C++ Design and Implementation Challenges in Technology Computer Aided Design Frameworks

Goodwin R. Chin (gchin@watson.ibm.com),† Dharini Sitaraman,‡ Chung Yang,‡
and Martin D. Giles (madagil@dip.eecs.umich.edu)‡

†*IBM T. J. Watson Research Center, Yorktown Heights, NY 10598.*

‡*Solid State Electronics Laboratory, University of Michigan, Ann Arbor, MI 48109.*

1 Introduction

Interoperability of physical simulation tools is often cumbersome due to differences in domain representation. The Technology Computer Aided Design (TCAD) community [1] has realized this problem and is working on providing an object-oriented interface for representing and manipulating the state of a semiconductor wafer during process and device simulation, the Semiconductor Wafer Representation (SWR) [1, 2, 3]. We describe an implementation of components of a 3D SWR designed to support the needs of many TCAD simulators. A set of classes provide support for solid modeling operations, mesh refinement operations, and data interchange. A layered memory subsystem allows either uniprocess or client/server operation. Run-time type identification provides dynamic extensibility of data types.

The TCAD world is generally divided into process modeling (simulating the fabrication of semiconductor device structures) and device modeling (simulating the electrical characteristics of these devices). Furthermore, process modeling tools can be partitioned into topography modeling tools which change the physical shape of the device and bulk processing tools which change the material and electrical properties of the device. Examples of topography processes include lithography (pattern transfer of the integrated circuit onto the semiconductor), deposition of electrically insulating and conducting materials, and etching (removal) of these materials. Bulk processing steps include the implantation of chemical species into the bulk substrate material and the diffusion of these species in the substrate to alter electrical characteristics. Steps that involve both topography and bulk processing include thermal oxidation and silicidation — steps common in the fabrication of modern day devices.

Frameworks facilitate the development and use of modeling tools by providing infrastructure and services [4]. Many of these framework services, such as user interface management and intertool communication, are domain-independent in that their value to a framework is independent of the particular application (e.g. electrical CAD or mechanical CAD). Services such as design data representation are domain-dependent in that they are strongly coupled to the application area to be supported. Development of framework standards is being promoted by the CAD Framework Initiative (CFI), a consortium of end users and vendors worldwide, dedicated to defining interface standards for the infrastructure of design automation tools and for design data used by these tools. When vendors adhere to these standards, interoperability between a heterogeneous set of design automation tools is ensured.

The design data representation requirements for TCAD can be supported by using two representation components for wafer state — a geometry to represent the shape of the wafer and a field to represent the values of a particular solution variable at various points in the domain. Figure 1 shows a 2-D cross section of a wafer with the geometry and field portions denoted. Field values are usually associated with a mesh across the domain that is used to solve partial differential equations that describe the physical processes that are simulated.

[1] TCAD is a subset of physical simulation that primarily focuses on the simulation of the fabrication and electrical testing of semiconductors.

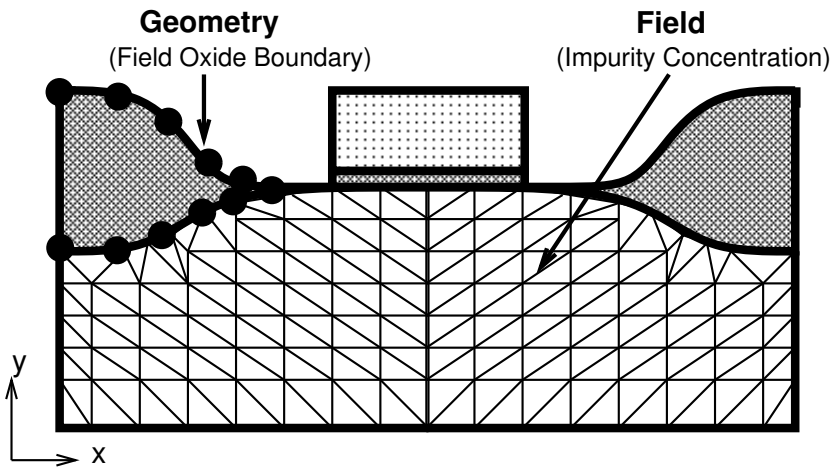

Figure 1: SWR Geometry and Fields

Section 2 describes an implementation strategy to support both geometry and mesh views of the domain. These representations pose different storage/functionality requirements and a need exists to allow easy access to both representations for the same domain. Implementation of classes to support meshing algorithms is discussed with analysis of the benefits of using C++ features such as templates and multiple inheritance with virtual base classes.

Section 3 describes implementation of a new flexible scheme to support field values on generic meshes with multiple interpolation functions. In particular, we present a run-time type library with automatic updating of virtual table pointers to allow sharing of C++ objects across multiple executables. This is an extension of the run-time typing necessary to express user-defined fields.

2 Cells

A *Cell* is a geometric object that can be used to describe the shape of the simulation domain (Figure 1). Cells can be simple (e.g. rectangle) or complex (multi-faceted polyhedral). A *Cell Complex* is formed from a set of cells with a common property (e.g. same material). *Cell Complex*es facilitate implementation of efficient material boundary queries.

Topography-based simulators typically operate in the following fashion:

- extract exposed top surface of the wafer (set of edges for a 2D wafer and a set of faces for a 3D wafer).
- using models, possibly physics-based, determine the new position of the surface. This may require information from the volumes below the surface.
- update the original wafer with the new surface. This may require adding new cells or modifying the boundaries of existing cells as volumes may be modified. Field values that are topography-related may also need to be updated.

Use of solid modeling boolean operations such as subtract and glue (Figure 2) provide robust update of the original wafer. To support efficient boolean operations, rich (in memory usage and functionality) data structures such as the "staredge" [5] are required. Fortunately many domains can be described using relatively few cells.

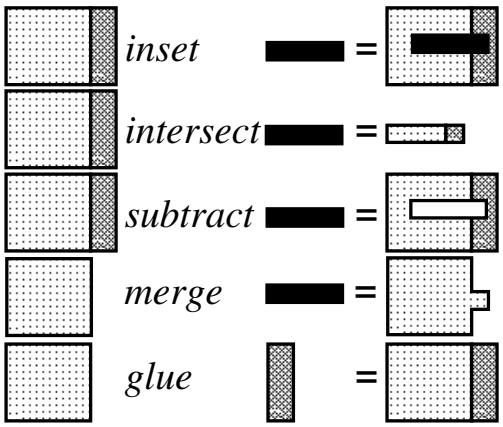

Figure 2: Set Operations

Data Structure	Memory Usage per Tetrahedral
Staredge	1.1 kBytes
Meshing Cells	450 Bytes
Minimal mesh description	76 Bytes

Table 1: Memory usage for a variety of data structures on a typical 3D mesh (729 vertices, 4184 edges, 6528 faces, 3072 tetrahedron)

Field-based simulators often use meshes to describe solution values on the domain and to discretize partial differential equations used to describe physical models (Figure 1). Cells can also be used to describe the elements within a mesh. As the number of elements in a mesh is typically orders of magnitude greater than the number of cells necessary to describe the shape of the domain, a compact data structure is required to minimize storage requirements. The minimal mesh representation adequate for efficient element-based matrix-assembly, a common operation for simulators, is an enumeration of all of the vertices (nodes) in the mesh, a list of elements, where each element contains an ordered list of nodes that define its boundary, and a list of neighboring elements. Table 1 shows the amount of storage necessary to store a tetrahedral mesh using a variety of data structures. Notice the dramatic increase in storage necessary to provide efficient boolean operations using the "staredge" data structure. Even supporting higher-level functionality such as meshing results in a 6-fold increase of storage over a minimalistic representation.

Many meshed-based simulation tools may require another service, mesh refinement, as numerical considerations may require the refinement of an element into a set of smaller elements. To support both mesh refinement and to improve the computational efficiency of matrix assembly, a *MeshingCell* class is introduced in Section 2.1.

Both the geometry cell and meshing cell classes are inherited from an interface base class as shown in Figure 3. EXPRESS-G notation[6] is used to describe relationships between classes[2]. Classes are enclosed in rectangles, with lines indicating inheritance. Relationships are unidirectional with the circle indicating the "to" side of the relationship. Errors in converting between the different representations can be minimized by having both types of cells obtain their vertices from a common pool. This ensures that corresponding vertices in the geometry and mesh representations correspond to exactly the same object rather than using numerical comparison to ensure that the vertices share the same location in space. Operations common to the base class include connec-

[2] In this paper only inheritance relationships will be shown. Containment relationships can be shown by using thin lines between classes.

tivity information such as location of the cells in space. These operations are sufficient to develop file-based translators to existing TCAD simulators. In addition, a minimal mesh description can be generated for minimal persistent storage.

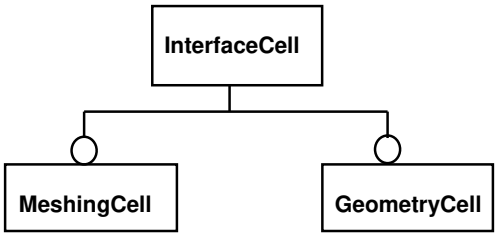

Figure 3: Cell Class Diagram in EXPRESS-G Notation

2.1 MeshingCells

To support the integration of existing meshers and to facilitate the implementation of new meshing algorithms, a set of classes have been developed as shown in Figure 4. A *MeshingCell* contains information to describe its boundary, cells of higher dimension that contain it, and the *Cell Complex* that contains it [3]. For example, edge "B" in Figure 5 would contain references to vertices "C" and "D" that describe its boundary, a reference to face "A" as a parent cell that contains it, and a reference to a cell complex. *MeshingCells* have two template arguments. The first argument is the topological dimension of the cell (e.g. a vertex has topological dimension 0, an edge has topological dimension 1, a face has topological dimension 2) and the second argument describes the dimension of the space the cell is embedded in. For example, an edge in a volume (3-space) would be denoted as *MeshingCell<1,3>*. A *MeshingCell* class might be declared as follows.

```
template <int topologicalDim, int spatialDim>
class MeshingCell
{
  private:
    Set<MeshingCell<topologicalDim-1, spatialDim> > boundary;
    Set<MeshingCell<topologicalDim+1, spatialDim> > parents;

  public:
    MeshingCell(Set<MeshingCell<topologicalDim-1, spatialDim> >&);
    Iterator<MeshingCell<topologicalDim-1, spatialDim> > boundary() const;
};
```

A *ShadowMeshingCell* is similar to a *MeshingCell* but adds additional references to interior subcells to support intermediate steps associated with mesh refinement. Consider a simple face refinement algorithm (Figure 6):

- add vertex in the center of the face (step 1).
- attach edges from the corners to the center point (steps 2 - 5).

The addition of the vertex forms an invalid cell as the point is neither on the inner or outer boundary. However use of the *ShadowMeshingCell* allows the temporary association of the point with the current face. Addition of the first edge again results in an invalid cell [4] — using the *ShadowMeshingCell* the edge can be associated with the face. Addition of the second edge causes the creation of two valid cells.

[3] This data structure is very similar to the "staredge" without explicit enumeration of "neighborhood" information (e.g. given a vertex in a particular Cell Complex, enumerate the faces which contain the vertex).

[4] we do not allow manifold cells in our representation.

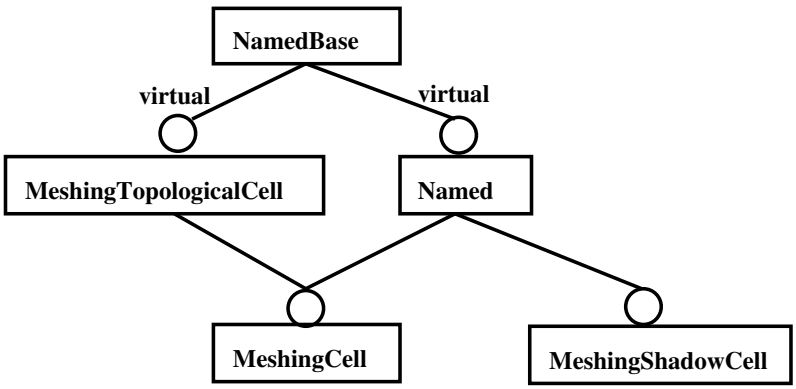

Figure 4: Meshing Classes in EXPRESS-G Notation

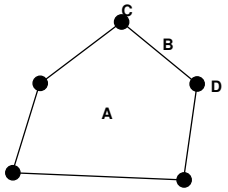

Figure 5: Cell Composition

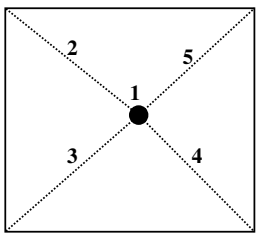

Figure 6: Refining a face

Many meshing algorithms are independent of spatial dimension (i.e. an algorithm for refining a plane should be the same whether the plane is embedded in 2-D or 3-D). However implementation of 2-D cell meshers usually assume that the cell is also embedded in 2-D — and the mesher will not be usable to mesh the surface of a volume. One way to provide surface meshing capability using an existing 2-D mesher is the following.

```
find a transformation that maps the face into the x-y plane
create a MeshingShadowCell<2,2> for the face
  map edges of the face to MeshingShadowCell<1,2> objects
  map vertices of the face to MeshingShadowCell<0,2> objects.
     this will require using the transformation on Point<3>
     objects.
  based on connectivity of the face, clone the connectivity
     hierarchy to the MeshingShadowCells.
extract information from the MeshingShadowCell<2,2> and
  send it to the existing mesher.
```

The *MeshingTopologicalCell* base class provides connectivity information in a spatially independent fashion such as a list of edges that comprise the boundary of a face. This information can be used to clone the structure of the face in the above algorithm.

2.2 Judicious Use of Virtual Base Classes

As discussed in Section 2.1, *MeshingShadowCells* can be used as reduced spatial dimension representations of *MeshingCells*. This relationship is maintained using a name matching scheme. During construction of the *MeshingShadowCell*, name matching between *MeshingTopologicalCells* and *MeshingShadowCells* of lower dimension is necessary as shown in the algorithm for spatial dimensional reduction of an edge of a face:

```
for (each vertex in the shadowed face)
  if (name of vertex == name of origin of edge)
    origin of shadowed edge =  current vertex of face
  else if (name of vertex == name of destination of edge)
    destination of shadowed edge =  current vertex of face
```

Two implementations for naming were investigated and class diagrams are shown in Figures 4 and 7 :

- Using virtual base classes to provide automatic name comparison. Classes *MeshingTopologicalCell*, *MeshingCell*, and *MeshingShadowCell* are all derived from the base class *NamedBase*.
- Explicitly providing comparison operators between *MeshingTopologicalCell* and *MeshingShadowCell* that work on name comparisons. This scheme does not require the use of virtual base classes.

The benefits of the virtual base approach are as follows:

- Virtual base classes make it clear that comparison between *MeshingTopologicalCell* and *MeshingShadowCell* objects is implemented as comparison between objects of the *Name* class and not due to some other criterion (e.g. same address).
- Virtual base classes reduce the amount of coding necessary to add another class which would need to use name comparison with either the *MeshingShadowCell* or *MeshingTopologicalCell* classes.

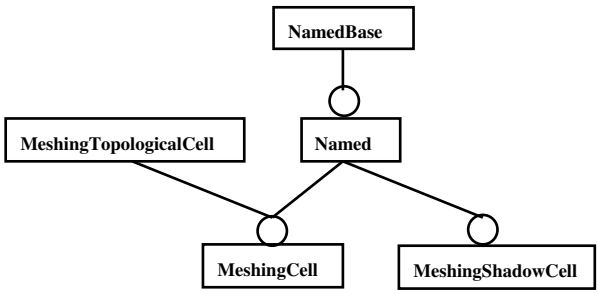

Figure 7: Class Diagram Using No Virtual Base Classes

On the other hand, there is usually a significant overhead necessary to support virtual bases because class-dependent offsets must be computed [5]. This problem is discussed in more detail by Nackman and Barton[7]. As cell shadowing occurs infrequently compared to other more time consuming operations that actually refine the cells, use of virtual base classes does not pose a large penalty in overall performance.

2.3 Use of Templates

Sometimes classes that seem perfect for templatization from a conceptual point of view become less attractive during implementation. Consider a point in space. Points could be templatized by spatial dimension and would possess common methods such as components of their coordinates and distances from other points of the same spatial dimension. A templatized class declaration might look like the following:

```
template <int spatialDim>
class Point  {
  public:
    Point(double* coordinateArray);
    double distance(const Point<spatialDim>&) const;
    double[spatialDim] components() const;
  private:
    double components[spatialDim];
};
```

While extendible from 1D to 3D, the form of the constructor and components method are awkward as the user would be forced to place the coordinates into an array rather than using a more friendly constructor such as `Point<2>::Point(double x, double y)`. As a friendly programming interface is more important than the number lines of code manually typed, the Point classes were implemented using template specialization (e.g. `Point<1>`, `Point<2>`, and `Point<3>` were defined and declared separately).

Another possible approach is to implement the *Point* class using inheritance. A base class containing the distance and components methods could be used with the corresponding derived class responsible for construction and data storage. This approach was rejected due efficiency and storage considerations associated with the use of the virtual table. Use of a virtual pointer requires the use of an extra word of storage for a *Point* — which can be a large percentage of the total space allocated for each point (2 words are necessary to store a 2D point and 3 words are necessary to store a 3D point). For compilers that cannot inline virtual functions, the overhead of calling non-inline versions of virtual methods of simple methods such as components() can be tremendous (100% time overhead on a RS/6000).

[5] Of course, the amount of overhead is implementation dependent.

The *MeshingCell* classes exhibit similar behavior for constructors. For example, a vertex is defined by a point and similarly an edge (*MeshingCell<1,3>*) is defined by a set of vertices. In addition only certain operations apply to particular objects. For example the distance operation only makes sense between two vertices. Users also complained about remembering that *MeshingCell<1,3>* was a edge in 3D. Using typedefs to translate the topological template argument into a more recognizable name help alleviate the above problem. Hence, conceptual templatization can break down in implementation.

3 Fields

A field is a mapping from a domain into a range set. For example, a topographic map displays a field which has as its domain a particular area of the country and as its range set real numbers corresponding to elevation above sea level. In the TCAD world, the field domain is usually the physical space occupied by the semiconductor wafer. The field range set depends on the physical quantity involved, and could be a real number representing atomic concentration, a real vector representing current flow, a complex number representing refractive index, or some larger combination of real and integer components. Most simulation programs use discrete approximations for fields, subdividing the domain into small, regularly shaped elements which comprise a mesh, storing range values at a limited number of locations within the mesh, and using an interpolation rule to estimate the field value at other locations across the domain. The goal of the Fields portion of SWR is to provide objects and methods to efficiently represent and manipulate mesh-based field descriptions for arbitrary range sets.

3.1 Representing Fields

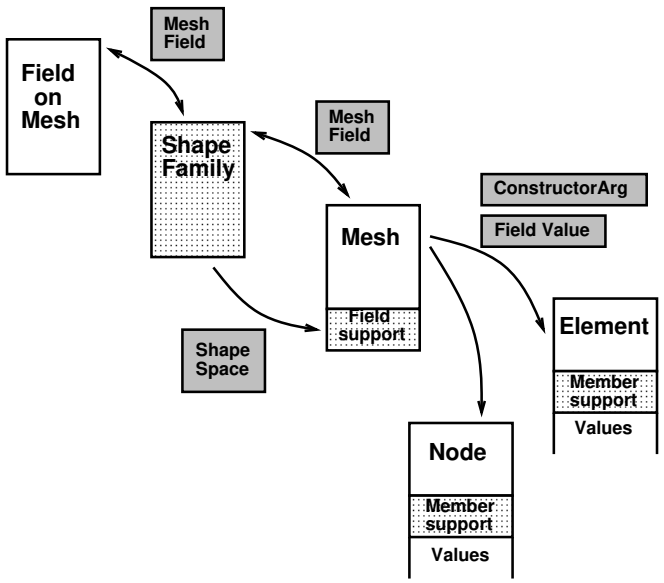

Figure 8: Objects and messages for field representation

An overview of the main field object and message classes is given in Fig. 8, where the arrows indicate messages passing between objects in our system. Field representation is divided into four levels to allow maximum sharing of information between related fields. At the bottom level are individual mesh components such as nodes and elements. A collection of mesh components filling the domain comprise a mesh. All of these objects are parameterized by spatial dimension only using the

template mechanism. The Shape Family level adds information about the storage and interpolation scheme to be used to represent the field values across the domain. This information is specific to the mesh spatial dimension but independent of the value type. At the top level, `FieldOnMesh` adds the range value type information to complete the field description. The components of a `FieldOnMesh` are connected by reference rather than by inheritance so that fields may share common information. For example, a single mesh can be used with different Shape Family interpolation rules, and a single Shape Family with different range value types. This sharing is essential for efficient use of memory since, in practice, most fields across a particular domain will use the same underlying mesh. Messaging classes are used to pass requests and responses between the levels of the field object hierarchy.

There is both good news and bad news in matching the requirements for fields against the capabilities offered by C++. On the positive side, template classes make it relatively simple to allow the user to work with fields of arbitrary range type. Our implementation of `FieldOnMesh` is parameterized by a range class and contains no type-specific information. Particular Shape Families require support for some algebraic operations from the range class so that they can perform interpolations, but if no interpolation is required then the range class has few restrictions. On the negative side, the strong type checking provided by C++ must be sidestepped in two ways. First, range class values are stored with the components of the underlying mesh. At that level there is no value type information available and the mesh components must be prepared to accept whatever mixture of value classes is supplied by the fields that reference them. In practice this does not pose a problem because the user always manipulates field value information through the `FieldOnMesh` class so value type information can always be recovered. Second, the collection of all fields across a domain form a heterogeneous set. Domain field queries can return a `FieldOnMesh` base class, but runtime type identification (RTTI) is necessary to recover the range-specific type. The set of possible range types is determined by the client applications and must be dynamically extensible so new range types can be added without rebuilding the Framework code. Our solution has been to develop a generic RTTI library supporting the operations required for Fields. The design is complicated by the distributed nature of the Framework, which requires the sharing of RTTI information between client processes.

3.2 Runtime Type Identification

Mechanisms for RTTI have been implemented in several C++ libraries (for example [8]), proposed as an extension to the C++ language[9], and finally adopted as an official language extension[10]. The RTTI scheme used here was designed based on the ideas presented by Stroustrup[11] before the language extension proposal was finalized. Our goal was to develop a separate RTTI library which could be applied to several application class hierarchies, which minimized the intrusion into the application code, and which could operate with other Framework components for storage management. A secondary goal was the ability to migrate to native language RTTI support when this eventually becomes available. Our primary application-level functions are packaged as macros to allow this without requiring application source code changes. In our Framework application, RTTI is used for all of the field classes parameterized by value type.

3.2.1 RTTI Interface

RTTI adds two practical capabilities to objects: the ability to downcast from a base class to a derived class in a type-checked way, and the ability to query the type of an object given a base class pointer or reference. Suppose we have a tree of application classes to which we wish to add this capability, such as is illustrated in Fig. 9. Adding our RTTI library requires two changes to the class header file and one change to the class implementation file. First, the class tree must contain the class `SwrTypeBase` as a base class:

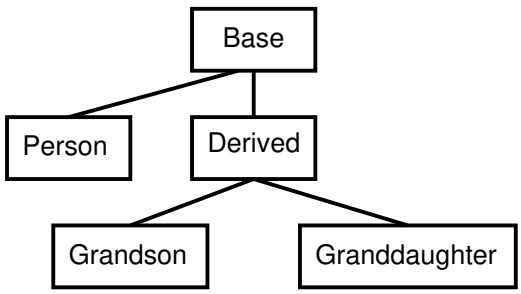

Figure 9: A simple application class hierarchy

```
class Base : public SwrTypeBase
   {
public:
    .......
   };
```

Second, any descendant of `Base` that you wish to cast into must register its type. It does this by calling a macro in the public portion of its class member declaration, passing its own class name as the argument.

```
class Derived : public Base
   {
public:
   TYPEDERIVEDH(Derived)
   ..........
   };
```

If you wish to be able to cast into the middle of a class hierarchy, each derived class must also specify its parent typed class in the hierarchy. It does this using a second form of the registration macro.

```
class Grandson : public Derived
   {
public:
   TYPEDERIVED2H(Grandson, Derived)
   ......
   };
```

Finally, the class implementation file must contain the corresponding implementation macros

```
   TYPEDERIVEDC(Derived)
   TYPEDERIVED2C(Grandson, Derived)
```

Although macros conveniently hide the details of the implementation, the current language specification does not allow them to work smoothly with template class names. If the derived class is a template class with multiple arguments, the comma in the class name confuses the preprocessor because it is taken as an argument separator. The RTTI library defines a preprocessor symbol `COMMA` to work around this. Template arguments are required to appear in two different forms (with and without the argument-declaration portion) so both must be supplied to the macro. The corresponding header and implementation file fragments are:

```
template <class D, class E> class Person : public Base
   {
public:
   TYPEDERIVEDH(Person)
   ........
   };

TYPEDERIVEDCT(Person, class D COMMA class E, D COMMA E)
```

At runtime, pointers and references to base classes can be cast into a derived class. Assuming `br` is a Base reference to a Grandson class object, we can recover the Grandson class object with

```
Grandson& gsr = REFERENCE_CAST(Grandson, br);
```

or get a pointer with

```
Grandson* gsp = POINTER_CAST(Grandson, &br);
```

We can also convert to an intermediate class in the hierarchy:

```
Derived* dp = POINTER_CAST(Derived, &br);
```

All of these calls generate a runtime error if the conversion is invalid (such as when `br` refers to a Granddaughter object). Alternatively, the call

```
Grandson* dp = POINTER_CAST_NOCHECK(Grandson, &br);
```

will return a null pointer is the cast is invalid. `Const` versions of each of these macros are provided.

The casting macros provided by our RTTI library provide an equivalent of the capabilities in the C++ language extension[10, 12]. For example, the reference casting macro could be defined using the extension as

```
#define REFERENCE_CAST(a,b) dynamic_cast<a&>(b)
```

and similarly for each of the other macros.

The second capability provided by RTTI is to allow queries of the type of an object given a base class pointer or reference. The primary purpose here is to allow comparison against a known type id. We provide this functionality using type comparison macros, such as

```
   if (POINTER_TYPE_MATCH(Grandson, &br))
     ....

   if (POINTER_TYPE_MATCH(Derived, &br))
     ....

   if (REFERENCE_TYPE_MATCH(Grandson, br))
     ....
```

In each case, the macro returns `true` only if a conversion is possible to the specified type. In our example above, each of the tests returns `true` because `br` is a Base reference to a Grandson object. This behavior does not match the type comparison scheme adopted in the language extension, which would return `false` for the both of the following tests

```
if (typeid(Derived*) == typeid(&br) ||
    typeid(Derived)  == typeid(br) )
  ....
```

but true for both of

```
if (typeid(Base*)    == typeid(&br) &&
    typeid(Grandson) == typeid(br) )
  ....
```

The language extension compares strictly on the basis of the value of the supplied expression and requires an exact match rather than a possible conversion path. The extension also distinguishes between polymorphic and non-polymorphic types and defines typeid information even for built-in types, which is beyond the capabilities of our library. Nevertheless, the limited form of type matching we provide has so far been sufficient for our application.

3.2.2 RTTI Implementation

Our implementation of RTTI uses static objects to allocate and administer type information, taking care that initialization occurs exactly once for each RTTI class. The `SwrType` class holds the type information for a particular class:

```
template <class T> class SwrType
   {
private:
   static SwrTypeCode t;
public:
   static const SwrTypeCode& code() { return t; }
   static int match(const SwrTypeCode t2) { return t.match(t2); }
   };
```

The constructor for `SwrTypeCode` ensures that each typecode has a unique value. Since `SwrTypeCode` is a static member of `SwrType`, exactly one unique typecode is allocated for each object type.

The `SwrTypeBase` base class adds two virtual functions to the application hierarchy which are then overridden by the `TYPEDERIVED` macros for each derived class that is participating in the typing scheme. These functions use the static type member functions to answer queries about the current object type:

```
class SwrTypeBase
   {
public:
   virtual void* swr_type_match(const SwrTypeCode& ) const;
   virtual const SwrTypeCode& swr_type_code() const;
   };

#define TYPEDERIVED2C(T,P) \
   void* T::swr_type_match(const SwrTypeCode& t) const \
      { return SwrType< T >::match(t) ? (void*)this : P::swr_type_match(t); }\
   const SwrTypeCode& T::swr_type_code() const \
      { return SwrType< T >::code(); }
```

The `swr_type_match` function recursively compares the requested type against the current and immediate base type until a match is found or the top of the hierarchy is reached. The result of a match is the address of the object when viewed as that particular type, or zero if no match is found. This bit pattern is cast into `void*` to maintain a consistent return type, from which a typed pointer can be later recovered without altering the bit pattern.

At the top level, the dynamic type casting macros use the virtual functions to determine the corresponding cast object address.

```
#define POINTERCASTNOCHECK(T,P) ((T*)((P)->swr_type_match(SwrType<T >::code())))
```

The overhead for adding our RTTI library to a set of applications classes is to force each participating class to be polymorphic. In the worst case, this may add storage for one virtual function table pointer for each level in the class hierarchy. Runtime overhead appears at the time a dynamic cast is made, adding one function call and comparison for each level examined in the class hierarchy.

3.3 Sharing Runtime Types

Memory management for the Framework is provided by a library which can be configured in two modes. In the "local process" mode, new objects are allocated on the heap as usual. In the "shared memory" mode, pools of shared memory are managed between client processes to allow sharing of objects. We do not support simultaneous access of objects by multiple clients. Instead, self-contained collections of objects describing the fields of a particular wafer state are passed from client to client.

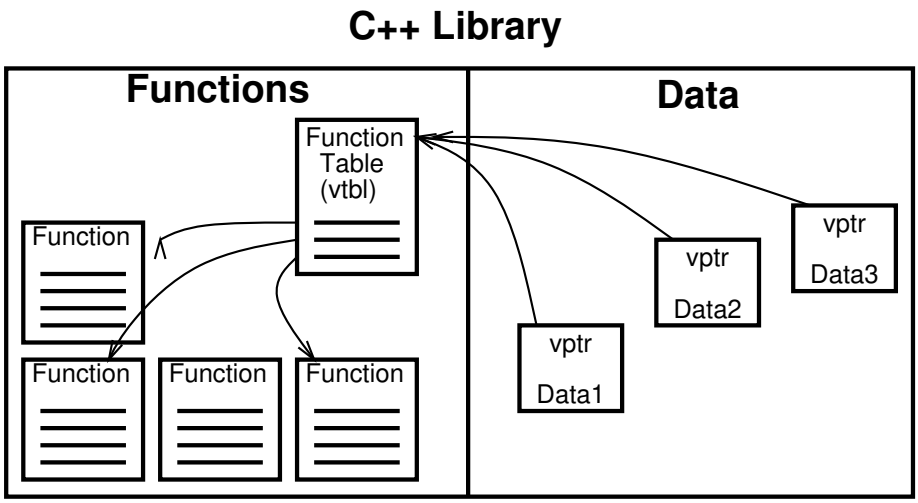

Figure 10: Components of a C++ library object

Sharing between C++ programs is considerably more complicated than with C or FORTRAN. C++ objects conceptually contain both data and functions to operate on the data. Data values can be different for each instance of the class, but the functions are common to all instances of the class. This leads to implementations where the C++ object is divided, with the data stored on the stack or heap and the functions compiled into the executable for the program, as illustrated in Fig. 10. In particular, C++ objects with virtual functions contain at least one virtual function table (vtbl) pointer which is set to the address of the vtbl in the executable which created the object. An object which is identical in all user-defined fields may be represented by a different bit

pattern if created by a different executable. This prevents naïve sharing of objects by saving and restoring dumps of memory locations or, in this case, by sharing a common memory pool between processes.

We have extended our RTTI scheme to allow objects to be "localized" to a particular executable by updating the process-specific information while leaving the other object fields untouched. This shared runtime type identification (SRTTI) allows collections of objects to be passed between clients without converting to an intermediate exchange format or otherwise packing and unpacking the object contents.

3.3.1 SRTTI Interface

Adding the capability to share objects requires two steps beyond the capability provided by the memory management library to allocate object in a shared memory area. First, the runtime type information must be extended to include additional bookkeeping. Second, clients must invoke the object localization procedure when a collection of objects moves under its control.

The change from plain RTTI to SRTTI does not require any source code changes to the application class descriptions. Turning on a preprocessor symbol `SHAREDTYPES` adds functionality to the existing `TYPEDERIVED` macros to include the infrastructure necessary to support type sharing. There are new constraints on the classes which are suitable for participation in SRTTI using the predefined macros: each base class of an SRTTI class must have a default constructor which does not modify the object state. This restriction can be removed if the user is willing to manually expand and modify the macro definition to add dummy base class construction arguments.

In our implementation, we have hidden the object localization procedure inside the mechanism that attaches a client to a pool of shared memory objects. We have assumed that each application will already have a mechanism in place to access every application-level object in the pool. At the time a new client attaches to an existing shared pool, it is required to call the virtual member function `Fixme` for every object in the pool. After this has been done, a final function call registers the current client as the object pool owner. No special action is required for subsequent object creation/deletion, nor when the client releases the memory pool.

3.3.2 SRTTI Implementation

In our plain RTTI scheme, objects are identified by the `SwrTypeCode` value returned by their `swr_type_code` member function. The type code itself is not a data member of the object – the virtual function mechanism is sufficient to ensure that the corresponding `swr_type_code` member function is called. One consequence of this is that each object type must have a corresponding unique virtual function table address because it describes a function that is unique to each typed class. SRTTI operates by building a mapping between type codes and vtbl addresses so that the localization of objects can be performed. Note that an object may contain multiple vtbl pointers. The vtbl used by the SRTTI mechanism is the one corresponding to the `SwrTypeBase` class which is a base class of all participating classes.

The SRTTI scheme operates in three phases. At static initialization time for a particular client, the SRTTI library builds a linked list of `SwrTypeCode`–vtbl pairs for all classes participating in SRTTI. This is accomplished by adding a second static member, `swrlink`, to the `SwrType` class with a constructor which accumulates the pairs. The list is held by a common base class of all `SwrType` classes called `SwrTypeShareBase`. We have also added two member functions to `SwrType`. The `Fixme` method will later be used to localize the object to a new executable. The `SwrTypeMyvtbl` is a compiler-specific routine to obtain a reference to the vtbl for this type which will be used to both read and write the vtbl information. The version shown below is appropriate for our Sun and IBM RS/6000 compilers.

```
template <class T> class SwrType : public SwrTypeShareBase
   {
private:
   static SwrTypeCode t;
   static SwrTypeLink<T> swrlink;
public:
   static SwrTypeCode& code() { return t; }
   static int match(const SwrTypeCode t2) { return t.match(t2); }
   static void Fixme(T* t,sList<SwrTypeList> *tlist);
// this is compiler specific
   static int& SwrTypeMyvtbl(SwrTypeBase& r) { return *(int*)((void*)(&r)); }
   };
```

In order for the `swrlink` constructor to be able to call the `SwrTypeMyvtbl` method, it needs an object of type `T` to work with. The `TYPEDERIVED` macros add a dummy constructor to the application class to facilitate this, but it does constrain the possible application classes to those where a temporary copy of the object can be created and deleted by the SRTTI library.

```
template <class T> SwrTypeLink<T>::SwrTypeLink()
   {
   T t(swrTypeDummyObject);
   SwrTypeShareBase::SwrTypeAddtoList(SwrType<T>::code(),SwrType<T>::SwrTypeMyvtbl(t));
   }
```

The second phase of SRTTI occurs when a new client attaches to an existing shared memory pool. This pool contains the collection of application objects and the linked list of type-vtbl pairs from the originating client. The second client also has its own linked list of type-vtbl pairs with the same type codes but different vtbl addresses. The `SwrTypeBase` class and `TYPEDERIVED` macros have been extended to include a new virtual function `swr_vtbl_fix` which is able to localize all of the vtbl pointers for an object using the dummy constructor and placement new syntax:

```
#define TYPEDERIVED2C(T,P) \
     .... \
   void T::swr_vtbl_fix() \
      { (void) new ((void*)this) T(swrTypeDummyObject); }
```

Initially, the second client is unable to call this virtual function because it too relies on a virtual function table pointer which still refers to the original client. When the application calls the `Fixme` method for a particular object, the (old) vtbl address from the object is used as a key to obtain the type code from the first client pair list. This type code is then used as a key to obtain the new vtbl address from the second client pairlist. After this new vtbl has been written into the object, the `swr_vtbl_fix` virtual function can be called to complete the localization process.

```
template <class T>
void SwrType<T>::Fixme(T* t,sList<SwrTypeList> *tlist)
   {
   int i = getTypeCode(t,tlist);  // map from old vtbl to typecode
   int vptr = getVtblPtr(i);      // map from typecode to new vtbl
   SwrTypeMyvtbl(*t) = vptr;      // bootstrap: first fix one vtbl entry
   t->swr_vtbl_fix();             // use virtual function to fix the rest
   }
```

After every object in the shared memory pool has been updated, the second client type-vtbl list replaces the original copy in shared memory so that it is available for use by a subsequent client.

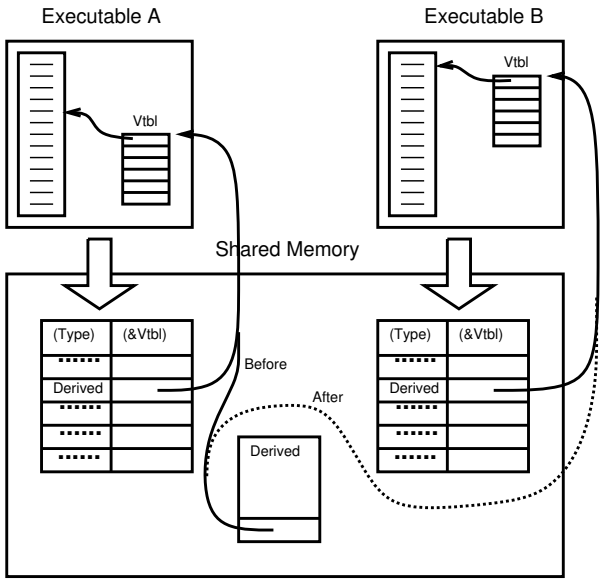

Figure 11: Sharing typed objects: before and after localization by **B**

As an example, consider the case where executable **A** has created an instance of a `Derived` object in shared memory which is now required by executable **B**, as shown in Fig. 11. Shared memory also contains the A-type-vtbl pair list for executable **A** created by the SRTTI scheme. At startup, executable **B** builds its own B-type-vtbl pair list during static initialization with same type information but its own vtbl addresses. When control of the `Derived` object passes from **A** to **B**, executable **B** calls the `Fixme` method in `Derived`, supplying the SRTTI list from executable **A**. `Fixme` uses the old vtlb information from the `Derived` object to look up its type code in the A-type-vtbl list. It then uses the type code to look up the local vtbl address in the B-type-vtbl list and updates the value stored in the `Derived` object. At this point it is free to call any virtual function from the `SwrTypeBase` class. A call to `swr_vtbl_fix` converts any remaining vtbl pointers to their appropriate local values.

SRTTI has an overhead in terms of both space and time. Space must be allocated in shared memory to hold the list of type-vtbl pairs which will have a size proportional to the number of classes participating in the SRTTI scheme. Runtime is consumed during the startup of each client as the local type-vtbl list is constructed, again proportional to the number of participating classes. There is also a runtime cost whenever a client attaches to a shared memory object pool, proportional to the number of objects in the pool. These runtime costs are small compared to alternatives requiring the full object contents to be manipulated.

4 Conclusions

We have successfully implemented components of a 3D TCAD Framework using C++. Language features within C++ were essential for achieving our design goals for development of classes representing meshes, geometries, and fields. Fields require run-time type identification and classes that operate independent of range value type to allow mesh and shape information to be shared among many fields. This scheme can be extended to support the use of shared objects in a client/server architecture by providing the means to update virtual function tables when a new client comes into scope. Cells are used for both mesh and geometry representation and are reconciled through a base class that provides containment and simple neighborhood information suitable for interchange with existing simulators. To support spatially independent mesh refinement, a shadowing scheme

is introduced which requires the use of name comparison. Two different implementations of name comparison trade off flexibility for performance. Some abstractions that lead naturally to template use in theory end up requiring template specialization to make the programming interface more user-friendly. The use of virtual functions for classes with a small number of member data and requiring many instances is very memory inefficient.

While the use of C++ provided an extremely attractive mechanism for problem abstraction along with easily maintainable and debuggable code, some real-world problems still exist with current commercial compilers. Implementation bugs related to templates and specialization are common. Other compilers take an extremely long time to link an executable containing many template classes because multiple instantiations of identical templates are formed during the template instantiation phase. These practical roadblocks are a significant factor weighing against C++ compared to more mature languages, but are not sufficient to outweigh the benefits for our particular application.

References

[1] SWR Working Group of the CFI/TCAD TSC. *Semiconductor Wafer Representation Architecture, Version 1.0.* CAD Framework Initiative, Austin, Texas, July 1992.

[2] SWR Working Group of the CFI/TCAD TSC, editor. *Semiconductor Wafer Representation Procedural Interface, Version 1.0.* CAD Framework Initiative, Austin, Texas, July 1992.

[3] M. D. Giles, D. S. Boning, G. R. Chin, W. C. Dietrich, 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. Computer-Aided Design*, January 1994.

[4] D. S. Harrison, A. R. Newton, R. L. Spickelmier, and T. J. Barnes. Electronic CAD Frameworks. *IEEE Proceedings*, 78(2):393–417, February 1990.

[5] M. Karasick. *On the Representation and Manipulation of Rigid Solids.* PhD thesis, McGill University, Montreal, Quebec, 1988. (available as Cornell Univ. Dept. of Computer Science 89-976, Ithaca, NY).

[6] P. Spiby and D. A. Schenck. Express language reference manual. ISO TC 184/SC4/WG5/P3, 1991.

[7] Lee Nackman and John Barton. Base class composition with multiple derivation and virtual bases. In *Proceesings 1994 C++ USENIX Conference*, April 1994.

[8] K. E. Gorlen, S. M. Orlwo, and P. S. Plexico. *Data Abstraction and Object-Oriented Programming in C++.* Wiley, 1990.

[9] B. Stroustrup and D. Lenkov. Run-time type identification for C++ (revised). In *Proceedings of the Usenix C++ Conference*, August 1992.

[10] B. Stroustrup and D. Lenkov. Run-time type identification for C++ (revised yet again). X3J16/92-0121=WG21/N0198, 1992.

[11] B. Stroustrup. *The C++ Programming Language.* Addison-Wesley, 2nd edition, 1991.

[12] Josée Lajoie. The new language extensions. *C++ Report*, 5(6):47–52, July 1993.