# A General OO-PDE Solver for TCAD Applications*

*D.W. Yergeau and R.W. Dutton*
*{yergeau,dutton}@gloworm.stanford.edu*
Integrated Circuits Laboratory, Stanford University, Stanford CA 94305-4055
*R.J.G. Goossens*
*ronald@ammon.nsc.com*
National Semiconductor Corporation, P.O. Box 58090, Santa Clara, CA 95052-8090

## INTRODUCTION

Technology CAD (TCAD) is a broad area of simulation which is concerned with modeling both the structural and the electrical properties of semiconductor devices. Although TCAD utilizes computational geometry (deposition, etching), monte-carlo methods (ion implantation, "first-principles" device simulation) and other computational approaches, only mesh-based PDE solvers are considered here. The models that these simulators implement have undergone increasing levels of complexity and sophistication. Unfortunately, existing simulators often make assumptions that make it difficult to incorporate new models or take advantage of new computational techniques and technology. To alleviate these difficulties, we are using object-oriented techniques to design a new base code for all of the PDE-based models. To justify the requirements for this code, the PDE models and the limitations of the current simulators are discussed. The requirements for this new base code are laid out. The design methodology and a portion of the final product of the design process are presented. Next, transformation of the design into C++ classes is discussed and an example is presented. Finally, we summarize our experiences in applying the methodology and make some recommendations.

## OVERVIEW OF TCAD

In device simulation, one tries to predict the electrical properties of a device based on given geometric features and material property data. In order to predict the current through a transistor as a function of bias conditions, one needs to calculate the flow of electrons and holes through the device. This is done by solving a system of PDEs. Depending on the degree of accuracy required, the model varies from the simple drift-diffusion equations (Poisson's equation coupled to one or two carrier continuity equations each of which consists of drift, diffusion, and generation/recombination terms) to advanced hydrodynamic formulations, which include lattice and carrier temperatures as well as energy and momentum transfer effects. Although there are several PDE models, calibration of the model is mainly achieved through the constitutive relations (e.g. the drift term and how carrier velocity relates to the electric fields, scattering effects) and the generation/recombination terms (e.g. electron+hole $\Leftrightarrow$ nil, scattering/knock-offs) rather than through modification of the PDEs themselves.

In process simulation, one tries to predict the geometric features and material properties of the device as they result from the manufacturing process. Geometric features are determined both through etching and deposition (these are not modeled with PDEs) and through oxidation steps. The material properties are determined by the distribution of impurity atoms introduced into the semiconductor through implantation. The oxidation problem involves the conversion of silicon to silicon dioxide which is associated with an increase in volume resulting in a viscous flow problem. This flow problem is modeled using a Lagrangian description and uses several different linear and non-linear constitutive models.

Oxidation requires high temperatures ($1000^{o}$C) which lead to a redistribution of impurity atoms due to

---

*This research is supported by ARPA under contracts DAAL03-91-C-0043 and DAABT63-93-C-0053.

solid-phase diffusion. This diffusion has been modeled using several different PDE models. The simplest is an equilibrium self diffusion described by

$$\frac{\partial C}{\partial t} = \nabla \bullet (D \nabla C)$$

Several constitutive models for the diffusion coefficient are used, depending on the impurity type, concentration, self-induced electric field, and other effects. An intermediate model couples each impurity with crystal defects. The most advanced models consist of coupled systems of reaction-advection-diffusion equations with the diffusing species consisting of singular impurities, defects, and pairings of impurities with defects, each of which can exist in several charged states.

As semiconductors were scaled by 2X every three years, more sophisticated physical models were required for the increasing importance of second order effects. However, the codes were never designed with the flexibility to incorporate needed changes. As a result, the addition of new functionality typically requires changes that permeate large portions of the code.

## OO PDE SOLVER FOR DIFFUSION SIMULATION

In order to more easily meet current and future research we decided to use object-oriented techniques to generate a new set of codes. As an initial step, we are developing a base set of modules for simulators that is not tied to any specific set of PDEs and eliminates the problems of our current simulators. To provide a more concrete target and limit the scope of our effort, we chose to tackle the most difficult problem from an equation/formulation perspective, namely non-equilibrium diffusion described by any of several systems of reactive-advective-diffusive equations. While our current simulators use a finite volume approach for discretization, we decided to base our new code on a non-linear finite element approach. Finite elements provide a better division between the equations, i.e. the physics, and the mesh. This is due to the use of integration kernels instead of lengthy premeditated expressions relating flux across edges/surfaces to nodal values. The better separation allows for more code localization. This base code is required to support:

- arbitrary element-based fields
- multiple regions, each with their own set of PDEs and constitutive equations
- consistent treatment of boundary and interface conditions
- formulation via a dial-a-kernel where the PDE operators are described at the weak form level as integration kernels
- field dependent constitutive relations that can be attached to these integration kernels

### *Design Methodology*

The design methodology chosen is that of Shlaer and Mellor [1,2]. It is divided into two areas

- Information modeling. Identification of the objects and the relationships among them. The end products of this step are an annotated hierarchical entity-relationship diagram and documentation defining the objects and relationships.
- State modeling. Identification of the states of the objects and how they progress through the states throughout their lifecycles. The end product of this step is state transition diagrams for the objects.

Although these techniques are common in areas of software engineering concerned with database systems, real-time programming, and even object-oriented analysis, they have not been widely used for numerical software, which has focused mainly on algorithm development. The underlying data and control organization for numerical codes has largely been ignored or approached only from an array and loop perspective.

In the Shlaer and Mellor methodology, both data and control are represented as objects (abstractions of sets of real-world things). Data objects typically represent tangible things: elements, linear systems, and integration kernels. Control objects are used to represent roles, incidents and interactions. These objects include quadrature integrators, nonlinear solvers, and element matrix assemblers.

The information and state modeling steps should be performed iteratively. During these iterations, it is important to ensure not only correctness of the information model and completeness of the state models, but also optimality with respect to an implementation. For correctness, one needs to consider the definitions of the objects, the location of attributes, and the form of the relationships with respect to ensuring that certain algorithms can be implemented. Optimality can be judged by looking at the way data is going to be accessed.

### *Information Model Overview*

Drawing from expertise in the construction of FEM solvers, we went through several iterations until we arrived at a stable model that adequately covered the requirements. We will present only a small portion of the information model in detail. The main reason for doing this is that we wish to present the essence of the methodology, rather than summarize modeling areas. A summary approach would also have the negative effect of confusing entity-relationship diagrams with other graphical representations of structure.

We will concentrate on tangible objects and discuss how the information model meets the goals of the new base code. Unfortunately, the lifecycles of these objects tend to fall into the born-and-die category. Rather than shortchange both modeling areas by trying to cover both in limited space, we will devote our attention only to information modeling. The state modeling aspects of Shlaer and Mellor are most useful for control objects, but we perceive that the state modeling for control objects, such as nonlinear solvers, is more easily derived than the information model, especially if the control object is to implement a role that is given as an algorithm.

### *Information Model Elements*

This section presents a small portion of the objects and relationships in a simplified information structure diagram (Fig. 1). These objects are used to represent element based fields in a form that is natural for a finite element solver. Due to space constraints, hierarchical relationships have been compressed to the abstract object, object names have been shortened, and attributes have been removed.

Within the information model, the associative objects (objects that are used to formalize a many-to-many relationship) need to be distinguished from the ordinary objects. Only the ordinary objects will map directly into classes in an implementation. Associative objects will be folded into either or both of the objects involved in the relationship. We will briefly define the ordinary objects.

- A *Geometry* [GEOM] is the computational domain for a specific coupled problem. For example, a *Geometry* can encompass a semiconductor device with dopant diffusion in the silicon and viscous flow in the oxide parts of the device. A *Geometry* is made up of one or more *Regions*.
- A *Region* [REG] is a subset of the *Geometry* that is computationally homogeneous. There are two sources of inhomogeneity that will lead to a breakup of the *Geometry* into separate *Regions*. First, subdomains may be described by different sets of PDEs and/or constitutive relations linking the *Fields*. Second, in the weak form formulation of the finite element method, a distinction is made between volume and boundary integrals. This implies that the boundary, itself, should be treated as a separate *Region*.
- A *Geometric Element* [GE] is an object that captures the geometric aspects of a mesh element, as defined by its geometric knots in the computational space.
- A *Geometric Element Type* [GET] describes the mapping properties of the geometric transformation between the computation space and an isoparametric element space.
- A *Local Geometric Point* [LGP] is the location of a geometric knot in the isoparametric space.
- A *Spatial Point* [SPPT] is a unique location in the computational space.

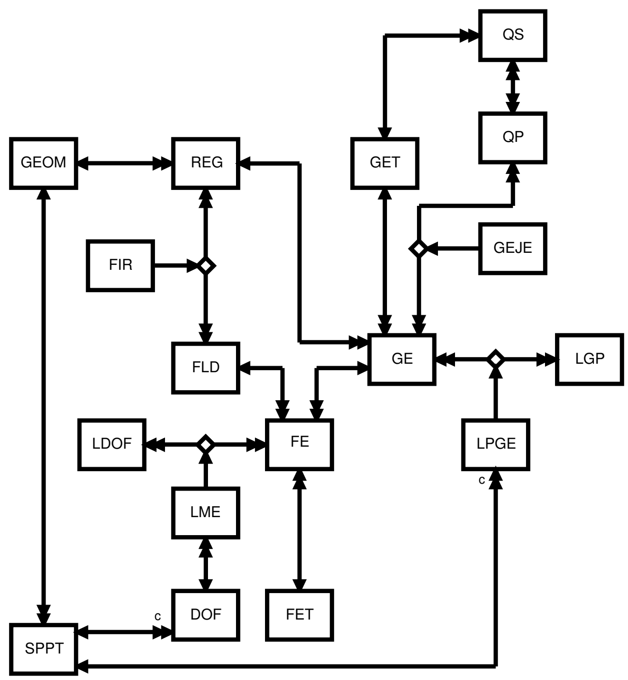


Fig. 1 Simplified information structure diagram for representing fields to be used by a finite-element code.

- A *Field* [FLD] is a discrete (mesh-based) approximation to a spatially varying quantity.
- A *Field Element* [FE] is an object that captures the interpolation of the field within an element, as defined by the degrees of freedom at the field knots in the computational space.
- A *Field Element Type* [FET] describes the mapping properties of the field transform between the isoparametric element space and the field space.
- A *Local Degree Of Freedom* [LDOF] is the value of the field at one of the field knots in the isoparametric space.
- A *Degree of Freedom* [DOF] is the value of a field at a spatial point.
- A *Quadrature Point* [QP] is a location in the isoparametric space where integration kernels will be evaluated to approximate an actual integral.
- A *Quadrature Scheme* [QS] contains the locations in isoparametric space and weights used to numerically integrate a function to a certain degree of accuracy

Associative objects formalize the relationship between two instances of objects that are described by a many-to-many relationship. They may be nothing more than the equivalent of a correlation table, or they may add information that is unique to the specific instances of objects. The *Location Matrix Entry* [LME] is of the correlation-table variety, but the others are true associative objects.

- A *Geometric Element Jacobian (Evaluated)* [GEJE] contains the jacobian determinant (length/area/volume multiplier) for a given *Geometric Element* and *Quadrature Point*.
- The *Field In Region* [FIR] serves a much greater purpose than is presented here and therefore many of its relations have been suppressed in Fig. 1. This is the object which provides the linking of the PDEs and constitutive equations that couple the *Fields* within a *Region*. The PDEs are modeled using integration kernels.

## *Hierarchy in the Information Model*

Although the simplified ISD presents much of the detail needed for an implementation, the hierarchy has been compressed. In Shlaer and Mellor, unlike many other object-oriented techniques, hierarchy is used to identify differences and may or may not imply inheritance. We will examine two of the hierarchical structures that occur in the complete ISD, and discuss the implications of their hierarchy.

The first hierarchical structure is shown in Fig. 2. This distinction is made between interior elements and boundary elements to enable handling of boundary and interface conditions. This structure does not lead to inheritance and could have been modeled as separate relationships from a *Region* to each division of *Geometric Elements*.

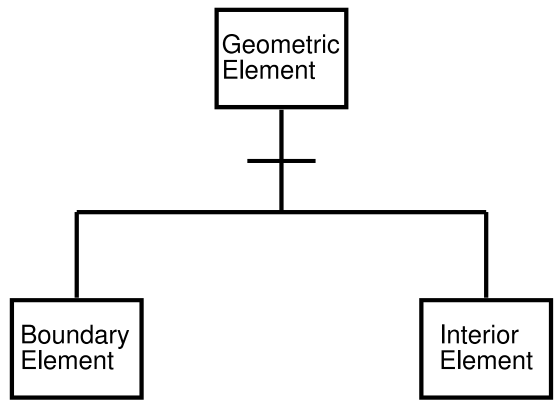


Fig. 2 Hierarchy used in the information model to distinguish between elements that are used in boundary and interior

The second hierarchical structure is shown in Fig. 3. This hierarchy does imply an inheritance structure, providing for common treatment of all instances of *Geometric Element Type* objects.

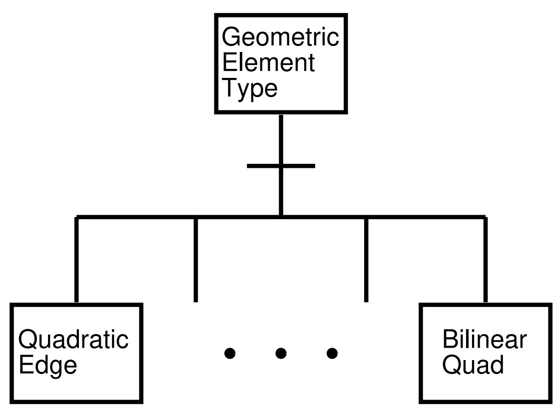


Fig. 3 A hierarchical structure that leads to inheritance.

## *Impact of Design Decisions on the Information Model*

Now that we have formally introduced the information model, we will discuss how design goals have influenced the form of the information model. We will do so by discussing a few examples that demonstrate how the information model can be used to analyze the behavior of a generic implementation based on this ISD.

Although we did not model the complete impact of a moving boundary, it did influence certain design decisions. For example, a *Spatial Point* could have been associated with any of several objects (*Geometry*, *Region*) or simply left dangling. By attaching it to the *Geometry* and enforcing a uniqueness condition, we can simply modify instances of *Spatial Points* to distort elements as the boundary moves. If the *Spatial Points* had been associated with *Regions*, this would not have been possible, and every time the boundary moves, the *Spatial Points* for multiple regions would need to be updated consistently. However, a second effect of moving *Spatial Points* is that, because of the distortion of elements along the boundary, the associated precomputed spatial jacobians should also be updated. Note that the presented information model is incomplete in the sense that it provides neither an explicit nor an easily derived relationship between these two objects.

Another requirement that influenced the information model is the need to support non-conforming field elements (i.e. elements whose field interpolation knots are not at the geometric knots of the element). The viscous flow problem is an example, and requires the pressure field to be a lower order interpolation (e.g. constant over an element) than the displacement fields. This lead to the separation of elements into their geometric and field components.

## Transformation to C++ Classes

As discussed above under "Information Model Elements", all ordinary objects in the ISD directly map into C++ classes. What is left to do, is to determine whether they are part of an inheritance structure. For associative objects, the situation is different. Whether or not such an object maps into a C++ class, depends on whether the association adds distinct attributes or whether it simply represents a correlation.

The first step then is to determine whether an object is ordinary or associative and whether or not the object is involved in an inheritance structure. These properties are trivially extracted from the ISD. The second step considers how relational and other attributes map into members of the class.

A C++ class is produced for each non-associative object. If the non-associative object is the parent in an inheritance structure, then a virtual base class [4] should be used.

Both non-relational and relational attributes lead to the methods defined in the class. Relational attributes provide links to instances of another class.

- One-to-one relationships provide a link to the other instance in either or both classes. The choice of which links to implement depends on whether or not the reference was used in the given direction during the modeling steps.
- One-to-many relationships require a list of links to instances of the class implementing objects on the "many" side. A singular link from the "many" class to instances on the "one" side is provided only if the reference will be used.
- An associative object may be similarly folded into either or both of the classes whose relationship they formalize.

### *Quadrature Scheme*

The *Quadrature Scheme* provides an example of an ordinary object which has relationships with other objects. Before discussing the translation itself, we will present a little more background on the meaning of this object as well as where it fits into "the big picture".

One of the fundamental operations in a finite element code is to integrate the weak form kernels over an individual element's domain ($\Omega^e$) in order to produce entries for the element. The continuous integral in the computational domain is first mapped into a continuous integral in the element's isoparametric domain,

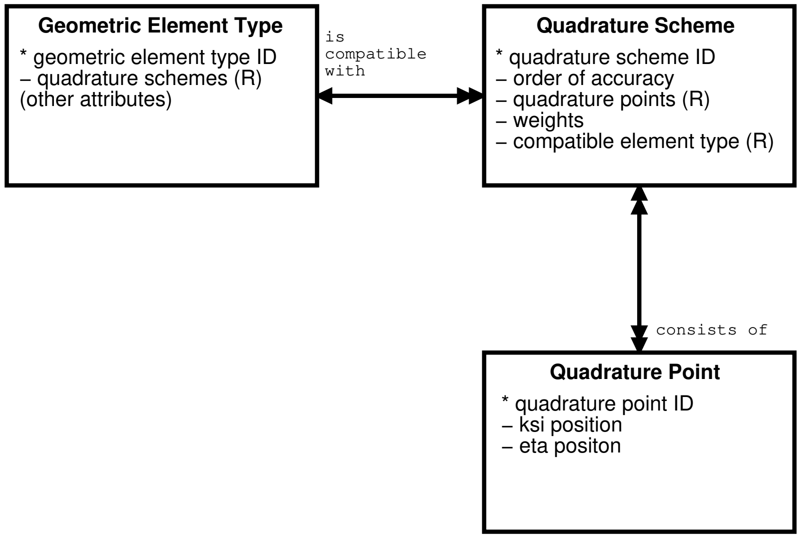


Fig. 4 Expanded view of objects related to *Quadrature Scheme*, including attributes and relationship specification.

which is then approximated by a fixed quadrature rule [3].

$$\int_{\Omega^e} f(x, y)\, dx\, dy = \int_{-1}^{1}\int_{-1}^{1} f(x(\xi, \eta), y(\xi, \eta))\, j(\xi, \eta)\, d\xi\, d\eta$$

$$\approx \sum_{l=1}^{N_{QP}} f(x(\xi_l, \eta_l), y(\xi_l, \eta_l))\, j(\xi_l, \eta_l)\, W_l$$

This operation would be performed by a *Quadrature Integrator* object. Such an object falls into the category of control objects and therefore, as argued above, we chose not to explicitely represent it in the simplified ISD. The *Quadrature Integrator* has relationships linking it to a *Geometric Element*, an *Integration Kernel*, and a *Quadrature Scheme*. A *Geometric Element* provides its mapping between isoparametric space and computational space, including the spatial jacobian $(j(\xi, \eta))$. An *Integration Kernel* $(f(x, y))$ provides a portion of weak form of the operator to be integrated. The *Quadrature Scheme* provides the *Quadrature Points* $(\xi_l, \eta_l)$ and their weights $(W_l)$. Without getting in unnecessary detail about the access methods of all of the objects involved, consider only the *Quadrature Scheme* and the objects that it directly relates to. Fig. 4 shows an expanded view of these object including both key (*) and relevant non-key (-) attributes. Relational attributes are indicated with a "(R)." A partial class definition for *Quadrature Scheme* is:

```
class QuadratureScheme {
public:
    QuadratureScheme();
    QuadratureScheme( const QuadratureScheme& );
    ~QuadratureScheme();
    QuadratureScheme& operator =( const QuadratureScheme& );

    int orderOfAccuracy():
    int nPoints();
```

```
        QuadraturePoint* quadraturePoint( int );
        Real weight( int );
        int isCompatibleWith( GeometricElementType* );
    private:
        int nPoints_;
        int orderOfAccuracy_;
        ListOfQuadraturePoints* quadraturePoints_;
        Real* weights_;
        GeometricElementType* compatibleType_;
    };
```

The first four members (default constructor, copy constructor, destructor, and assignment operator) are present to meet the requirements of orthodox canonical class form [5] which provides for safety in the use of C++ classes. The remainder of the class members provide access to the attributes in the ISD. The specification of the relationships determines the implementation of the member function for relational attributes. For example, the relationship specification "a *Quadrature Scheme* consists of *Quadrature Points*" produces the access function shown. Similarly, the relationship specification "a *Quadrature Scheme* is compatible with a *Geometric Element Type*" produces the query function shown.

Again, we have intentionally left off methods to initialize the data in the class as that would require a discussion of modeling of the dynamics of relationships. Instead, we have simply assumed that both *Quadrature Points* and *Geometric Element Types* are created before and die after *Quadrature Schemes*.

## EXPERIENCES AND SUGGESTIONS

This section is a summary of the lessons that we (painfully) learned and the mistakes we made throughout the modeling process. Much of this is probably common sense, but experience has shown that with respect to software projects, common sense is often one of the first things to go.

- Expect to throw the first few versions out. Use them as a learning experience to help to refine the goals of the design and to evaluate if the domain is too large. Since these versions are expendable, work quickly at first and avoid too much concentration on details.
- Keep in mind the purpose and meaning of the diagrams, especially the ISD. It is an entity-relationship diagram, and should be used only to model relationships between specific instances of objects. It is often tempting to try to use it in a more abstract sense, for example, by viewing the entities as something other than instances of objects. Since this is a major violation of the methodology, the end products are likely to lead to significant problems when it comes to implementation as well as to inconsistencies in the information model.
- Deal with the details in the model instead of pushing them aside to the implementation. If there are difficulties in examining details within the information model, these difficulties are still going to be there when it comes time to implement, and they will probably be amplified by then. *Inability to deal with details within the information model is a sure sign that the information model is not complete.*
- Work with at least one person whose expertise is outside of the domain being modeled. Although this may seem counterproductive, the questions generated will force "the expert" to produce solid definitions of all of the objects within the information model and therefore are likely to highlight any problems that exist.
- Don't prematurely rush to implementation. The information and state models should allow thought experiments and discussion of algorithms and code structure without writing a single line of code or defining a single data structure or class.
- There will probably be far more relationships than will be used in an implementation. This especially seems to be the case with objects used to represent low-level numerical objects. E.g., consider possible relationships between a Region and a Spatial Point or between a Field and a Degree of Freedom.

Although these relationships may have some usefulness in other domains, they would not be used in a finite element simulator. Implementing these relationships would have added unnecessary storage and management overhead to an implementation. They are also derivable by combining other relationships. We suggest only identifying and documenting the relationships that are critical to an implementation. This will also help to keep the information model to a size and complexity that is easily dealt with.

- The fact that a given set of objects and relationships seems correct does not mean that it is optimal. Try many variations and select the one that works best for the way you will use the objects in an implementation. It is far easier to change a diagram and a couple definitions than it is to change hundreds of lines of code. An early version of the model used the following structure (Fig. 5), which although it can be used to model elements supporting multiple fields, it is not very useful from the point of view of element based assembly.

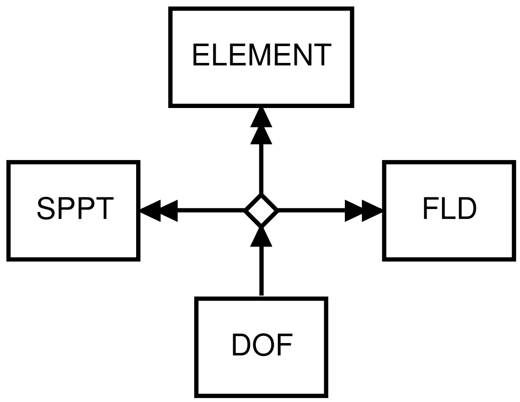


Fig. 5 Early attempt to capture the relation among fields, elements, spatial points and degrees of freedom. Although conceptually attractive, this construct leads to a needlessly complicated implementation.

- Keep documentation of all objects up to date and refer to it frequently when checking the consistency of the design. Without definitions, it is easy to treat objects inconsistently depending on what relationship is being examined. We initially failed to distinguish between an instance of a *Geometric Element* in a mesh and an instance of the mapping properties of an element of that type (*Geometric Element Type*). Although this would probably not have been fatal to an implementation, it was a source of confusion in the discussion of the model.
- Watch out for multi-level associative relationships. An earlier version of the ISD had the structure in Fig. 6. Forms of this type tend to generate multi-level folding without having an intermediate object. This can get very complicated if it is necessary to fold both ways in an implementation. One way to avoid this problem is to introduce a non-associative object and change one of the associations as is shown in Fig. 7,

## SUMMARY

We have presented an object-oriented design for a general PDE solver. We used the methodology as presented by Shlaer and Mellor to do the design. The final design maps very straightforwardly onto an implementation in an object-oriented language such as C++. We have also presented some of the difficulties and potential pitfalls in using Shlaer and Mellor's methodology for numerical applications.

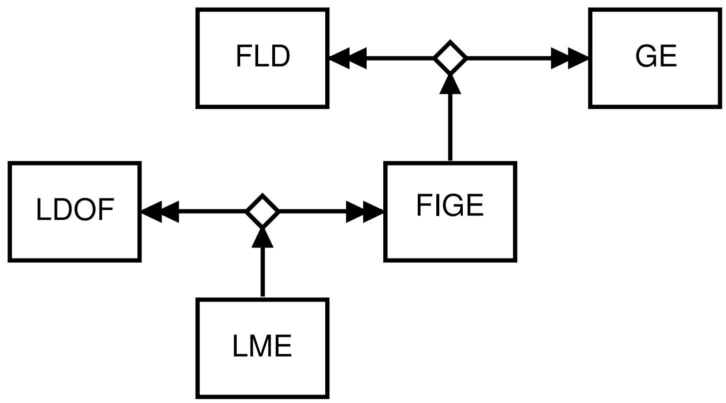


Fig. 6 Example of a multi-level association. (FIGE is *Field In Geometric Element*.)

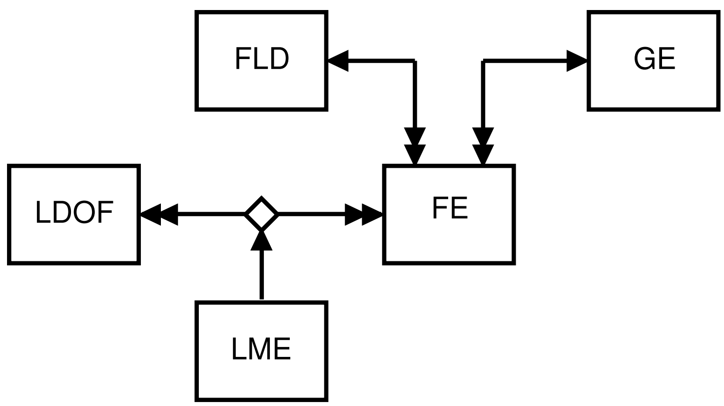


Fig. 7 Possible resolution of the multi-level association of Fig. 6

## ACKNOWLEDGEMENTS

We would like to acknowledge the help of Bob Hartzell, who introduced us to the Shlaer and Mellor methodology and whose critical assessment of early versions of our information model prevented us from making many mistakes in applying the methodology to this project. We would further like to acknowledge fruitful discussions with Dr. Edwin Kan regarding the transformation into C++ classes.

## REFERENCES

1 Sally Shlaer and Stephen J. Mellor, *Object-Oriented Systems Analysis: Modeling the World in Data*, Prentice Hall, Englewood Cliffs, NJ, 1988.

2 Sally Shlaer and Stephen J. Mellor, *Object Lifecycles: Modeling the World in States*, Prentice Hall, Englewood Cliffs, NJ, 1992.

3 Thomas J.R. Hughes, *The Finite Element Method: Linear Static and Dynamic Finite Element Analysis*, Prentice Hall, Englewood Cliffs, NJ, 1987.

4 Margaret A. Ellis and Bjarne Stroustrup, *The Annotated C++ Reference Manual*, Addison-Wesley, Reading, MA, 1990.

5 James O. Coplien, *Advanced C++ Programming Styles and Idioms*, Addison-Wesley, Reading, MA, 1992.