Computing in Civil Engineering

AND GEOGRAPHIC INFORMATION SYSTEMS SYMPOSIUM

Proceedings of the Eighth Conference
held in conjunction with A/E/C Systems '92

sponsored by the
Technical Council on Computer Practices
of the American Society of Civil Engineers

Hyatt Regency Dallas Hotel
Dallas, Texas
June 7-9, 1992

Edited by Barry J. Goodno
and Jeff R. Wright

Published by the
American Society of Civil Engineers
345 East 47th Street
New York, New York 10017-2398

Two Paradigms for OOP Models for Scientific Applications

T.J. Ross[1], Member ASCE, J. P. Morrow[2], L.R. Wagner[3] and G.F. Luger[4]

This paper summarizes an alternative approach using declarative programming methods for the object oriented implementation of numerical codes. The alternative approach is illustrated through a comparison to an approach using the procedural language C++. The paper provides two separate OOP examples: (i) one making use of C++ and its message passing capabilities within an object framework, and (ii) one example which replaces message passing with simple assertions of and requests for object states.

Introduction

Object oriented programming (OOP) is quite promising as a representational tool for large scale scientific code development. Previous research efforts [Peskin, et. al., 1988; Angus and Thompkins, 1989; Forslund, et. al., 1990; Filho and Devloo, 1991; Ross, et. al., 1991] have discussed the use of object oriented programming (OOP) methods in the development and use of such codes. Implicit in these works was the desire to create an OOP environment where user code is easily modifiable and maintainable, where large-scale scientific user code is portable across a variety of architectures, and where the solution process is efficient when compared to serial, vector, or parallel FORTRAN environments. The key to effective OOP implementation in any environment is to create a representative object oriented design of the physical problem and its mathematical abstraction in the first place. This design places a premium on understanding the physical problem from the perspective of class structure, inheritance, and communication.

Constructing object based models requires knowledge of the structure and behavior of the objects in the system and of the interactions between the objects. Many OOP approaches provide constructs which allow for declarative descriptions of object structure, but they insist that the programmer model object behavior and interactions with a procedural "message passing" paradigm. Although procedural descriptions are useful where the intent is to model processes, it may be more natural to describe object system dynamics through declarations of the governing laws.

[1] Assoc. Professor, Civil Engineering Department, Univ. of New Mexico, Albuquerque, NM 87131
[2] Computer Scientist, Air Force Phillips Laboratory, Kirtland AFB, NM 87117-6008.
[3] Graduate Student, Computer Science Department, Univ. of New Mexico, Albuquerque, NM 87131
[4] Professor, Computer Science Department, Univ. of New Mexico, Albuquerque, NM 87131

Given that the current trend in OOP for scientific codes is the procedural message passing approach, we propose an alternative approach using declarative programming methods. To illustrate these approaches, this paper compares two separate OOP paradigms with regard to scientific computing problems with time dependencies: (i) an approach making use of C++ and its message passing capabilities within an object framework, and (ii) an OOP alternative which replaces message passing with simple assertions of and requests for object states. Prior to this comparison we describe a "semantic gap", which represents the chasm between the human and computer in the representation of a problem. The goal of eliminating this gap provides the incentive to explore an alternative OOP paradigm.

In the first approach, called the C++ paradigm, various OOP issues associated with the language C++ are discussed. Among these issues are portability, where user codes need not be altered when moving them to radically different computer architectures (such as from serial machines to vector machines to massively parallel processor systems), and efficiency, where the goal is typically optimum use of CPU time. A simple scientific example is given for this fist paradigm.

The second approach, called the alternative paradigm, provides for the encapsulation of an object's behavioral properties along with its structural properties. Using this alternative, a programming task is decomposed into three functions: 1) describe all pertinent object system structure and behavior, 2) identify external events, and 3) pose queries against the system.

The Semantic Gap

Computers are often used as tools to assist the scientist in understanding physical phenomena. Modelling physical systems with a computer can improve the accuracy of results and provides for a depth of analysis not possible with strictly "human" approaches. But current day computer-based modelling facilities typically require that the scientist perform a substantial translation from his/her "reality model" (the way the scientist sees the world) into an "information system model" (the way the computer requires the world be described). This results in what is known as a semantic gap: the difference between the scientist's description of the problem and the description required by the computer. The larger the semantic gap, the greater the potential for errors in the implementation of the model.

The behavior of physical phenomena is governed by a collection of clearly defined laws. Scientists normally describe these laws in the form of facts and rules, which contain references to the object(s)/attribute(s) of interest. But more often than not, as the scientist sets out to convert his/her reality model into an information system model, these declarative descriptions are necessarily translated into procedures with an associated loss in semantics accompanying the translation.

OOP helps reduce the disparity between the reality model and the information system model by providing the scientist with constructs which allow for a declarative description of object structure. But popular OOP languages (C++, Smalltalk, etc.) nevertheless insist that the scientist describe object behavior and interactions in terms of a procedural "message passing" paradigm. An extension to OOP which allows the scientist to describe object-system dynamics, behaviors of, and interactions between, objects using straightforward declarations of the governing physical facts and rules, could significantly minimize the time and effort required to create computer-based models, as well as improve the accuracy of simulations based upon these models.

The C++ Paradigm

C++ is not a pure object oriented language. One of the key design goals of the language can be stated as "you should not have to pay in storage or CPU time for what you do not use". Most OOP languages were designed with the overriding goal of mirroring a clean object oriented paradigm, without regard for efficiency at the language design level. Programmers pay an efficiency penalty for this.

A more fundamental example of the difference between a pure OOP language and C++ is the ability to inline expand small functions. Consider the problem of adding two matrices (C=A+B). Operator + on matrices is a function call, which itself makes calls to operator [] on matrices (returning a vector) and operator + on vectors. The vector object has a similarly defined operator + and operator []. Also, operator = is a function on both matrices and vectors. That is a lot of potentially expensive function calls for very little actual code. Adding the keyword *inline* in front of a function definition requests that the compiler inline expand the code rather than make a function call. Whether or not a message is a function call or inline expanded has no effect on the semantics of calling that message. This eliminates function call overhead at runtime, but can lead to increased object file size if applied to large functions.

Another feature of C++ is that default name resolution is static, which is to say that objects and messages between them are bound at compile time. In a pure OOPL (e.g., Smalltalk) name resolution is dynamic, which is to say binding is done at runtime and thus incurs a runtime penalty. An overall goal of C++ is to "do the object oriented material that can be done at compile time," therefore bypassing many of the run-time costs involved in the OO paradigm. So long as name resolution is static, good software tools can determine many useful things at compile time. For example, they could determine that many functions within an object semi-lattice will not be accessed by a particular piece of code, thereby greatly reducing the size of object files. In many OO languages, name resolution is far more (often totally) dynamic, and such optimizations are impossible.

Dynamic name resolution, is, however, a very powerful feature of OOP, and C++ does allow this, after a fashion. By declaring a class or function to be "virtual", the user is directing the program to figure out the relevant typing information at run-time. If the user derives everything from a virtual base class, he/she has gained much of the power of the dynamic system, but has lost much of the optimization ability.

Since C++ lets the user choose the type of name resolution, and the compiler knows what is and is not statically resolved, this allows the builder of the library to determine whether these efficiency trade-offs are desirable. When incorporating other people's work, however, different views of the importance of efficiency may lead to performance trade-offs that are not apparent to the user; this is not unique to OOP.

C++ allows the user to define an object which is essentially a fundamental data type. A major goal of a C++ environment is to allow user code to be portable AND efficient across different architectures. For example, on a Cray™ a matrix might be stored as series of row vectors (or column vectors in FORTRAN) with operations set up to increment within the vector on the innermost loop; a matrix addition in FORTRAN for a Cray that incremented across columns instead of across rows within a column on its innermost loop would have no vectorization gain. But on a Connection Machine™, it would be better to store a matrix in block form such that each block is contiguous in memory. Matrix addition is most efficiently implemented in block form, with each processor performing addition on fixed-size submatrices.

An Example

The problem of one-dimensional heat conduction along a rod involves a partial differential equation relating the change in temperature, T, to position, x, along the bar as a function of time, t, and the coefficient of thermal diffusivity, k. The governing equation is of the parabolic form and is given here.

$$k\frac{\partial^2 T}{\partial x^2} = \frac{\partial T}{\partial t}$$

The figures below illustrate the aggregation hierarchies (Figure 1) and the inheritance hierarchies (Figure 2) of the object structure for this simple problem. Each individual node in the discretized model is an object, as would the rod itself. The node object has three attributes and one of these, the node type, has two more attributes - Lagrangian and Eulerian (these are an exclusive-or pair). The rod object is represented by three attributes, length, material, and rod_nodes. The material attribute in this problem actually models the parameter k, but could include general properties for other problems. The aggregation paths are represented by "hasa" in the form of arrows in Figure 1. There are as many rod_nodes as there are nodes in the discretization of the rod. Figure 2 shows the paths of inheritance with "isa" arrows. Note that rod_node inherits properties of the objects node and rod. For this particular problem rod_node has a specific attribute associated with an Eulerian node.type.

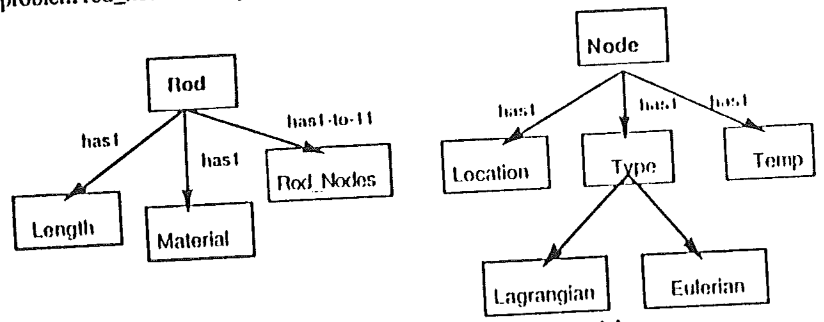

Figure 1. Aggregation Hierarchies

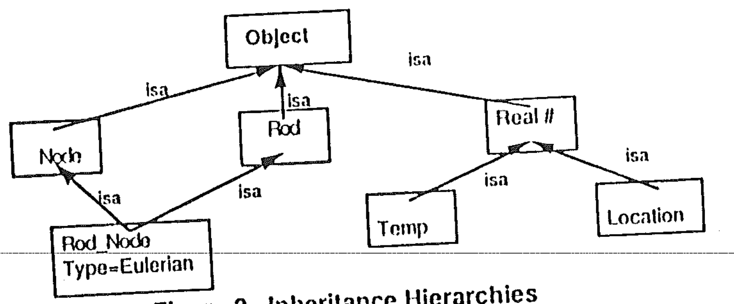

Figure 2. Inheritance Hierarchies

What is interesting about the inner workings of this procedural example is that the explicit solver knows nothing about boundary conditions. It is a simple recursion of the form,

$$T_i^{t+1} = T_i^t + \lambda(T_{i-1}^t - 2T_i^t + T_{i+1}^t)$$

for each spatial coordinate along the rod i, and each time t, and where λ is the stability number of the problem.

The solver is written to treat all rod_nodes as internal. A C++ user code could be written, for example, to heat a rod in the middle, and observe how the temperature radiates outward. The user code would apply a method to the center rod_node rather than to the ends, and initialize this rod_node to the value of the heat source with the other rod_nodes initialized to background. The governing equation (and the explicit solver embedded therein) would be left unchanged.

The implementation of the recursion in C++ is by passing messages to the associated rod_node's temperature attribute to change the temperature of the rod_node according to the formula. The temperature attribute in turn passes a message to the rod_node. If a rod_node is a temperature boundary node, it simply ignores the message. While boundary conditions are not difficult to model in 1D, many 2D and 3D problems can have some complicated geometries on the boundaries. In this object-oriented system, the discrete solver is simply written as if all rod_nodes were on the interior, and the boundary nodes modify the operation or cancel the operation themselves as necessary.

The user never needs to refer to the parameters k or Δt, the increment of each time step. The governing equation knows to set

$$\Delta t = \lambda \frac{\Delta x^2}{k} \qquad \text{where } \lambda = \text{stability constant and } k = \text{largest } k \text{ in any node.}$$

The solver knows the values of Δx and k from querying the rod_node's location and material attributes. The stability constant, λ, can be changed by the user, but also can have a default value. If a user is interested in the rate of convergence, the time step object is queried and a step method is used which applies the solver for one time step. In this one-dimensional example, all nodes are of the same material, but if there was a rod made of two or more materials, the material with the largest coefficient of thermal diffusion would determine the time step for solving the problem. If the user is interested in the steady state temperature distribution, he/she would use a converge-object on the solver, which simply applies the solver at each time step and checks to see if the temperature at any rod_node has changed by more than a user prescribed tolerance. When this condition is no longer met, the method on the converge object will halt.

The Alternative Paradigm

This section contains a discussion of possible extensions to OOP which could appreciably limit the number of declarative-to-procedural translations required by most OOP languages. Additionally, these extensions provide for the encapsulation of an object's behavioral properties along with its structural properties, thereby increasing the ease with which the object may be reused in related applications. This section also demonstrates how conventional message passing can be replaced with simple assertions of and requests for object states. This would simplify the user-interface to something similar to that found in conventional data-base systems.

Our "Alternative Paradigm" is largely derived from the ontological approach for evaluating system design/analysis methodologies described by Wand and Weber (1989). We substitute the word "object" in this paper for the word "thing" used by Wand and Weber. The following points provide a quick overview of the pertinent aspects of the ontological mindset:

• The world is comprised of objects having properties (attributes).
• An object may contain other objects (that is, each object need not be atomic).
• Objects may be organized into an object system (a set of objects).
• An object is modelled in terms of a functional schema (a set of functions), where each function assigns a value to an attribute at a given time.
• The collection of every attribute's value(s) comprises the state of the object (a perceptible change to an object occurred only if its state changed).
• A change in the state of an object is termed an event.
• There are two types of events: external, where another object has asserted a new state for the object and, internal, where the object has changed its own state through the application of its stability laws.
• An object's stability laws define its stable states and its response to those external events which request that the object assume an unstable state.

We augment the ontological approach with a formal treatment of time, similar to that in the infological model discussed by Langefors (1980). In this treatment an object assumes a particular state *at a particular time.*

Our extensions primarily employ a hybrid of the ontological and infological models, where a programming task may be decomposed into three functions: 1) using declarations of the governing facts and rules, describe all pertinent object system structure and behavior, 2) identify external events (changes in the state of one- or-more objects), and 3) pose queries against the system.

Structure and Behavior of the Object System

We describe object structure using conventional inheritance ("isa") and aggregation ("hasa") hierarchies. Object behavior is described using a rule-based syntax (similar to traditional expert-systems) where each rule's head identifies an unstable state, and each rule's body maps the unstable state onto a stable one, i.e.,

```
/* object structure */
window isa object which has size, location, and color.
window.size has width and height.
window.size.width isa integer.
...
window.location has top_left and bottom_right.
window.location.top_left has row and column.
window.location.top_left.row isa integer.
...
/* object behavior */
if window.location.top_left.row < 0 then window.location.top_left.row = 0.
...
window.location.bottom_right.row = window.location.top_left.row - window.size.height.
```

Note that this model makes no distinction between an object and its class. If window has yellow color and square shape, and window-1 is a window, then window-1 also has yellow color and square shape.

An Example

Suppose we wanted to examine the one-dimensional heat conduction along a rod, as discussed earlier. The object system (itself an object) is the rod. The description of the rod would contain stability rules and facts which define the heat conduction along the rod among the rod_nodes. In this example, the stability rules would be comprised of the explicit solver, which is used to determine the state of the internal rod_nodes. We augment the definition of node with the rule,

```
If node.type = Eulerian
Then node.temp @(time=t+1 and location=i)=
        node.temp@(time=t and location=i) +
        lambda*(node.temp@(time=t and location=i-1)
                - 2*node.temp@(time=t and location=i)
                + node.temp@(time=t and location=i+1))
```

The scientist initializes the system by specifying the initial and boundary conditions, by asserting values for temperature and location attributes of each rod_node. So, if the external boundary conditions were temp=100°C and temp=50°C at the left end and right end of the rod, respectively, and the internal nodes were set to some ambient temperature, say temp=0°C, then following facts would represent the state of the rod,

```
node.temp@(location=0)=100
node.temp@(location=rod.length)=50
node.temp@(location!=0 and location!=rod.length and time=0)=0
```

When the scientist queries the system for the state of a rod_node at a new time, this places the object system in an unstable state. For example,

```
node.temp@(location=3 and time=5)=?
```

The object system then applies its stability rules to adjust each rod_node until the object system is again stable, and returns the answer to the scientist's query. Alternately, the scientist can query for the time at which a rod_node has a specific temperature, which again causes the object system to fire its stability rules. For example,

```
node.temp@(location=3 and time=?)=70.
```

Conclusions

This paper introduces two separate OOP paradigms for addressing scientific computations, one dealing with the language C++ and another called the "alternative" paradigm. The two paradigms are distinguished in terms of their design for OOP for the following items. The paper gives an example of programming with the alternative paradigm, and provides technical models using each of the paradigms.

In the conventional C++ approach, knowledge about the structure of an object is separated from the knowledge of its behavior; message passing specifies the behavior. A procedural language is used to explicitly code the physical laws. Each user code, although modular, has a specific structure associated with a specific physical application. Any changes to one area of the knowledge base can have an impact on other knowledge in the database.

For the alternative OOP paradigm, knowledge of the behavior of an object and knowledge of the structure of the object are both attributes of the object. A declarative language codes the knowledge of the physical laws as facts and rules so they don't need to be translated into procedures, as is the case using a procedural language. The rules and physical laws are stored in the knowledge base and various applications are implemented as specific questions about the "state" of the object. State changes are handled by rules that specify how to transition from the unstable state back to a stable state. Changes to one area of the knowledge base have few or no effects on the rest of the knowledge base.

Although much remains to be done to make OOP a realistic and useful computational tool for scientists and engineers in their attempts to model the physical world, the few works developed in this field have shown tremendous potential.

References

Angus, I. G. and Thompkins, W. T. (1989) "Data Storage, Concurrency, and Portability: An Object Oriented Approach to Fluid Mechanics", 4th Conference on Hypercubes, Concurrent Computing, and Applications, Monterey, CA.

Filho, J. S. R. A. and Devloo, P. R. B. (1991), "Object Oriented Programming in Scientific Computations: The Beginning of a New Era", Engineering Computations, Vol. 8, pp. 81-87.

Forslund, D., Wingate, C., Ford, P., Junkins, S., Jackson, J. and Pope, S. (1990), "Experiences in Writing a Distributed Particle Simulation Code in C++", Proceedings of the 1990 Usenix C++ Conference, Usenix Association, pp. 117-190, Berkeley, CA.

Langefors, B. (1980). " Infological Models and Information User Views", Information Systems, 5, pp. 17-32.

Peskin, R. L. and Russo, M. F. (1988), "An Object-Oriented System Environment for Partial Differential Equation Solution", Proc. ASME Computations in Engineering, pp. 409-415.

Ross, T., Wagner, L. and Luger, G. (1991), "Object Oriented Programming for Scientific Codes: Practical Examples in C++", in review.

Wand, Y. and Weber, R. (1989), "An Ontological Evaluation of Systems Analysis and Design Methods", Information Systems Concepts: An In-depth Analysis, Falkenber, E. and Lindgreen, P. (eds), Elsevier Science, pp. 79-107.