Heterogeneous Process Simulation Tool Integration

Zakir H. Sahul, Kenneth C. Wang, Ze-Kai Hsiau,

Eugene W. McKenna, and Robert W. Dutton

AEL 231C, Integrated Circuits Laboratory

Stanford University

Stanford, CA 94305-4055

Phone: (415)723-1482

email: sahul@gloworm.stanford.edu

Abstract

An unified strategy for achieving heterogeneous tool integration within the technology computer-aided design (TCAD) realm is presented. Geometry, grid and surface mesh servers are defined and implemented in a program that contains the different data and provides common services for TCAD applications. These services form the backbone of the integration framework. Functional abstraction is used to provide unified access to the servers' data and procedures. The role of each server and their interactions, including those with applications, are delineated. Existing tools such as SUPREM-IV and SPEEDIE, together with new tools have been integrated to illustrate the utility and versatility of the approach. This paper discusses the integration strategy, shows results and fully specifies standard functional server interfaces needed for TCAD tool integration.

1 Introduction

Full process simulation involves the numerical solution of equations describing the physics of diffusion, oxidation, lithography, ion implantation, etching and deposition steps, resulting in geometry and doping profiles that define a device. Traditionally, different tools have focused on the solution of each of the different process steps. SUPREM-IV [9] and FLOOPS [10] solve diffusion and oxidation equations; CREEP [19] solves only the oxidation equations; SPEEDIE [11] and SAMPLE [12] solve boundary motion equations for etching and deposition. For greatest flexibility, it is desirable to use the best available tool for simulating a given process step, thus creating the need for tool integration.

Three different levels for achieving this integration are considered: 1) file formats, 2) information models and 3) server models. If the output data file from any tool can be used as the input data file for any other tool, all tools can be seamlessly used together. A standard file format provides this capability and obviates the need for format conversion programs. If all tools use a standard internal data structure or information model, each tool and its subroutines can be used as subfunctions within a larger simulation environment. The tools can be developed independently and used interchangeably. However, while an information model is useful for new tool development, existing tools can be integrated only through a file in lieu of a total program rewrite. Conversely, a file format alone, while ideal for integrating existing tools does not provide the benefits of code-reuse (assuming no supporting library) or standard information models for writing new tools.

Moreover, data file formats and information models are not sufficient for tool integration. The problem arises from incompatible data representations and manipulation techniques of the different tools. Consider the integration of a grid based diffusion tool and a surface mesh based etching tool. The diffusion tool output consists of a grid with dopant concentration solutions. A surface mesh can be easily constructed from the grid[1]. This surface mesh can be transformed by an etching solver to create a new device geometry. For further diffusion simulation, the new geometry must be gridded, and the dopant solution values interpolated from the old grid - services that etching solvers typically do not provide. Clearly, a "server" to reconcile the new geometry with the old grid is needed.

In this paper, a new heterogeneous TCAD tool integration strategy focusing on server functionality is presented. Data used by the various TCAD programs is identified. The *Forest* system implements geometry, grid and surface mesh servers that are carefully limited in scope and defined with functional abstraction to contain and operate on the data. Both existing and new tools benefit from the clearly delineated roles and interactions of the three servers. Results from integrating existing tools, such as SUPREM-IV and SPEEDIE, as well as other new tools are used to illustrate the strategy. Finally, a minimal standard server API to aid tool integration is specified.

Note that the term "server" refers to a combined function library and data repository that provides data and services for client TCAD applications. While similar to the traditional client-server concept, in the current implementation, the server is not a separate process; instead it is linked with the client applications during compilation.

After reviewing previous work in Section 2, the integration architecture and servers are presented in Section 3. 2D implementation results are presented in Section 4 and 3D considerations are presented in Section 5. Summary and concluding remarks are presented in Section 6. Server functions that define a new tool integration standard are presented in Appendix A.

[1] The term "grid" is used exclusively to refer to boundary *and* interior discretizations while the term "mesh" is used exclusively for boundary discretizations. Grids span areas in 2D and volumes in 3D. Meshes span curves in 2D and surfaces in 3D.

2 Previous Work

Earliest work in tool integration involved data conversion programs interfacing process and device simulators. The multi-organization European DAMSEL [3] effort introduced common FORTRAN-77 data structures defining geometries, grids, solution values and matrices for integrated process and device simulation. While several integrated process and device simulation tools were written with the data structure, integration of other independently developed tools was not considered.

The pioneering work of Duvall [4] at Intel defined a “Profile Interchange Format” (PIF) enabling broad inter-operability among TCAD applications. PIF specified a self documenting dimensionality independent data file format for storing all simulation data. The highly successful VISTA system [5] from TU-Vienna overcame minor ambiguities in PIF by further standardization and provided a toolkit library. Independent applications were easily interfaced into the system. Modified PIF formats were also used in the Philips design environments IDDE [7] and PRIDE [18].

Working with PIF objects (coordinates, lines, regions, elements, grids, etc), the in-memory information models PIF/Gestalt [1] at MIT and BPIF [22] at Berkeley were developed. PIF/Gestalt differed from BPIF only in the degree of definition formality and conceptual data hiding. Object manipulation operations were mirrored in the PIF database. As an information model, however, the use of PIF/Gestalt for integrating existing tools was limited and tools like SUPREM-IV were integrated using a file interface. Both these techniques were also very closely linked to and constrained by the original PIF model.

Berkeley's SIMPL-IPX system [21] provided utilities to integrate 2D topography and impurity profile simulators. Geometries were updated from SAMPLE etch profiles and SUPREM-IV grids were updated from geometry profiles.

Recent work towards tool integration has been based on an object-oriented client-server architecture called the Semiconductor Wafer Representation (SWR) [6] [23]. The multi-organization American SWR effort defined an information model consisting of a hierarchy of C++ classes and utilities for geometry and field (grid) servers. However, the complete specification of the class hierarchy and utility methods discouraged alternative implementations. The inconsistent separation between geometry and field servers also led to complex server-server and server-application interactions.

The technique presented in this paper builds upon the experiences of these previous approaches. Applications are presented with a data repository-like interface to servers that also provide a functionally abstracted and simplified information model. Server utilities provide all the necessary functions for integrating new and existing tools.

3 Integration Architecture

As illustrated in Figure 1, three different types of tools are typically used for TCAD simulation: 1) Grid based tools (eg. diffusion solvers) solve for quantities inside the device body. 2) Surface mesh based tools (eg. etching and deposition solvers) solve for quantities on device boundaries. 3) Geometry based tools "simulate" topography steps by performing boolean geometric operations. Other hybrid simulators (eg. oxidation programs) solve for quantities in the interior and on the device boundaries, typically using gridded structures. Since the different types of tools each use different device and attribute representations, any integration environment must support all these data representations and adjudicate incompatibilities.

The integration architecture of the *Forest* system is shown in Figure 2. It consists of three core modules - a geometry modeler, a surface mesh generator, and a grid generator. The core modules together with access functions comprise a "server[2]." The access and utility functions specify minimum server capability. The data within these modules is accessed and manipulated only via an application procedural interface (API). The interface is kept as simple as possible without any hierarchy to allow easy translation, #define or inline function access to the internal data of the core modules. Applications use the API to obtain data from the servers, perform computations on their local representations and update the servers as needed. The process can be viewed as simple input/output (I/O) between an application and the server even though the server performs many functions that are hidden from the application as discussed below. The advantage of the data repository I/O view of the server is that existing applications can be easily integrated without extensive source-code changes.

Figure 2 shows schematically a restructured version of SUPREM-IV that supports the integration architecture. The code example shown in Figure 3 represents information interactions between *any* pair of oxidation and diffusion modules. Considering only the diffusion step in Figure 3, the current solution values and grid are obtained from the server and the diffusion module's data structures are constructed (line 3 in the figure) at each time-step. The diffusion module operates on this data and computes a new solution for each time-step (line 4). Upon completion of the time-step, the new solutions are used to update the server's data areas and the diffusion module's data areas are deleted (line 5). This procedure is continued throughout the solution process. The scheme adds a minor data conversion overhead at each time step, while providing the benefit of allowing specialized grid techniques to be applied and changed seamlessly.

Note that an error estimate is used for grid adaption (lines 8 and 9). The error estimator's data representation is different from that of the diffusion module. Hence, the error estimator obtains grid and fields data directly from the server (line 10 in Figure 4) and sends updates to the server (line 12).

[2] The collection of the three servers is also called "server." The context is used to determine the appropriate definition.

Integration of a stand alone oxidation program[3] also called from the diffusion solver is also illustrated. The oxidation solver is called from within the diffusion loop (line 1) only to compute *velocities* on the grid points. Grid motion is performed by the server. Moreover, since the oxide solver uses a different grid than that used by diffusion, it calls the grid server separately (lines 13 and 14 in Figure 4) and updates velocity solutions (line 16) independently from the diffusion solver. Upon completion of each diffusion step, dopant data is updated, error estimations are computed and grid adaption is performed. However, the diffusion solver must be able to adjust the grid points to properly account for dopant flux across moving interfaces. Since the responsibility of boundary motion is allocated to the server, a potential conflict arises. This difficulty is resolved by requiring the diffusion solver to update the new dopant concentration solutions, including grid motion, before the server actually moves the grid points. When the server moves the grid points (line 6), concentrations are then assigned to the correct spatial points. The actual boundary motion performed by the grid server also invokes the geometry server as described in Section 3.4. Yet, these details are hidden from the diffusion and oxidation applications.

This overall strategy allows large, complicated applications and servers to be developed separately and linked with any set of other applications and servers. Data structure or programming language constraints are not imposed on either applications or servers. The oxidation and error estimation modules, for example, are in fact separate applications. In this manner, a number of different diffusion, oxidation and error estimator tools can be used collectively and interchangeably without requiring custom interfacing. Only the translation routines that create each tool's data structures and update server data need to be created and they are the responsibility of application programs. Applications can access data from any or all of the three servers as appropriate.

The functionality of three - geometry, surface mesh, and grid - servers in *Forest* and their use in an integrated TCAD simulation environment are described in the following subsections. Any server implementation that provides the same capability using the API described in Appendix A can be used in the integration framework. The specific server implementations and associated algorithms of this work are discussed in Section 4.

3.1 Geometry Server

The geometry server stores and manipulates the gross structural features of a device by tracking material boundaries. Geometric capabilities support both physical simulation and rapid structural prototyping. The geometry server has the following functionality:

[3] one that is independent of the diffusion solver.

- Read/store a geometry from persistent storage,
- Update geometry with a new geometry,
- Determine if a geometry is valid,
- Move the surfaces of the geometry,
- Repair the geometry if it becomes invalid,
- Determine the maximum surface displacement,

Geometry applications typically compute a change in the geometry and request the server to update the current global geometry. Each application is given the freedom to destroy a geometry and to construct a new one. In-place geometry modification is provided to support simulation steps such as oxidation and etching/deposition even though applications might corrupt the geometrical connectivity. While detecting anomalies and repairing them is extremely difficult in the most general case, even in 2D, a few operations are useful for full process simulation. Thus, the geometry server provides three types of geometry checking and and repair functions. First, since material growth (topography and oxidation) simulations generally create loops particularly at concave corners, loop detection and removal are provided. Second, as some applications use only a subset of the geometrical structure to perform computations, an advancing front might crash into another region. Crash detection and boundary collapsing operations are provided by the server. Third, since the geometry is sometimes built from a dense grid or surface mesh, the geometry data is often complicated. Smoothing operations are used to reduce the complexity.

For many processing steps, most notably lithography and topography steps, a geometry operation is used instead of a physically based simulation [20]. The types of operations which fall into this category include boolean set operations (e.g. intersection), geometric constructions (e.g. sweeps), translations, and rotations. The operations required for such manipulations have been relegated to applications.

3.2 Grid Server

Many TCAD applications require gridded structures - a collection of elements whose union forms the device cross section and solution values defined over grid points or elements. In addition, a set of utilities is required for grid manipulation as the structure changes physically due to processing. The following functions are provided by the grid server to support these requirements:

- Generate area/volume grid on the geometry,
- Define solutions over the grid points and elements,

- Adapt grid for solution efficiency and accuracy,
- Move grid to simulate material transport and structural change,
- Determine if grid connectivity is valid,
- Repair corrupted grid,
- Determine maximum nodal displacements to preserve grid validity.

The grid server within this framework includes a grid generation program that can generate grids for arbitrary multi-layer and multi-material geometries. Higher order elements are supported to allow grid points anywhere within an element. The server also maintains solution values defined over the grid points and over the elements. Applications add and delete solution variables and values. The grid server interpolates solutions from one grid to another as the grid changes. The type of interpolation used for different solutions is specified by the application (eg. linear, log, dose conserving, etc.)

Grid adaptation is performed by the grid server, thus decoupling adaptation from the solver. The criteria for adaptation is computed by an error estimate application. The grid server refines or coarsens the elements and interpolates solutions between the original grid and the adapted grid. Upwinding solvers that compensate for grid adaptation induced artificial diffusion [15] need to instruct the grid server to hold the displacement and previous node positions as extra solutions to be used at the next timestep.

The grid server moves the mesh to simulate material transport; applications only compute nodal velocities or displacements. Since grid motion may result in an invalid grid, the grid server determines the maximum displacement or time-step that can be used before the grid becomes invalid. Even if the grid is corrupted or its quality diminishes, the grid server makes the necessary repairs. The grid server also interacts with the geometry server continuously to avoid geometrical anomalies arising from grid motion.

3.3 Surface Mesh Server

Programs like SPEEDIE solve for quantities on device boundaries using a surface mesh. For moving boundary problems, the surface is advanced based on a velocity solution at the mesh points. Since there is no fundamental difference between grids and meshes, the functions defined for the surface mesh server are analogous and equivalent to those defined for the grid server. Differences resulting from application specific use are illustrated in the following subsection and sections.

3.4 Interaction between Geometry and Grid/Mesh Servers

For moving boundary problems such as oxidation, etching and deposition, the geometry, surface mesh and grid should be kept consistent between processing steps. The required interactions between the respective servers are discussed in this section. Note that the responsibilities of the different servers is unambiguously divided.

After a geometry application has finished its operations, the grid and surface mesh need to be computed. As all grid (or mesh) based applications call the grid (or mesh) generation function before they access the grid (or mesh), re-synchronization between the geometry and grid (or mesh) is performed by the grid (or mesh) generation functions.

During surface mesh based etching and deposition simulation, the surface mesh server moves the surface mesh and calls the geometry server to update and repair the geometry. If the geometry changes due to repair, the surface mesh is regenerated. While the surface mesh and geometry are continuously updated, the grid is updated only upon completion of the unit process simulation step and only when a grid based application calls the grid server to generate a grid.

For oxidation simulation, the grid server moves the grid points based on the computed displacement field. The grid server calls the geometry server to update the geometry; in turn, the geometry server performs geometry validity tests and repairs the geometry as needed. The grid is then regenerated with this new geometry and solutions are interpolated. In this case, the geometry and grid servers are continuously re-synchronized. The surface mesh is updated only upon completion of the entire oxidation simulation and only when a surface mesh based application calls the surface mesh server to generate a mesh.

Some etching and deposition tools use a cell based approach [13] - adding or deleting cells of materials and thereby simulating material growth. Since cells are elements in a grid, the grid server is used to generate the cells. In this case, however, an extra "gas" region above the device cross section needs to be defined and gridded by the grid server. Cell deletion and addition is performed by changing the material attribute of interface elements. Upon completion of the growth simulation, the new geometry is extracted and updated. The surface mesh is updated upon a subsequent call to the surface mesh generator.

Implicit in their interaction is the ability of the three servers to update their respective view of the device based on the view of another server. The geometry server has to update the geometry based on a (changed) grid or mesh. The grid and mesh servers need to update the grid or surface mesh based on a new geometry. Since the grid and mesh servers encapsulate a grid or mesh generator, grid and mesh updates are performed when an application specifies a (re-)generation. On the other hand, the geometry server is updated when the grid or mesh server *computes* a new geometry based on its view and sends the new geometry to the geometry server. The geometry server incorporates this new geometry with its current view.

Note that consistency does not imply congruency between the different data representations. For example, the grid only approximates the geometry of the device. Similarly, surface mesh nodes and volume grid nodes do not necessarily coincide. Applications that require congruency should use one representation to build the other representations using the provided utilities. The grid representation can be used to build congruent surface mesh and geometry representations while the surface mesh representation can be used to build a congruent geometry representation.

4 Integration Results

The server based approach is used to integrate *existing* tools such as SUPREM-IV and SPEEDIE together with several new applications. *Forest* [16] is used as a geometry modeler, gridder, and surface mesher. SUPREM-IV provides ion implantation, diffusion, and oxidation simulation capabilities and SPEEDIE models topography changes due to deposition and etching. New error estimation tools support grid adaptation and new geometry applications provide boolean operations. The *Forest* server is first described and then its functionality is illustrated with example interactions with the various tools.

Forest is implemented in C++ with approximately 15,000 lines of code (including comments) for the geometry server and approximately 20,000 lines of code for the combined grid and mesh servers. Support classes and methods comprise another 4,000 lines of code.

The *Forest* geometry is a boundary representation model that uses a hierarchy of points, edges, boundaries, surfaces and volumes to represent geometries (Figure 5). Pointers are used to represent the hierarchy. Edges hold pointers to two points; boundaries hold pointers to several edges; surfaces hold pointers to boundaries, etc. Pointers to edges in a boundary are ordered such that consecutive edges form a closed loop. None of the objects within the server is duplicated. When a new object is added, the server searches through the existing object lists to find a matching object. A new object is added only if a matching object does not exist. Since all algorithms within the geometry server start from the highest dimensional object, parent query operations for the lower dimensional objects is not provided. For example, adjacency information between two boundaries is determined by checking to see if the boundaries hold pointers to the same edge.

Surface mesh generation is performed by decimating geometry edges. Grid generation within the server uses a tree based approach: the geometry is enclosed in a root box that is recursively decimated into smaller boxes. End boxes are tessellated using templates [16]. The grid server information model is shown in Figure 6. The mesh server information model is similar. The grid consists of elements and grid points with solutions and solution values defined over them. Similar to the geometry server, objects are not duplicated.

A hash table is used to pre-sort grid points according to their coordinates to reduce search times. Unlike the geometry server, however, the lower dimensional grid point objects hold pointers to their parent elements. While this relationship is not made public through the API, it is used internally to build element adjacency information.

Interaction between the geometry and mesh servers is illustrated by the new boolean applications which provide geometric etching and deposition. The applications compute a new geometry and update the geometry server directly. Figure 7 illustrates the functionality of the deposition application and the interaction of geometry and grid servers. In Figures 7(a) and 7(b), the data of the geometry and grid server before the boolean geometry operation is shown. The boolean operation is performed as illustrated in Figure 7c. The grid is subsequently regenerated as shown in Figure 7d.

The functions of the surface mesher and geometry modeler are illustrated in Figure 8 for a 0.25 micron thick oxide spacer deposition simulation. SPEEDIE uses the *Forest* server for all surface mesh and geometry operations. It obtains a surface mesh from the server, computes velocity vectors for the mesh points and then instructs the server to move the mesh. The server, after moving the mesh, computes a new geometry from the surface mesh. The geometry server checks for loop formation by testing for intersections between non-consecutive edges in a boundary. All edges between the intersecting edges form the loop and are excised by the geometry server during delooping. The surface mesh is then regenerated by the mesh server for further simulation. CPU usage for the simulation is listed in Figure 9. Time spent in the server performing surface movement, loop detection and removal takes only 6% of the total CPU time.

Grid server capabilities are illustrated in Figure 10 which shows sequential grid frames during grid adaption for an implant simulation. Each individual module within the SUPREM-IV program is restructured to use the *Forest* grid server for data and services as discussed in Section 3. The SUPREM-IV ion implantation module uses the grid adaption features of the server to accurately resolve the implant profile. The error estimation application uses solution variation across an element to determine if the element should be refined, coarsened or left as is. The tri-valued flag stored as a solution is used by the server for grid adaptation. To minimize interpolation error, the implant application recomputes the profile and updates the server after each adaption step. Grid adaptation is performed by pruning or growth of the tree data structure in the grid generator.

The diffusion application is, from the server's perspective, the same as the ion implantation application. Figure 11 shows a junction diffusion simulation with grid adaption based on the SUPREM-IV link to *Forest*. Since the diffusion application in SUPREM-IV computes a symbolic factorization of the global stiffness matrix based on the current grid, a new grid at every time step results in expensive repeated symbolic factorizations. But the grid connectivity is changed only by regridding during oxidation or a grid adaption step; hence, no symbolic factorization is performed unless these conditions occur.

When the server is updated within an adaption or time-stepping loop, the number of updates can be large and efficiency is of the utmost importance. The penalty for using the server, defined as the total server access time, is the sum of the times taken to 1) obtain grid and solution data from the server, 2) to build the local representation (including building all derived connectivity information and calculation of coupling coefficients, etc.) and 3) to update server solution values at each time step.

Figure 12 compares the *Forest* server access time with the total CPU time for several SUPREM-IV diffusion simulations. Breakdown of the CPU time between the three operations is also shown. For the simplest diffusion simulation of a boron implant drive-in using the "fermi" model, server access time can be as high as 20% of the total CPU time and is nearly independent of the number of nodes. The biggest part of the total server access time for SUPREM-IV, however, is spent building a local congruent geometrical representation from the grid representation that is *not* used for diffusion calculations. Simulation times for more advanced diffusion models are higher and consequently, the fractional cost of server access is smaller.

Use of the coupled diffusion/oxidation modules shown in Figure 3 is illustrated in detail as shown in Figure 13. In this case, SUPREM-IV provides the functionality of both diffusion and oxidation applications; the server brokers sub-sets of the data for the separate application modules. The diffusion solver uses the entire domain (all material regions) for computing dopant concentrations while the oxidation solver uses only the oxide and nitride regions to compute grid point velocities. Data translation routines access the necessary elements in the grid and build a separate representation for the oxidation application. Such an interaction would be extremely cumbersome in a file based implementation.

The oxidation application obtains an initial grid from the grid server and computes velocity fields over the grid points. The grid server is then called to obtain the maximum allowable time step to maintain grid validity. Grids are considered valid as long as element nodes do not overtake one another (determined by negative element areas). The geometry server is then called to obtain the geometry limited time step to avoid boundary loops and region crashings. The application then calls the server to move the nodes for a given time-step which may be greater than the maximum time steps discussed above. The geometry server moves the geometry and repairs anomalies - loops and region crashes. Region crashes are detected by checking for intersections between edges of different regions. Crash removal is similar to delooping where a loop is defined as edges of one region that are in another. *Forest* also collapses nearly crashing regions. If a point along a boundary is very close (within a user defined threshold) to a second boundary, the point is snapped onto the second boundary and the geometry connectivity is rehashed. The grid server moves all the grid nodes and repairs the grid by regeneration. Since a tree based approach is used, grid regeneration adds nodes only in areas where the boundary has changed [17]. The simulation is then continued with the next time-step.

Figure 14 shows the linking of SPEEDIE and SUPREM-IV based on common use of geometry/mesh/grid services. After the geometry is initialized by the server, SPEEDIE uses the surface mesh server to compute a rate based etch profile. A grid is then generated by the grid server and used by SUPREM-IV for subsequent ion implantation and diffusion steps. These heterogeneous tools, with totally different data representations and internal data structures, have been successfully integrated using the server methodology discussed above.

5 Future Extensions to 3D

The tool integration effort presented in this paper has mainly concentrated on 2D applications and all server capabilities as described in Sections 3.1 through 3.4 and Appendix A have been implemented in 2D. In this section, 3D extensions are considered.

The data representation in all three *Forest* servers is fully 3D. The geometry representation can be used to manipulate 3D volumes as well as 2D regions. The mesh and grid information models are also dimension independent. Elements of arbitrary dimension linked to an arbitrary number of grid points are supported. However, the servers in this integration strategy are defined *not* merely in their ability to represent data in various forms or dimensions but in terms of their functionality and interactions with one another and with TCAD applications.

While server-server and server-application interactions delineated in Section 3 are directly applicable in 3D, the limiting factor in 3D is the required server functionality. Several functions need to be implemented. For the geometry server, validity and repair operations are the most critical. Despite the complexity, however, an efficient $O(N \log N)$ loop detection and removal algorithm in 3D has been reported [8]. For the grid and mesh servers, generation and adaptation is the most critical. 3D grid generation algorithms specialized for TCAD applications have also been published [2]. Thus, the integration strategy discussed in this paper can be (and is being) implemented in 3D. Moreover, the server API listed in Appendix A can be used *without* modification.

6 Summary and Conclusion

A new TCAD tool integration strategy focusing on server models has been presented. Three servers - geometry, mesh and grid - capable of representing and providing data for the various TCAD tools have been defined. Minimum server functionality has been specified for tool integration. The servers consist of core modules and support utilities. The grid server, for example, is built around a core grid generator. Functional abstraction is used to provide access to the data within a server. Different core modules (grid generators, geometry modelers, etc) can thus be easily used within the integration framework owing to the simplified procedural interface.

For grid based applications, the onus of grid generation, adaption, and movement has been relegated to the grid server. Applications are presented with function calls for grid and solution data access and update. For surface mesh based solvers, the functions of mesh generation and movement have been encapsulated within the surface mesh server. Functions that mediate between geometric, mesh and grid operations are also defined as part of the integration framework. Successful implementation of this integration strategy has been demonstrated with the integration of SUPREM-IV, SPEEDIE, a new error estimator, and boolean geometric applications each supported with the grid, mesh and geometry modules of *Forest*.

A Server Functional Interfaces

The functional interface of the three servers is documented in this section[4]. The interface is general to accommodate all the needs of all the various TCAD programs. It is also dimension independent - equally applicable to 3D as well as 2D servers. The underlying data structures and implementation of the servers is hidden by functional abstractions that use and/or return handles to objects.

While handles in the API are conceptualized as pointers, the pointers are never dereferenced allowing bindings for different computer languages. If the servers are implemented in FORTRAN for example, the "iterator" functions would return integers that represent array indices as handles to objects. An application can then query the server for properties of the object by passing the integer handle to a server query function. The query function looks up the object in the data array and returns the desired properties.

A common attribution mechanism is provided for all objects in all servers. Applications specify the name of an attribute, and the attribute itself, which is stored as a character string. Three functions are provided to "set," "get" and "remove" attributes.

- void *object***SetAttribute**(*object* *, char *attrName, char *attrVal)
- char **object***GetAttribute**(*object* *, char *attrName)
- void *object***RemoveAttribute**(*object* *, char *attrName)

The *italicized* object name is replaced by one of the server objects: **Point**, **Edge**, **Boundary**, **Region**, **Surface**, **Volume**, **Geometry**, **GridPoint**, **Element**, **Grid**, **MeshPoint**, or **SurfaceMesh**. Attributes

[4] #defines and numbering conventions are found in header files and are not documented here due to space considerations. They can be found in [24].

are persistent - an attribute set by one application is available to all applications. Similarly, an object numbering scheme is also provided to set and get integer attributes.

- void *object***SetNumber**(*object* *, int number)
- int *object***Number**(*object* *)

Unless otherwise noted, functions return TRUE on successful completion and FAIL on error. However, functions that return handles return NULL on error. Persistent storage functions are specified for all three servers.

A.1 Geometry Server

Geometry modelers typically use a NURBS [14] information model or a boundary representation model. In this work, functions are defined to access and manipulate the boundary representation geometry model. Since most geometry applications change the geometry, the geometry server has all the functions needed to construct a geometry.

A.1.1 Geometry Functions

Geometry functions are listed in Figure 15. **GeometryRead** and **GeometryStore** facilitate persistent storage of geometries. **GeometryReplace** replaces the server's geometry with a new geometry. **GeometryMove** is used to simulate material movement and serves as an in-place operation. The geometry server provides the **GeometryMaxDt** function to return a maximum time step so that an application can anticipate the geometry becoming invalid. **GeometryValid** and **GeometryRepair** are used to validate and to repair an invalid geometry. The "functions," **ForEachPointInGeometry**, **ForEachEdgeInGeometry**, **ForEachBoundaryInGeometry**, **ForEachRegionInGeometry**, **ForEachSurfaceInGeometry** and **ForEachVolumeInGeometry**, represent sequencers over all the points, edges, boundaries, regions, surfaces or volumes in the geometry. The application passes a geometry handle argument and the "function" returns a handle to the desired object. Figure 16 illustrates the use of these "functions." Moreover, addition and deletion operations are provided by functions of the form shown below. The *italicized* object name is replaced with one of: **Point**, **Edge**, **Boundary**, **Region**, **Surface**, or **Volume**.

- *object* ***GeometryAdd***object*(Geometry *, *object* *)
- void **GeometryDelete***object*(Geometry *, *object* *)

A.1.2 Component Functions

Standard functions that operate on geometry objects are documented in Figures 17 - 22. For points, functions to set and get the coordinates, velocities and displacements are provided. For edges, access (set and get) to the end points is provided. For boundaries, access functions are provided for the boundary edges. Similar functions are provided for regions, surfaces and volumes. The nomenclature used in the figures is self-explanatory.

A.2 Grid Server

Grids are defined as objects with lists of elements, grid points, solutions, and connectivity between grid points and elements. Solutions can be defined on the elements or on the grid points.

A.2.1 Grid Level Functions

Functions that operate on grid are listed in Figure 23. **GridGenerate** generates a grid of type *gridtype* over a geometry. All applications must call **GridGenerate** before attempting to access a grid. Grid adaption is performed based on solution names representing adaption criteria passed to **GridAdapt**. Grid motion is supported by **GridMove** whose arguments are velocity solution names. **GridMaxDt** determines the maximum time step that can be taken before the grid becomes invalid. The arguments specify the solution names of the velocity field and the maximum allowed time step. **GridRepair** is used to re-validate an invalid grid. **GridConstruct** is used to construct a grid. The function arguments consist of coordinate values, element connectivity, element regions, and boundary conditions. **GridAddSolution** and **GridRemoveSolution** add and remove solution sets from the grid server. All solutions are defined to be of type double precision. *Solutiontype* specifies whether the solution is defined over grid points or over elements. Interpolation types are specified by the argument *interpolationtype*. The "functions," **ForEachGridPoint**, **ForEachElement**, **ForEachPointSolution**, and **ForEachElementSolution** represent sequencers over all the points, elements, point solutions, or element solutions in the grid. They are similar to the sequencers described for the geometry server.

A.2.2 Grid Point Functions

Grid points are defined to have the functions listed in Figure 24. **GridPointCoord** returns the desired coordinate value. The second function is a sequencer "function" on regions at a point, similar to the ones described above. **GridPointSetSolution** and **GridPointGetSolution** set and get a solution value at the grid point associated with the specified region.

A.2.3 Element Functions

Figure 25 lists functions defined over elements. **ElementPoint** returns a handle to a grid point for the specified element. The *gridPointNumber* argument is the local point number. **ElementRegion** returns the element's region. **ElementNeighbor** returns a handle to a neighbor element. **ElementType** returns the element's type, for example a 3 noded triangle or a 20 noded hexahedron. The boundary condition along the specified face of the element is returned by **ElementFaceBoundaryCondition**. Finally, **ElementSetSolution** and **ElementGetSolution** set and get specified solutions on the element.

A.3 Surface Mesher Interface

The interface for a surface mesh server is the same as for the grid server. The functions have one-to-one correspondence. The naming convention is different however. **Grid** in all the names is replaced with **SurfaceMesh**; **GridPoint** is replaced with **MeshPoint**.

References

[1] D. S. Boning, M. L. Heytens, and A. S. Wong, "The Intertool Profile Interchange Format: An Object-Oriented Approach." IEEE Trans CAD. vol 10. no. 9. pp 1150-1156. Sep 1991.

[2] P. Conti, N. Hitschfeld, and W. Fichtner, "Omega -an octree-based mixed element grid allocator for the simulation of complex 3-D device structures." IEEE Trans CAD. vol. 10, no. 10, pp 1231-41. Oct. 1991.

[3] C. H. Corbex, A. F. Gerodolle, S. P. Martin and A. R. Poncet, "Data Structuring for Process and Device Simulations." IEEE Trans CAD. vol. 7. no. 4. pp 489-500. Apr 1988.

[4] S. G. Duvall, "An Interchange Format for Process and Device Simulation." IEEE Trans CAD. vol. 7. no. 7. pp 741-754. Jul 1988.

[5] F. Fasching, C. Fischer, S. Selberherr, H. Stippel, W. Tuppa and H. Read, "A PIF Implementation for TCAD Purposes." In *Proceedings of 4th Intl Conf on Simulation of Semiconductor Devices and Processes.* Zurich. Switzerland. pp 477-482. Sept. 1991.

[6] M. D. Giles, D. S. Boning, G. R. Chin, W. C. Dietrich Jr., M. S. Karasick, M. E. Law, P. K. Mozumder, L. R. Nackman, V. T. Rajan, D. M. H. Walker, R. H. Wang, and A. S. Wong, "Semiconductor Wafer Representation for TCAD." IEEE Trans CAD. vol. 13. no. 1. pp. 82-95. Jan 1994.

[7] P. A. Gough, M. K. Johnson, P. Walker, and H. Hermans, "An Integrated Device Design Environment for Semiconductors." IEEE Trans CAD. vol. 10. no. 6. pp 808-821. Jun 1991.

[8] J. J. Helmsen, E. W. Scheckler, A. R. Neureuther, and C. H. Sequin, “An efficient loop detection and removal algorithm for 3D surface-based lithography simulation.” Proc. of Intl Workshop on Numerical Modeling of Processes and Devices for Integrated Circuits. (NUPAD IV). IEEE, 1992. pp. 3-8.

[9] M. E. Law, and R. W. Dutton, “Verification of Analytic Point Defect Models Using SUPREM-IV.” IEEE Trans on CAD. Vol. 2 No. 2. pp. 181-190. Feb 1988

[10] M. E. Law, “FLorida Object Oriented Process Simulator Manual” Dept of Elect Eng, Univ. of Florida. 1994.

[11] J. P. McVittie, J. C. Rey, and K. C. Saraswat, “Speedie User's Manual. Version 0.3.” Stanford Univ., Stanford, CA Mar 1991.

[12] W. G. Oldham, A. R. Neureuther, C. Sung, J. L. Reynolds, and S. N. Nandgaonakar, “A general simulator for VLSI lithography and etching processes: Part II-Application to deposition and etching.” IEEE Trans Electron Devices, vol. ED-27. pp. 1455-1459. Aug 1980.

[13] J. Pelka, “Simulation of ion-enhanced dry-etch processes.” Proc. of SPIE. Advanced Techniques for Integrated Circuit Processing. Vol. 1392. pp 55-66. Oct. 1990.

[14] L. Piegl and W. Tiller, “Curve and surface constructions using rational B-splines.” Computer Aided Design. vol. 19, no.9 pp 485-98. Nov. 1987.

[15] M. R. Pinto, D. M. Boulin, C. S. Rafferty, R. K. Smith, W. M. Coughran, Jr., I. C. Kizilyalli, and M. J. Thoma, “Three-dimensional characterization of bipolar transistors in a submicron BiCMOS technology using integrated process and device simulation.” IEDM'92 Tech Digest. pp 923-926. Dec. 1992.

[16] Z. H. Sahul, R. W. Dutton, and M. Noell, “Grid and Geometry Techniques for Multi-Layer Process Simulation.” Proc. of 5th Intl Conf on Simulation of Semiconductor Devices and Processes. Vienna, Austria. pp 417-420. Sept 1993.

[17] Z. H. Sahul, E. W. McKenna, and R. W. Dutton. “Grid Evolution for Oxidation Simulation Using a Quadtree Based Grid Generator.” Proc. of Intl Workshop on Numerical Modeling of Processes and Devices for Integrated Circuits. (NUPAD V). IEEE, 1994. pp. 155-158

[18] M. R. Simpson, “PRIDE: An Integrated Design Environment for Semiconductor Device Simulation.” IEEE Trans CAD. vol. 10. no. 9. pp 1163-1174. Sep 1991.

[19] P. Sutardja, “Finite-Element Methods for Process Simulation Application to Silicon Oxidation”. PhD Thesis. U.C. Berkeley. May 1988.

[20] K. Wang, F. Rotella, T. Chen, D. Yang, A. Lee, Z. Yu, R. W. Knepper, J. Watt and R. W. Dutton, "Layout-based Extraction of IC Electrical Behavior Models". IEDM Technical Digest 1994. IEEE. Dec 1994.

[21] R. H. Wang, A. Gabara and A. R. Neureuther, "BTU - Berkeley Topography Utilities for Linking Topography and Impurity Profile Simulations." Proc. of Intl Workshop on Numerical Modeling of Processes and Devices for Integrated Circuits. (NUPAD IV). IEEE, 1992. pp. 225-230.

[22] A. S. Wong, and A. R. Neureuther, "The Intertool Profile Interchange Format: A Technology CAD Environment Approach." IEEE Trans CAD. vol. 10. no. 9. pp 1157-1162. Sep 1991.

[23] C. Yang and M. D. Giles, "Architecture and Implementation of 3D field support for Semiconductor Wafer Representation." In *Proc of NUPAD-V* IEEE. Jun 1994.

[24] D. Yergeau, Z. Sahul, E. McKenna and R. Dutton, "HDF-Vset File Format for Persistent Storage of Semiconductor Simulation Data." Technical Report. Integrated Circuits Lab. Stanford University. Dec 1994. (WWW document: http://www-ee.stanford.edu/ee/tcad/programs/hdfvset.html)

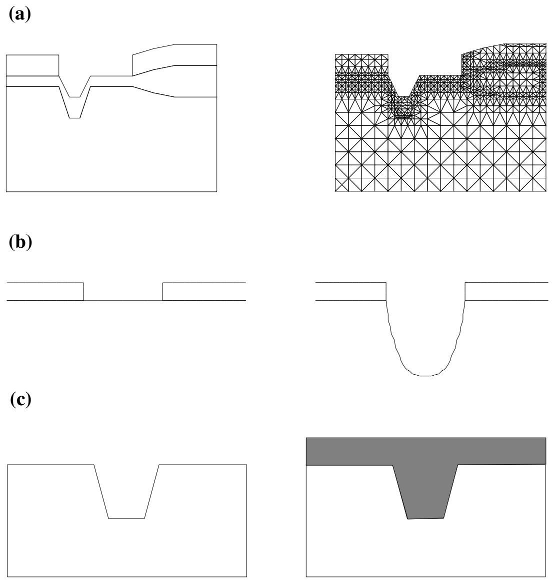

Figure 1: Different tools used in modern day TCAD require different data types. (a) Tools that solve for interior quantities require grids. (b) Tools that solve for boundary quantities require surface meshes. (c) Geometry manipulation tools require a wire geometry.

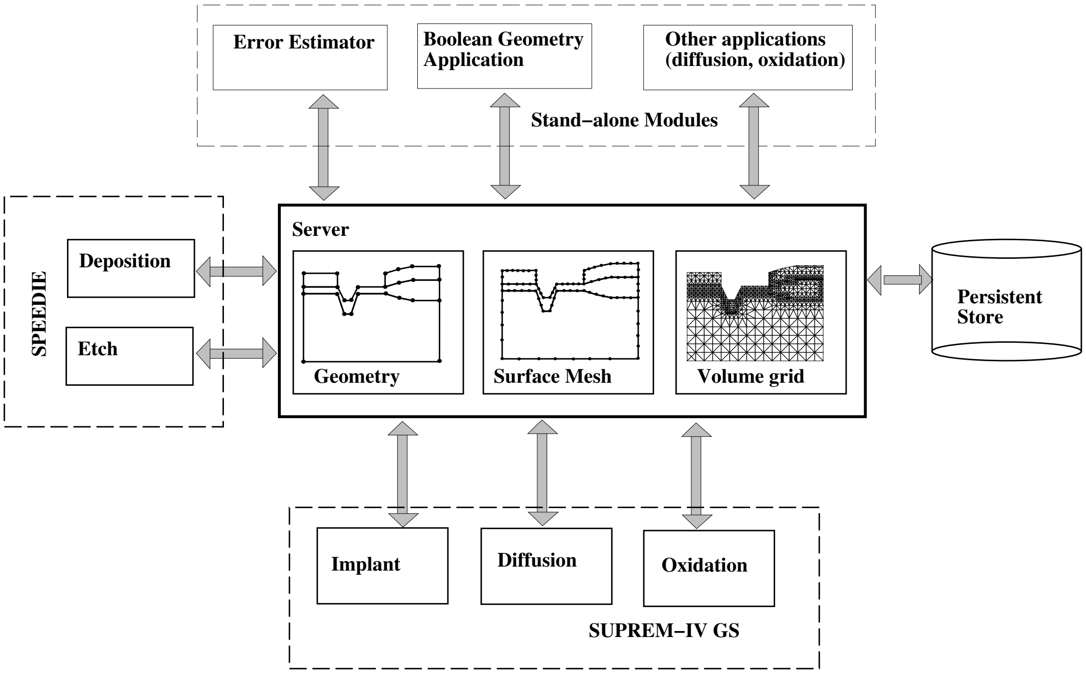

Figure 2: Integration architecture with server and several application tools. Three core modules - geometry modeler, surface mesh generator and grid generator - comprise a server that applications programs can use for data and functionality.

```
#include ``Grid.h''    // server header file

diffuse(double time, double temperature)
{
 Grid *vgrid;
 MyGrid local_grid;           // structure that will hold current grid
 MyField local_fields;        // and concentrations data


 vgrid = GridGenerate( Tri3 );  // server to generate a triangle grid


 // do the time loop

 delta_t = inital_time_step;

 while ( t < time ) {
    if (oxidizing) {
(1)    do_oxide( temperature );

(2)    oxide_t = GridMaxDt(vgrid, delta_t );
      }

(3) ServerToDiffuse(&local_grid,    // Translate data from Server
                   &local_field);  // and get all fields as well

                                                 // solve diffusion step
(4) delta_t = do_diffuse(local_grid, &local_field, oxide_t, temperature);

(5) DiffuseToServer(local_grid, local_field); // update solutions in server

(6) rc = GridMove(vgrid, oxide_t, ``xvel'', ``yvel'', NULL );

(7) t += oxide_t;

(8) error_estimate(adaptname);                // compute error estimates

(9) rc = GridAdapt(vgrid, 1, adaptname);     // have server adapt the grid

  } // end of while loop
} // end of function diffuse()
```

Figure 3: Code fragment illustrating information exchange between server and multiple applications for a coupled oxidation/diffusion simulation.

```
void error_estimate( char **adaptname )
{
 EEGrid ee_grid;  EEField ee_field;  Grid *vgrid;

    vgrid = GridGenerate( Tri3 );

                        /* first translate server data to estimator data */
(10) ServerToErrorEstimator(&ee_grid, &ee_field);

    /* do the actual computation and mark elements for adaptation */
(11) do_estimate(&ee_grid, &ee_field, adaptname);

                         /* back to server */
(12) ErrorEstimatorToServer(ee_grid, ee_field);
}

void do_oxide( double temperature )
{
 OxideGrid  oxgrid;  OxideField oxfield;  Grid *vgrid;

(13) vgrid = GridGenerate( Tri7 );          // generate quadratic elements

(14) ServerToOxide(&oxgrid, &oxfield);    // translate to oxide info model

(15) ComputeVelocities(oxgrid, &oxfield); // acutal computation done here

(16) OxideToServer(oxgrid, oxfield);      // send velocities back to server

} /* end of function do_oxide */
```

Figure 4: Continuation of code fragment from Figure 3

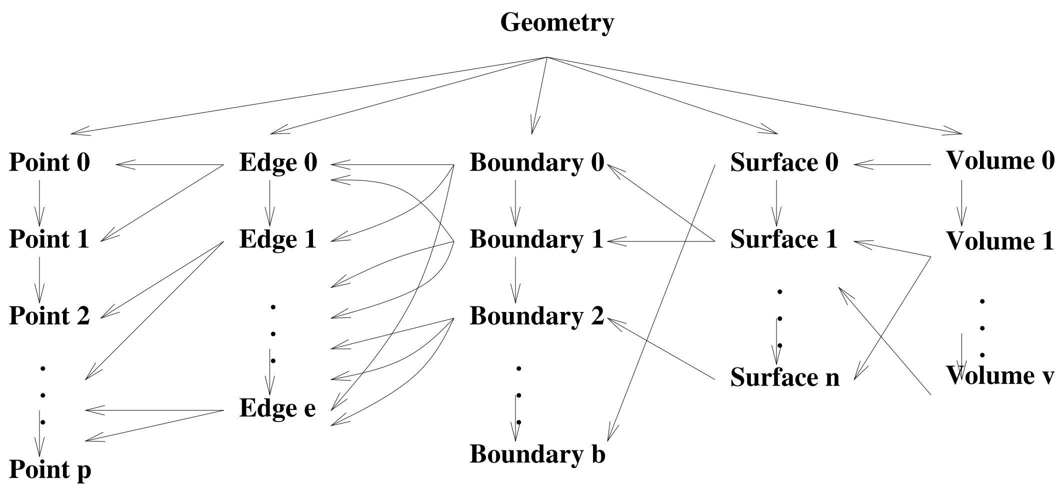

Figure 5: Boundary representation geometry server information model for 3D geometries in *Forest*. Arrows represent handles. The tail of the arrow is from an object that holds the handle(s) of object(s) pointed by the head of arrow.

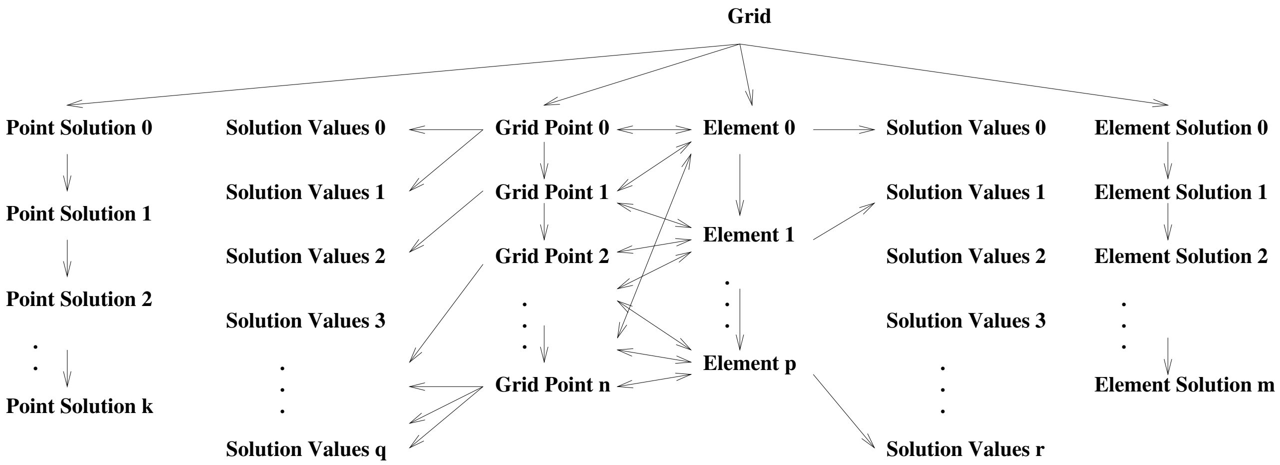

Figure 6: Information model of the *Forest* grid server. Grids are a collection of elements and grid points with solutions and solution values defined over them. Note that solution values can be accessed only via its parent element or grid point.

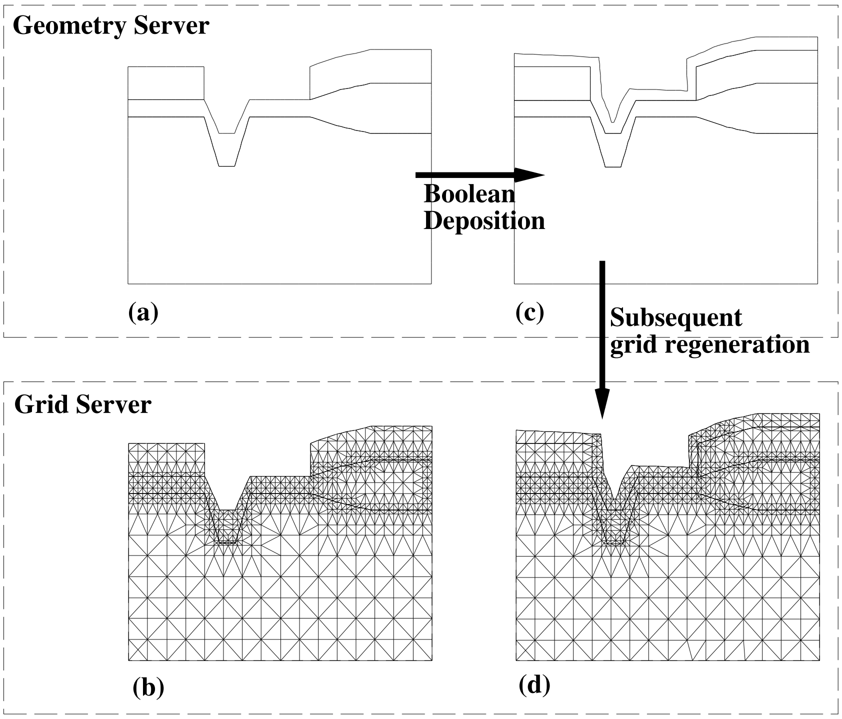

Figure 7: Illustration of boolean applications using the new integration framework. Operations are performed with the geometry server. The grid is generated subsequently as needed.

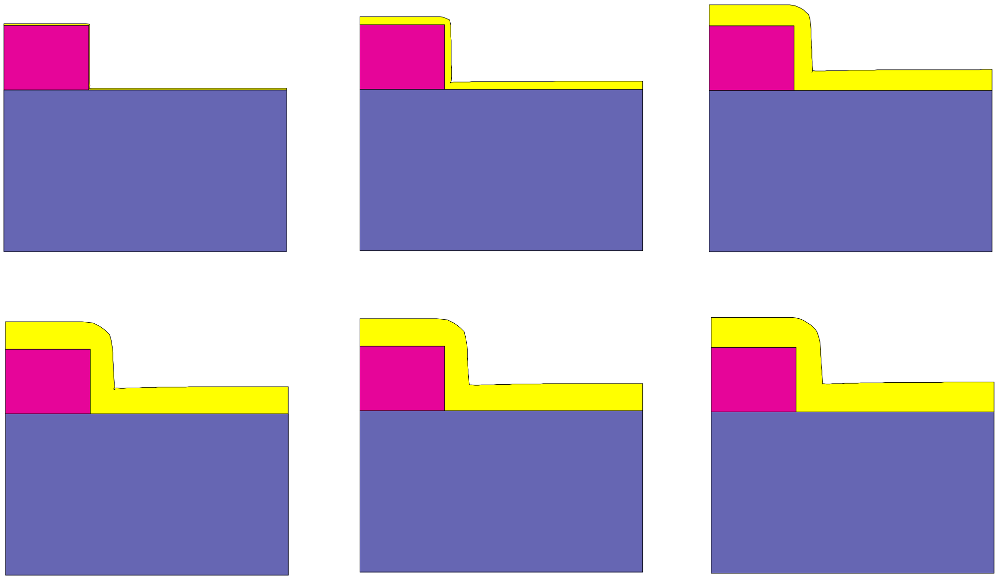

Figure 8: Example use of deposition solver (SPEEDIE) with the new integration framework. Several frames during a spacer oxide deposition simulation are shown. The geometry server removes loops that are formed as material is deposited at each timestep.

Operation	CPU time (s)
Get Data from Server	1.4
SPEEDIE Velocity Calculation	208.1
Server Update	10.7

Figure 9: CPU usage for the SPEEDIE oxide deposition example shown in Figure 8. 0.04 micron long surface mesh elements ($\simeq$ 100 mesh points) were used to deposit a 0.25 micron thick layer. 28 timesteps were used to complete the simulation.

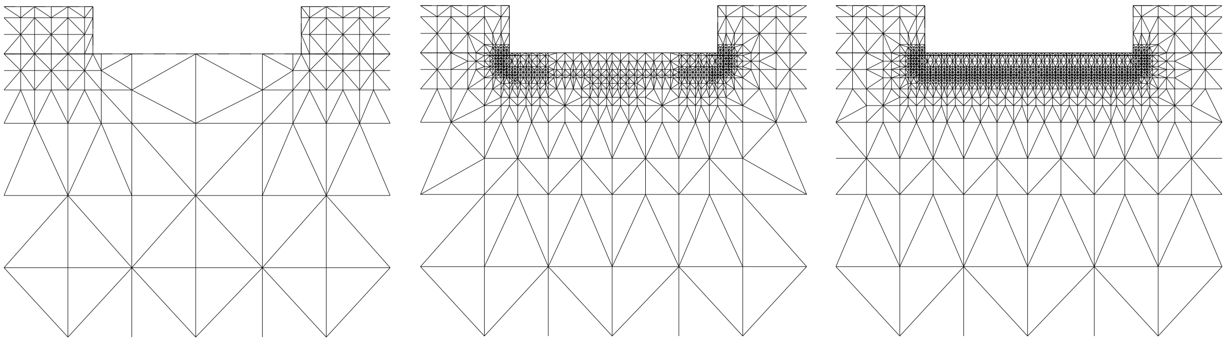

Figure 10: Example use of ion implantation with grid adaption in the new integration framework. The implantation module is from SUPREM IV. The error indicator is a new application. Grid adaptation is performed by the grid server.

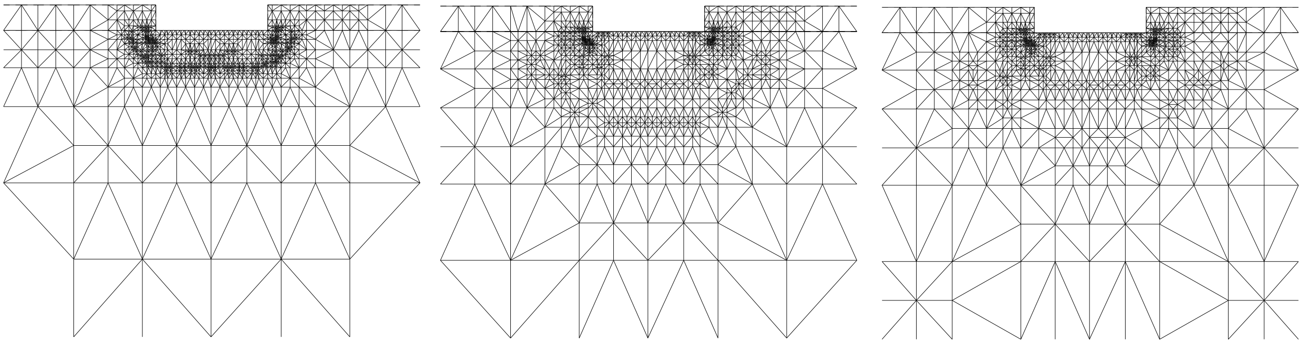

Figure 11: Example use of diffusion with grid adaption in the new integration framework. The diffusion module is from SUPREM IV. The error indicator application is the same one used for ion implantation. Grid adaptation is performed by the grid server.

			Server Access Time				
No of grid nodes	Defect Model	No of time steps	Data get time (sec)	Local data build time (sec)	Server update time (sec)	Total simulation time (sec)	% Server access time
736	Fermi	12	5.9	15.9	4.3	132.5	19.7
894	Fermi	13	8.2	22.1	5.6	160.4	22.4
1631	Fermi	17	20.5	58.5	22.2	461.5	21.9
894	Transient	15	9.2	25.3	6.5	342.2	12.0
894	Fully Coupled	15	9.2	25.3	6.6	370.2	11.1

Figure 12: Comparison of server access time with total 2D diffusion simulation time using SUPREM-IV and *Forest*. Boron was implanted through an oxide window (dose = 1e14, energy = 40keV) and driven in at 1100C for 30 minutes.

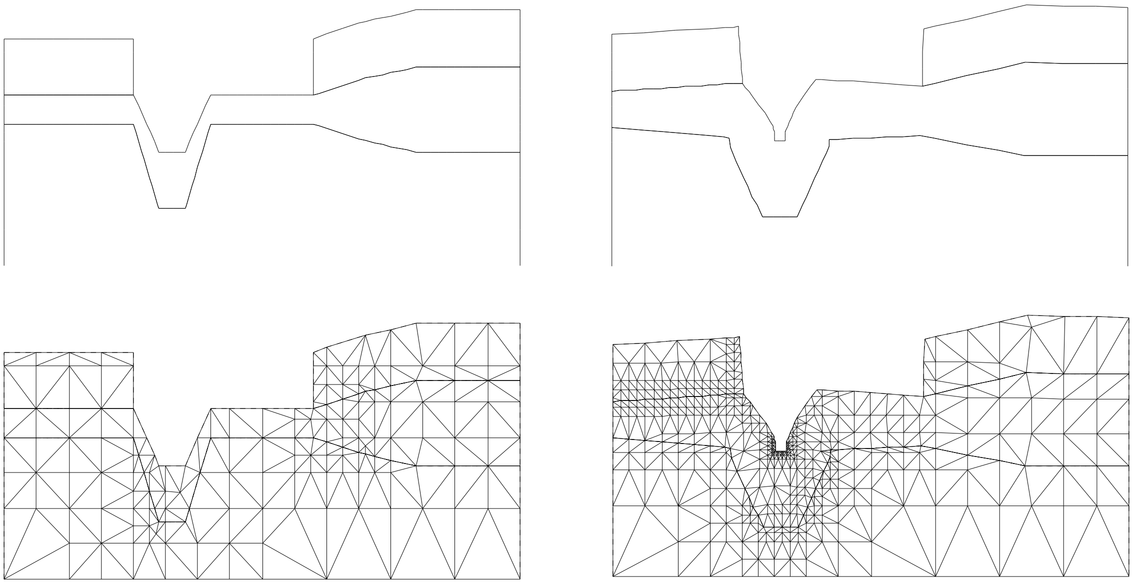

Figure 13: Example use of oxidation/diffusion solver with the new integration framework. The diffusion and oxidation modules are both from SUPREM IV. Grid movement and geometry updates are performed by the grid and geometry servers respectively.

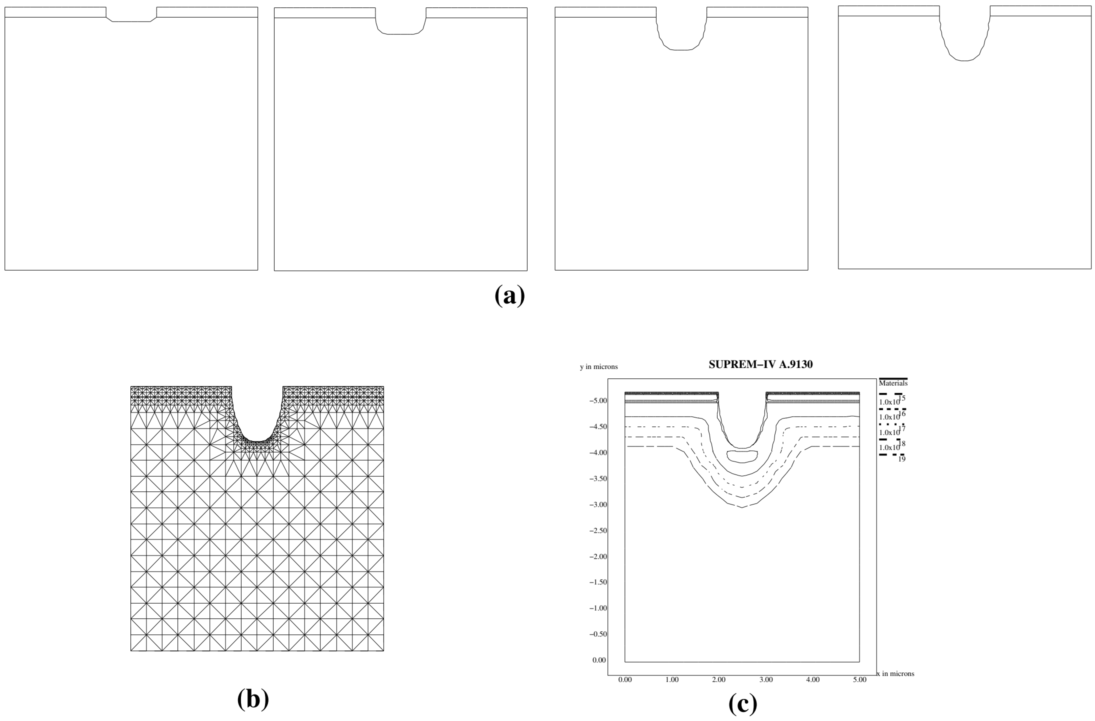

Figure 14: Use of SPEEDIE and SUPREM with *Forest* for etching, implantation and diffusion simulations. (a) Geometry frames during etching simulation. (b) Regeneration of volume grid after etch simulation. (c) Contour plots from boron implantation and drive-in simulations using SUPREM-IV.

1. Geometry ***GeometryRead**(int *numFilenames*, char ***filenames*)
2. int **GeometryStore**(Geometry *, int *numFilenames*, char ***filenames*)
3. int **ReplaceGeometry**(Geometry **oldg*, Geometry **newg*)
4. int **GeometryValid**(Geometry *)
5. int **GeometryMove**(Geometry *, double *time*)
6. double **GeometryMaxDt**(Geometry *, double *timeLimit*)
7. int **GeometryRepair**(Geometry *)
8. **ForEachPointInGeometry**(Geometry *, Point *)
9. **ForEachEdgeInGeometry**(Geometry *, Edge *)
10. **ForEachBoundaryInGeometry**(Geometry *, Boundary *)
11. **ForEachRegionInGeometry**(Geometry *, Region *)
12. **ForEachSurfaceInGeometry**(Geometry *, Surface *)
13. **ForEachVolumeInGeometry**(Geometry *, Volume *)

Figure 15: List of geometry server functions.

```
#include ``Point.h''
#include ``Geometry.h''

void MyFunction(Geometry *mygeom)
{
 Point *mypoint;
 double xcoord_sum = 0.0;

 ForEachPointInGeometry(mygeom, mypoint)
   {
    xcoord_sum += PointCoord(mypoint, X);
   }

 cout << ``total = '' << xcoord_sum << ``\n'';
}
```

Figure 16: Code fragment illustrating use of iterators.

1. double **PointCoord**(Point *, int *axis*) // return the coordinate
2. double **PointVelocity**(Point *, int *axis*) // return the velocity
3. double **PointDisplacement**(Point *, int *axis*) // return the displacement
4. void **PointSetCoord**(Point *, int *axis*, double *val*) // set coordinate
5. void **PointSetVelocity**(Point *, int *axis*, double *val*) // set velocity
6. void **PointSetDisplacement**(Point *, int *axis*, double *val*) // set displacement

Figure 17: Utility functions to support geometric point objects.

1. Point ***EdgeStartPoint**(Edge *) // return the two points
2. Point ***EdgeEndPoint**(Edge *) // that make the edge
3. void **EdgeForm**(Edge *, Point **startp*, Point **endp*) // construct edge from 2 points

Figure 18: Utility functions to support geometric edge objects.

1. int **BoundaryNoofEdges**(Boundary *) // how many edges on the boundary?
2. Edge ***BoundaryEdge**(Boundary *, int)
3. void **BoundaryForm**(Boundary *, int *numEdges*, Edge ***edgeptrs*) // construct boundary from edges

Figure 19: Utility functions to support geometric boundary objects.

1. int **RegionNoofBoundaries**(Region *) // how many boundaries?
2. Boundary ***RegionBoundary**(Region *, int)
3. void **RegionForm**(Region *, int *numBd*, Boundary ***bdPtrs*) // construct regions from boundaries

Figure 20: Utility functions to support geometric region objects.

1. int **SurfaceNoofBoundaries**(Surface *) // how many boundaries?
2. Boundary ***SurfaceBoundary**(Surface *, int)
3. void **SurfaceForm**(Surface *, int *numBd*, Boundary ***bdPtrs*) // construct surfaces from boundaries

Figure 21: Utility functions to support geometric surface objects.

1. int **VolumeNoofSurfaces**(Volume *)
2. Surface ***VolumeSurface**(Volume *, int)
3. void **VolumeForm**(Volume *, int numSf, Surface **sfPtrs) // construct volume from surfaces

Figure 22: Utility functions to support geometric volume objects.

1. Grid ***GridGenerate**(int *gridType*, Geometry *)
2. int **GridStore**(Grid *, int *numFilenames*, char ***filenames*)
3. int **GridValid**(Grid *)
4. int **GridAdapt**(Grid *, int *numNames*, char ***solutionName*)
5. int **GridMove**(Grid *, double *time*, char **xvelName*, char **yvelName*, char **zvelName*)
6. double **GridMaxDt**(Grid *, char **xvelName*, char **yvelName*, char **zvelName*, double *maxTime*)
7. int **GridRepair**(Grid *)
8. int **GridConstruct**(Grid *, int *numPoints*, double **xcoord*, double **ycoord*, double **zcoord*, int *numElements*, int **connectivity*, int **numNodesPerElement*, char ***elementRegion*, char ***boundaryCondition*)
9. int **GridAddSolution**(Grid *, char **solutionName*, int *solutionType*, char **interpolationType*)
10. int **GridRemoveSolution** (Grid *, char **solutionName*)
11. **ForEachGridPoint**(Grid *, GridPoint *)
12. **ForEachElement**(Grid *, Element *)
13. **ForEachPointSolution**(Grid *, char *)
14. **ForEachElementSolution**(Grid *, char *)

Figure 23: List of functions that operate on grids.

1. double **GridPointCoord**(GridPoint *, int *axis*)
2. **ForEachRegionAtGridPoint**(GridPoint *, char **regName*)
3. void **GridPointSetSolution**(GridPoint *, char **regName*, char **solName*, double *solValue*)
4. double **GridPointGetSolution**(GridPoint *, char **regName*, char **solName*)

Figure 24: List of functions that operate on grid points.

1. GridPoint ***ElementGridPoint**(Element *, int *gridPointNumber*)
2. Region ***ElementRegion**(Element *)
3. Element ***ElementNeighbor**(Element *, int *neighborNumber*)
4. int **ElementType**(Element *)
5. char ***ElementFaceBoundaryCondition**(Element *, int *face*)
6. void **ElementSetSolution**(Element *, char **solName*, double *value*)
7. double **ElementGetSolution**(Element *, char **solName*)

Figure 25: List of functions that operate on elements.