REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
		FINAL REPORT 1 Sep 92 - 31 Aug 96

4. TITLE AND SUBTITLE

OBJECT-ORIENTED FORMULATIONS FOR PARTICLE-IN-CELL (pic) plasma

5. FUNDING NUMBERS

61102F
2301/ES

6. AUTHOR(S)

Dr Rine

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

George Mason University
Fairfax, Virginia 22030-4444

AFOSR-TR-96

0551

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

AFOSR/NE
110 Duncan AVenue Suite B115
Bolling AFB DC 20332-8080

10. SPONSORING / MONITORING AGENCY REPORT NUMBER

F49620-92-J-0478

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION / AVAILABILITY STATEMENT

APPROVED FOR PUBLIC RELEASE: DISTRIBUTION UNLIMITED

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

During this period we completed the development of a World Wide Web version of the Users Manual, Developers Manual, and the Algorithm Manual. Because this is a new generation PIC code, an online manual version was considered appropriate. The on-line Web site for the OOPIC manuals is:

http://ptsg.eecs.berkeley.edu/-peter/manuals.html

In producing a high level design of the OOPIC software, the Object Modeling Technique (OMT) has been used. In constructing the Object Model view, the researchers have broken up the software into two major models, PIC and GUI. The PIC, or physics content modele implements the scientific computations needed to run PIC simulations. The GUI module, represents the MS-Windows software that interacts with the user. The principal reason why OOPIC was made an object-oriented PIC code in C++ is to allow physicists to add new models to the code while at the same time not having to worry about changing the other 99% of the source code.

14. SUBJECT TERMS			15. NUMBER OF PAGES
			16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	

GMU Final Technical Report

1.1 Introduction

This research effort is a collaborative effort between George Mason University, FM Technologies, the University of California at Berkeley, and Berkeley Research Associates. Thus, the project status reflects the work of all the researchers. The particular responsibilites of George Mason University in this effort is to provide overall support for integration of the code among the various groups, and development of the GUI. This project was headed by Prof. James Acquah of the Computer Science Department.

Student researchers at GMU performed the following tasks:

Hadon Nash: Participated in the ongoing dialog with the Berkeley team about data structures which will be shared between the GUI and the expert rules in OOPIC. Hadon also responded to questions from the Knoxville team regarding the GUI architectures, and met with the Technical Director, Dr. Bob Barker to demonstrate the latest features installed in OOPIC. Finally, Hadon installed some high-priority features in the OOPIC GUI that were requested by FMT.

Chao-Chung Lee: Took charge of merging different versions of the code. Reviewed the system control logic (system reset) and debugged some fatal bugs. Added generic label display features (including MKS unit display) to the GUI plot display. Finished and refined batch print magnagement features for multiplot diagrams and clipboard-based copy for Windows word-processing environments. Continued to debug system memory resource allocation problems and put stable GUI classes into the repository.

During the past year, GMU's prime goal was to ensure development of a 32-bit version of PC OOPIC which was as bug-free and reliable as possible.

During this period we completed the development of a World Wide Web version of the Users Manual, Developers Manual, and the Algorithm Manual. Because this is a "new-generation" PIC code, an online manual version was considered appropriate by AFOSR technical supervisor, Dr. Robert Barker.

The on-line Web site for the OOPIC manuals is:

http://ptsg.eecs.berkeley.edu/~peter/manuals.html

2.1 *The Object Modeling Technique*

In producing a high level design of the OOPIC software, the Object Modeling Technique (OMT) has been used. This technique was selected because of its rising popularity among government agencies and its official acceptance as the method of choice by the Defense Information Systems Agency (DISA). Utilizing OMT involves the production of three views for a software system. The following subsection briefly explains and presents the high level object model view for OOPIC.

The object model view represents a diagram of the classes (represented by a rectangular box), along with a number of lines that connect related classes and that convey various relationships among these classes (e.g. inheritance, one-to-one association, etc.). The relations and associations usually have a direct correspondence to the code. For example, an association between two classes is usually implemented by a pointer, or a message (function call). Each box usually contains the name of the class, a list of its attributes (data variables and structures), and a list of its methods (functions). The attribute list includes the name of each attribute, its type, and an optional initial value. The method list includes the name of each method, the names and types of its (possibly empty) list of arguments, and the type of the value returned by the method (possibly void). Attribute and method lists may be omitted from high-level diagrams. The object model view included in this document shows class names only since it constitutes a high level design of the OOPIC system.

OOPIC High Level Design

In constructing the Object Model view, the researchers have broken up the software into two major modules, PIC and GUI. The PIC, or physics content module illustrated in Figure 1, represents the software that implements the scientific computations needed to run PIC simulations. The GUI module, Figure 2, represents the MS-Windows software that interacts with the user. Connecting points between these two modules have been illustrated by an object definition box representing the definition's name. For the sake of clarity and brevity, the attributes and operations have been omitted on these diagrams. At the detailed design and implementation level, each OMT object definition is transformed into a C++ class, and each relation association definition is transformed into an annotated inheritance or client-server relation.

Perhaps the principal reason why OOPIC was made an object-oriented PIC code in C++ is to allow physicists to add new modules to the code while at the same time not having to worry about changing the other 99% of the source code. This chapter will detail how a physicist with only a modest knowledge of C++ might add a new class (or object) to the physics code.

To be specific, let us imagine that a physicist or engineer working at a research laboratory is developing a new kind of microwave device. Let us say that this device requires a very efficient horn to propagate out the microwave power (i.e., any reflections will kill the device operation). Now imagine that he also needs to simulate the device with OOPIC, and that the wave

transmitting boundary for OOPIC is only first order, and hence not good enough to get rid of the waves from the simulation box.

This physicist will now do a literature search and find that a particular kind of algorithm can be used to get rid of electromagnetic waves out of a region. To implement this in OOPIC, the physicist would first look at the class hierarchy in OOPIC and determine what kind of boundary object would be best suitable for his particular boundary to *inherit* from. *Inheritance* is a powerful feature in object-oriented languages since it provides the code developer with the option to add a new object by simply appropriating most of the properties of this object from an already-existing (and similar) object. Then the developer would construct the ".h" file of the new class, and the corresponding ".cpp" file.

Before we deal with the actual implementation of a new class, it will be important to look at some utility routines which OOPIC provides in classes such as vector.h, grid.h, and other such files. The developer should look at these files to see how OOPIC defines important quantities. For instance, in vector.h we have the following helpful definitions:

```
    Vector3(Scalar _x1=0.0, Scalar _x2=0.0, Scalar _x3=0.0)
{x1=_x1; x2=_x2;    x3=_x3;};

    Scalar    e1() {return x1;};
    Scalar    e2() {return x2;};
    Scalar    e3() {return x3;};
```

To a developer what this means is that he can (for example) define a three-dimensional vector named "foo" by the definition

```
    Vector3 foo;
```

To obtain the x-component of this vector he could write

```
    x_component = foo.e1();
```

and similarly with the other components. Similarly, he could set components of a vector by using the definitions:

```
    void set_e1(Scalar _x1) {x1 = _x1;};
    void set_e2(Scalar _x2) {x2 = _x2;};
    void set_e3(Scalar _x3) {x3 = _x3;};
```

Finally, vector classes can be added one to another. The "overloaded" operator "+" is defined by:

```
    Vector3   operator + (Vector3& v) {return Vector3(x1 + v.x1,
x2 + v.x2, x3 + v.x3);};
```

Now let us investigate the grid.h file further. We first note that the units of OOPIC are in mks units, but the code also uses grid coordinates which are dimensionless. Grid coordinates consist of pairs (j+w1, k+w2), where j, k are grid indices and w1, w2 are fractions of the jth, kth cell. Note that in non uniform grids, the mapping is not linear between the coordinates. Routines exist to transform between the two coordinate systems.

```
    inline Vector2 getMKS(Vector2& x);       //MKS from grid
coords
    Vector2   getMKS(int j, int k) {return X[j][k];}; //MKS from
grid                                                  indices
    Vector2   getGridCoords(Vector2& xMKS); //grid coords from
MKS
```

Hence, if one sets up a vector in grid coordinates, say

```
Vector2 foo = Vector2( 2.3, 4.3);
```

then in mks units the vector would simply be

```
Vector2 fooMKS = getMKS(foo);
```

Note that the different member functions in OOPIC take different arguments (i.e., a vector argument might be required in grid coordinates or in mks units). In such a case there are utility member functions to convert from one set of coordinates to the other. The function

```
Vector2   getMKS(int j, int k){return grid->getMKS(j, k);}
Vector2   getMKS(Vector2 x){return grid->getMKS(x);}
```

is overloaded to accept either a pair of grid indices (j,k) or a vector given in grid coordinates and convert it to mks units. Similarly, a vector in mks units can be converted to one in grid coordinates with the function

```
    Vector2   getGridCoords(Vector2 xMKS){return
grid->getGridCoords(xMKS);}
```

2.2 A Look at Some Source Code

Consider the current example of a new wave-transmitting boundary condition. In this case, the physicist might look at the OOPIC class hierarchy (for example, Figure 6 of Chapter 1 in *The Algorithm Manual*) and determine that the **Port** class inherits from the **Boundary** class. The **Port** class is the class from which classes like the OutgoingWave or IncomingWave classes

inherit. This is the logical class from which to inherit any new wave boundary class. In the next section we will investigate how to actually implement the boundary so that it appears in the boundary list exhibited by the GUI.

A documentation file on the **Port** class should look something like this:

Public Methods:
Port(int j1, int k1, int j2, int k2); Constructor
virtual void applyFields(Scalar t);
Method to apply a specified time dependent boundary condition on the electric and magnetic fields or on the current. The argument t (seconds) is the time at which the boundary condition should be applied.
virtual void setPassives(); Method to apply a boundary condition on the electric and magnetic fields which is independent of time.
virtual ParticleList& emit(Scalar dt);
Method to emit particles from the Port object. The return value is a reference to the list of particles that are emitted. The argument dt is the timestep (seconds).
Source File(s):
port.h, port.cpp

Reuse Class Library

The procedure that has been followed to add a class to the reuse library has been to circulate a paper that briefly explains the metrics used in evaluating the software quality of the class submissions. The metrics paper appears in section 2.2 of this manual. In addition, a template for the class documentation file was agreed upon by the researchers and librarian (see section 1.2 for a template and an explanation of the documentation files). Submissions for each class were then evaluated according to the guidelines explained in the paper and were either accepted or returned for review.

The problem of developing quantitative metrics to measure software quality has been a research topic of interest for many computer scientists. The term "quality software" is generally understood to embody attributes such as software that is:

- easy to understand
- easy to use
- less error prone, etc.

Considerable progress has been made in developing quantitative quality measures for software developed using conventional (procedural) programming languages. Furthermore, software quality for structured software has been studied and measured using these measures.

The results are well documented in the literature, and the quantitative measures used in these studies are widely accepted in the software engineering community.

However, Object-Oriented (OO) programming, is a relatively new concept. While the merits of OO programming have been widely recognized and accepted by computer scientists and the software engineering community, research concerning the application, or development of new, quantitative measures, is rare. At the very best, the old measures have been applied to Ada software (Object-Based). In other words, metrics and their associated measurements for use on object-oriented code components have not been formally defined or rigorously validated. While some of the merits of Object-Oriented Programming (OOP) have been widely recognized and accepted by the software engineering community, research concerning the application and development of new quantitative measurements for OOP are few [LORENZ94]. In an attempt to fill the need for OO metrics and measurements, Chidamber and Kemerer [CHIDAMBER94] propose a set of six measurements developed specifically for measuring the quality of OO classes. Another collection of measurements were identified by Fonash [FONASH93] for use with Ada packages. In view of these facts, and due to the need that software developers working on the OOPIC project have for quality metrics, the following section lists and briefly explains the measurements identified from these sources [CHIDAMBER94], [FONASH93].. The list is compiled with the sole intention that the measures be used as a general guide to OOPIC quality programming.