (this depends on the type of material of which the facets form the boundary).
- (STEP 6) Two facets which lie in a common plane, and which share one or more boundary segments, are merged into a single facet (the boundary segments will become interior segments of the new facet). This is repeated for all pairs of facets, until no more facets can be merged.
- (STEP 7) Any segment which lies in the interior of one facet, and is not part of any other facet, is deleted.
- (STEP 8) Any node which joins exactly two segments which lie on a common line is removed, and the segments are joined into a single segment.
- (STEP 9) Any node which is no endpoint of a segment is deleted.

Note that, due to the approximative nature of floating-point numbers, most of the geometric tests are performed while

keeping a certain margin. For example, in STEP 2, we consider a node to lie in the interior of a segment if it lies sufficiently "close" to the segment, but not "close" to one of its endpoints. Thus, the second stage is not robust against all kinds of overlapping PLC's, but it will be able to handle them in most practical cases. If the overlapping PLC's are generated from VLSI data (as in our case), robustness is no practical issue, since VLSI data generally has to obey design rules which are much stricter than the rules we need to impose for robustness.

Figure 4 shows an overlapping PLC which was created from the layout of a cmos inverter cell. Figure 5 shows the PLC which resulted after the second stage of the front-end. Of course, it is not possible to see which elements overlap in this figure, but notice that many unnecessary features have been removed. Figure 6 shows the final mesh of the layout.

XI. CONCLUSIONS AND FUTURE WORK

We have given a detailed description of the design and implementation of a Delaunay-based mesh generator.

Some issues remain open. One issue was given in Section IX: the inherent roundoff in floating-point numbers results in the nodes of a facet to not lie exactly in a common plane. It is not clear how to adapt the geometric predicates such that correct operation of the algorithm can be theoretically guaranteed. Related problems exist. For example, when determining whether a point p encroaches upon a subsegment, we need to determine the distance of p to the center of the subsegment. However, such distance can be computed only to some accuracy. It is unclear whether we need to resort to exact (or adaptive?) computations, or whether some margin is warranted. As a final example, when computing the circumcenter of a tetrahedron, some rounding error is effectively made in general. It is of course important to know whether this rounding error can be safely ignored (in fact, we deliberately add some noise to the coordinates of nodes to reduce the frequency of degeneracies, as was noted in Section III, and it would be good to have a theoretical bound on the amount of noise that is permitted).

Other issues are the restriction of the PLC to minimum angles of $\pi/2$, and the existence of 'slivers' in the resulting mesh. However, these issues are not important in all applications.

Despite the deficiencies mentioned above, our implementation works well in practice, and is capable of generating about 15.000 tetrahedra per second on a modern Intel-based workstation. A copy of the implementation may be obtained by contacting the authors.

REFERENCES

[1] W. Kao, C.-Y. Lo, M. Basel, and R. Singh, "Parasitic extraction: current state of the art and future trends," *Proceedings of the IEEE*, vol. 89, no. 5, pp. 729–739, May 2001. [Online]. Available: http://ieeexplore.ieee.org/iel5/5/20097/00929651.pdf

[2] Bern and Eppstein, "Mesh generation and optimal triangulation," in *Computing in Euclidean Geometry, Edited by Ding-Zhu Du and Frank Hwang, World Scientific, Lecture Notes Series on Computing – Vol. 1*, 1992.

[3] H. Edelsbrunner, *Geometry and Topology for Mesh Generation.* Cambridge University Press, 2001.

[4] J. R. Shewchuk, "Delaunay refinement mesh generation," Ph.D. dissertation, School of Computer Science, Carnegie Mellon University, 1997.

[5] H. Edelsbrunner and E. P. Mücke, "Simulation of simplicity, a technique to cope with degenerate cases in geometric computations," *ACM Trans. Graphics*, vol. 9, pp. 66–104, 1990.

[6] J. R. Shewchuk, "Adaptive Precision Floating-Point Arithmetic and Fast Robust Geometric Predicates," *Discrete & Computational Geometry*, vol. 18, no. 3, pp. 305–363, Oct. 1997.

[7] B. Delaunay, "Sur la sphère vide," *Bull. Acad. Sci. USSR(VII)*, pp. 793–800, 1934, classe Sci. Mat. Nat.

[8] A. Bowyer, "Computing Dirichlet tessellations," *Computer J.*, vol. 24, pp. 162–166, 1981.

[9] D. F. Watson, "Computing the n-dimensional Delaunay tessellation with application to Voronoi polytopes," *Computer J.*, vol. 24, pp. 167–171, 1981.

[10] E. P. Mücke, I. Saias, and B. Zhu, "Fast randomized point location without preprocessing in two- and three-dimensional delaunay triangulations," in *Proceedings of the 11th Annual Symposium on Computational Geometry*, 1996, pp. 274–283.

[11] Cadence Design Systems, Inc./Calma, *GDSII Stream Format Manual*, Feb. 1987.

[12] A. J. van Genderen and N. P. van der Meijs, "VLSI modeling and verification," 1994–2006, home page of the modeling and verification project, available at URL `http://space.tudelft.nl`.

[13] N. P. van der Meijs, "Accurate and efficient layout extraction," Ph.D. dissertation, Delft University of Technology, Delft, The Netherlands, Jan. 1992.