A Framework-Based Environment for Object-Oriented Scientific Codes

ROBERT A. BALLANCE[1,2], ANTHONY J. GIANCOLA[1], GEORGE F. LUGER[2], AND TIMOTHY J. ROSS[3]

[1]*Kachina Technologies, Inc., Albuquerque, NM 87110*
[2]*Computer Science Dept., The University of New Mexico, Albuquerque, NM 87131*
[3]*Civil Engineering Dept., The University of New Mexico, Albuquerque, NM 87131*

ABSTRACT

Frameworks are reusable object-oriented designs for domain-specific programs. In our estimation, frameworks are the key to productivity and reuse. However, frameworks require increased support from the programming environment. A framework-based environment must include design aides and project browsers that can mediate between the user and the framework. A framework-based approach also places new requirements on conventional tools such as compilers. This article explores the impact of object-oriented frameworks upon a programming environment, in the context of object-oriented finite element and finite difference codes. The role of tools such as design aides and project browsers is discussed, and the impact of a framework-based approach upon compilers is examined. Examples are drawn from our prototype C++-based environment. © 1994 by John Wiley & Sons, Inc.

1 FRAMEWORKS

Object-oriented scientific programming aims to harness the power of object-oriented design and representation to the task of scientific computing. The goals of our work are:

1. To provide useful computational tools to scientists and engineers so that they need not become programmers.
2. To enhance user productivity via component and design reuse.
3. To support a spectrum of computer architectures, including sequential, vector, and massively parallel processors.
4. To reclaim any computational costs introduced while satisfying the first three goals by developing and applying new translation technology to the resulting programs.

Reusable designs, implemented as object-oriented frameworks [1–3], are key to object-oriented scientific programming. A framework describes the basic elements used to create a general solution. But a framework is not just a collection of design guidelines or libraries; it is an integrated collection of components and interfaces that is designed to be easily extended into a working code. By choosing among various components, and by tailoring components to the exact problem at hand, a user can adapt a framework to yield the desired computation.

Framework-based programming is a logical extension of object-oriented programming. Quite of-

Received April 1993
Revised June 1993

Work supported in part by the Air Force Phillips Laboratory under Contract F29601-91-C-0074 and by the University of New Mexico College of Engineering.

Scientific Programming, Vol. 2, pp. 111–121 (1993)
CCC 1058-9244/94/040111-11

ten, a developer needs only to tailor an existing component by adding new functionality, or by modifying existing behavior. Object-oriented languages support this activity by providing ways to incrementally layer new behavior on to existing components. Adding the new behavior derives a new kind of component while the existing components are not changed.

The end user of an object-oriented framework is principally involved in specializing existing class elements to provide new or refined behavior. Thus, the dominant use of inheritance in a framework is to support specialization of classes. This usage leads to widespread use of inheritance, "fat" interfaces [4] and dynamic function dispatches.

In providing design reusability, frameworks offer

1. A design structure for developing reusable components
2. A unit of transportability among diverse computer architectures
3. A semantic structure for developing framework-cognizant tools.

1.1 Reusability of Components

Designing reusable components is difficult. First, a reusable component is necessarily general. When an already developed module is to be made reusable, the designer's hardest task is to decide how the module should be generalized. Second, components are never used in isolation, but are combined with other components. Without considering the desired forms of interaction, a reusable component might not be, in fact, usable when combined with other components. A design framework provides the guidelines needed to solve these problems. Because the framework itself is general, the framework implicitly determines those ways in which a component must be general. Because the framework specifies the necessary interactions among its components, the framework explicitly determines the various interactions among components that must be supported.

Adoption of a common set of frameworks also creates the opportunity to exchange components of codes, as in the object-component industry now arising in general programming. Because the components will work with the common framework, they will automatically work together. These features make it possible in the future to develop new codes rapidly, to modify existing codes readily, and to reuse codes in a matter of hours or days.

1.2 Transportability

Because frameworks provide domain-specific abstractions, frameworks provide a natural structure for moving codes among architectures. A framework should be architecture independent, but may be reimplemented using different library components for different architectures. In effect, the framework stays the same while implementation details differ.

Four levels of transportability are evident in a framework-based environment. Language translators provide the first level; by compiling and optimizing for different architectures, programs can occasionally be ported without change. Component definitions provide the second level of transportability: different implementations of matrices, for example, might be targeted for different architectures. The supporting code in the framework provides a third level. At this level, the choices made by a designer will still hold, but an alternative implementation of the framework itself may be useful. At the fourth level, the user may choose to revise prior implementation choices to derive a new program from the framework. For example the user might choose to move from an implicit to an explicit solution method using framework-cognizant tools to replay and revise an earlier design.

1.3 Framework-cognizant Tools

Frameworks provide a meaningful and useful structure for developing support tools. Experience with language-based "intelligent" program editors, such as the *Pan* system [5], indicates that attempting to provide extensive support at the level of a programming language is probably inappropriate; even the forms of global information available to a user are limited to artifacts of programming [6]. For example, a language-based environment can often provide an answer to the question, What functions call function A? Without design-specific knowledge, however, it cannot answer the question, Why is function A called? Frameworks, with distinct operational components and definable interactions, provide a higher-level "pattern language" to which a useful design semantics can be applied. The connections among components can be annotated with semantic information, and the choices made by a user can be traced and gathered into a design rational. It is at the framework or design level that intelligent tools are integrated.

2 EXAMPLE: THE COMPUTATION-SPACE-QUANTUM-CONTROLLER FRAMEWORK

A framework provides a way to reuse designs: in our case methods for solving scientific problems [7–9]. In this section, we provide a brief introduction to the Computation-Space-Quantum-Controller framework for finite element and finite difference codes.

Figure 1 shows a high-level overview of the framework. As shown, each numeric code consists of one or more Computations. Each Computation provides the data and the solution for a single problem. Within a computation are a Controller and a Computational Space. The Computational Space (or Space) is where the actual computation takes place; the Controller is an object that starts (and continues) the computation by sending messages to the Space. The effect is to move the outer loop of a numeric computation into the Controller. Because Spaces contain the data being manipulated, they are responsible for managing their own input and output.

Within Computations, there are several interlocking and mutually dependent classes: LoopControl, Controller, ObjectHavingState, Space, and Quantum. Each of these is an abstract base class. Figure 2 illustrates the derivation hierarchy for these core classes. Multiple inheritance is used to enforce two restrictions: that a Space support the same control interface as a LoopControl, and that a Space and a Quantum share the properties common to ObjectHavingState. A Space can also contain objects derived from ObjectHavingState. Thus, Spaces can contain embedded subspaces as well as Quanta.

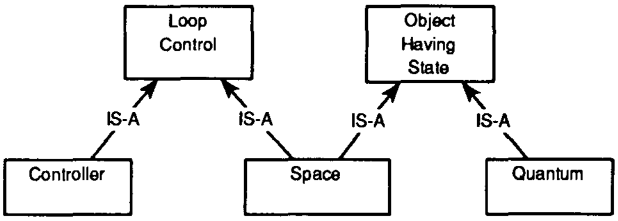


FIGURE 2 Derivation hierarchy for core classes.

Each of the core classes plays a single role in the code: a LoopControl provides the switches and dials for controlling iteration and checkpointing. ObjectHavingState is the basic container for data and scientific behavior. Class Quantum is primarily a computational element. The Space is an abstraction that provides both data and control. Finally, the Computation class provides the stage for the other players.

The ownership relationships (HAS) among the core classes are shown in Figure 3. Comparison between Figures 2 and 3 shows that a space both IS-A and HAS an ObjectHavingState. The ability of a Space to contain objects that are also Spaces provides much of the flexibility in the framework. Spaces may organize their subcomponents by

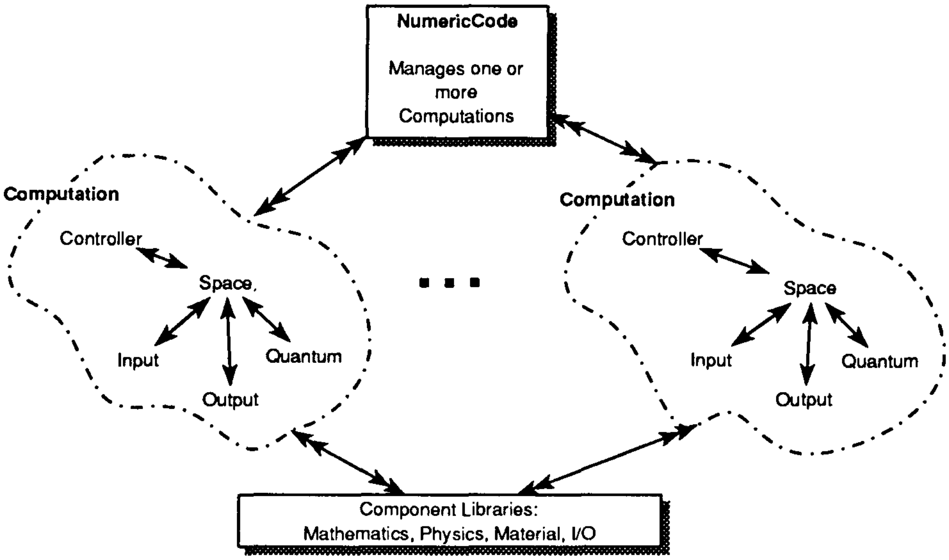


FIGURE 1 Overview of framework.

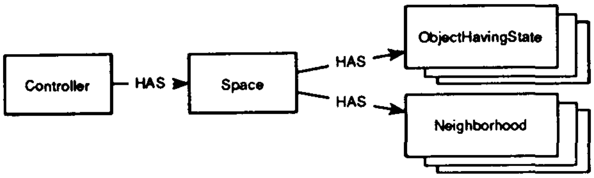


FIGURE 3 Key ownership relations.

means of an abstract supporting class called a Neighborhood. Conceptually, the Space manipulates many instances of objects derived from ObjectHavingState, and many instances of Neighborhoods. In practice, a Space may be able to minimize the actual number of instances by representing them implicitly, or by carefully managing a few prototypical instances. An example of the latter case is when a variant of a Neighborhood is used to represent an element in a finite element code; in many cases it is more efficient to swap Quanta representing nodes in and out of a single prototypical Element than to create one instance of an Element for each element appearing in the model.

Why the distinction between Space and Quantum? In our model, a Space glues a number of ObjectHavingState objects (typically Quanta) together. Within the Space, the Quanta may be represented explicitly as objects; implicitly by storing only their data fields; or as virtual Quanta in which the internal representation may not be visible, but the Quanta appear to be explicitly represented. There is only a tenuous connection between explicitly represented Quanta and explicit codes; the framework allows an explicit code to be written using either implicit or virtual Quanta, but the use of explicit Quanta generally indicates the use of an explicit solution method.

3 SUPPORTING FRAMEWORKS IN A PROGRAMMING ENVIRONMENT

Effective use of design frameworks and object-oriented development requires an innovative development environment. The tools in the environment must understand and respond to the underlying framework and programming techniques, as well as the programming language itself. Such tools include design aides that assist a user to elaborate a framework into a working code, browsers that allow a user to manage libraries of components, and compilers that directly support the framework-based approach.

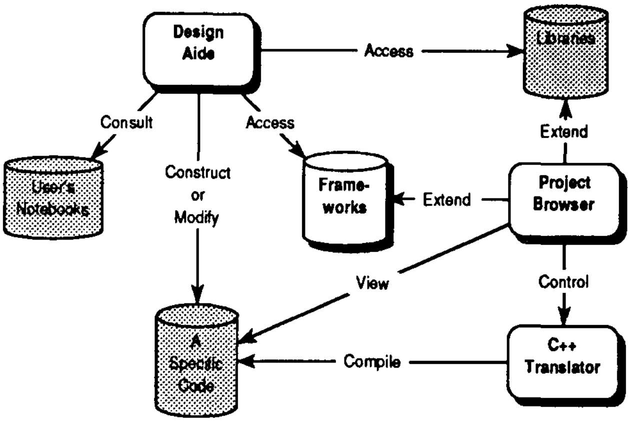


FIGURE 4 An environment for framework-based programming.

Figure 4 illustrates the overall architecture of a framework-based environment. As expected, the frameworks themselves play a central role. The essential tools include a design aide that mediates between users and the framework, a project browser that simplifies access to the code-level aspects of an application, and a translator (or compiler) that is tuned for compiling the resulting programs. The data managed by the environment include the frameworks themselves, the libraries of components, the configured programs, and records of the choices made by a user in elaborating her programs. The latter is denoted by the "User's Notebooks" in Figure 4. The next sections discuss the role of the design aide and the translator in more detail.

Although our experience to date is limited, we remain convinced that frameworks also provide a structure for building truly knowledgeable tools. Prior experience with language-based tools [5, 6] has shown that the programming language level is too low to provide the kinds of assistance an end user can best use. Because a framework provides a problem-solving approach that is domain specific, it provides a natural structure for supporting high level assistance. For example, although a framework may support arbitrary combinations of its elements, a framework-cognizant design tool can provide many techniques for guiding the user's choice of components.

4 DESIGN AIDES AND PROJECT BROWSERS

A design aide is a sophisticated mechanism that helps the user elaborate a framework into a complete program. The design aide uses information accompanying the framework, object, and class modules to assist the user. For instance, a particular finite element code may make commitments on boundary conditions. The design aide can recognize this, using information about the methods and class specifications of the objects and classes the user has chosen. This will assist not only the sophisticated user in designing complex codes quickly, but also remind the novice of the important parameters of the code building process. In a simple sense the design aide can be seen as a novice tutor, assisting the beginning numeric programmer in understanding the important aspects of the code being developed.

The design aide also assists the user by maintaining notebooks of designs for scientific codes. By replaying a design from the notebook, a user can reconstruct (or modify) a previously implemented code. Finally, an intelligent design aide, one augmented with rule-based expertise, can help to guide a user in making her choices.

A screen dump of a prototype design aide for the Computation-Space-Quantum-Controller framework is shown in Figure 5. This prototype allows a user to select and modify components, either through the interface or through a combination of the design aide and direct browsing. In the illustration, our user has selected several components, indicated by the filled-in selection wells. At this point, the Quantum component has yet to be elaborated.

In Figure 6, the user has added instance variables and equations to the Quantum. The illustration shows both the browser for equations and fields, and the inspector being used to add a heat transfer equation. In this case, the user has chosen to implement her equations directly in C++, augmented by higher level operations supported by the design aide. The notation "N.sum(temperature)" in the equation will be translated into a special C++ iterator function that sums the values of the "temperature" instance variable over all of the neighbors of the Quantum.

Project browsers help users to inspect the objects, classes, and inheritance relations in the environment. Unlike design aides, which are oriented toward nonprogrammers, project browsers allow programmers to view, extend, and modify frameworks, libraries, and codes at the source code level. Additionally, a project browser can exploit annotations present in frameworks to provide enhanced functionality.

A key feature of a project browser is its ability to provide integrated configuration management. By collecting information about the entire program into a program repository, the environment can better support both incremental compilation and interprocedural optimization.

5 LANGUAGE AND TRANSLATOR

The underlying object-oriented programming language has several effects on the environment. First, it determines just what techniques can be used. For example, C++ supports multiple inheritance, making it easier to describe and implement the Computation-Space-Quantum-Controller framework presented above. Second, the language determines the degree of optimization that an environment can provide. Third, the size and complexity of the language have many implications for the ease with which an environment can be constructed, extended, and modified.

The presence of a translator that is fully integrated into an environment allows other tools to use translator facilities without replicating code.

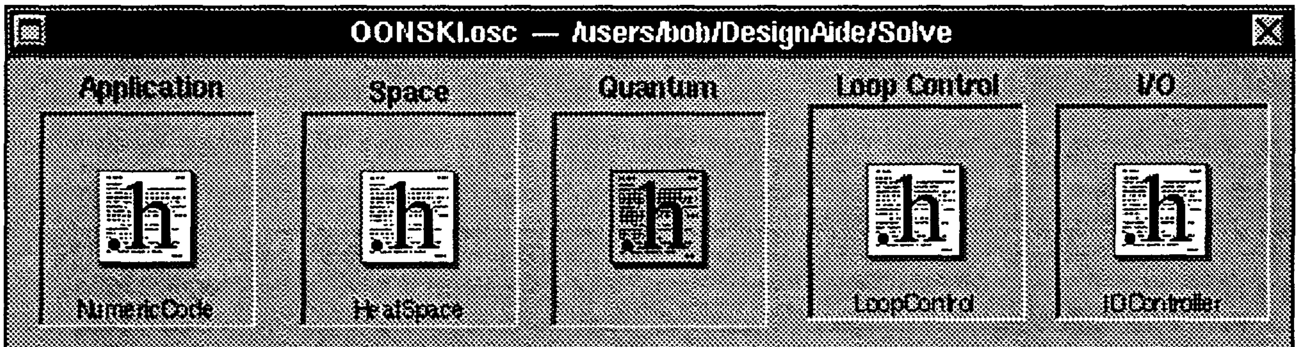
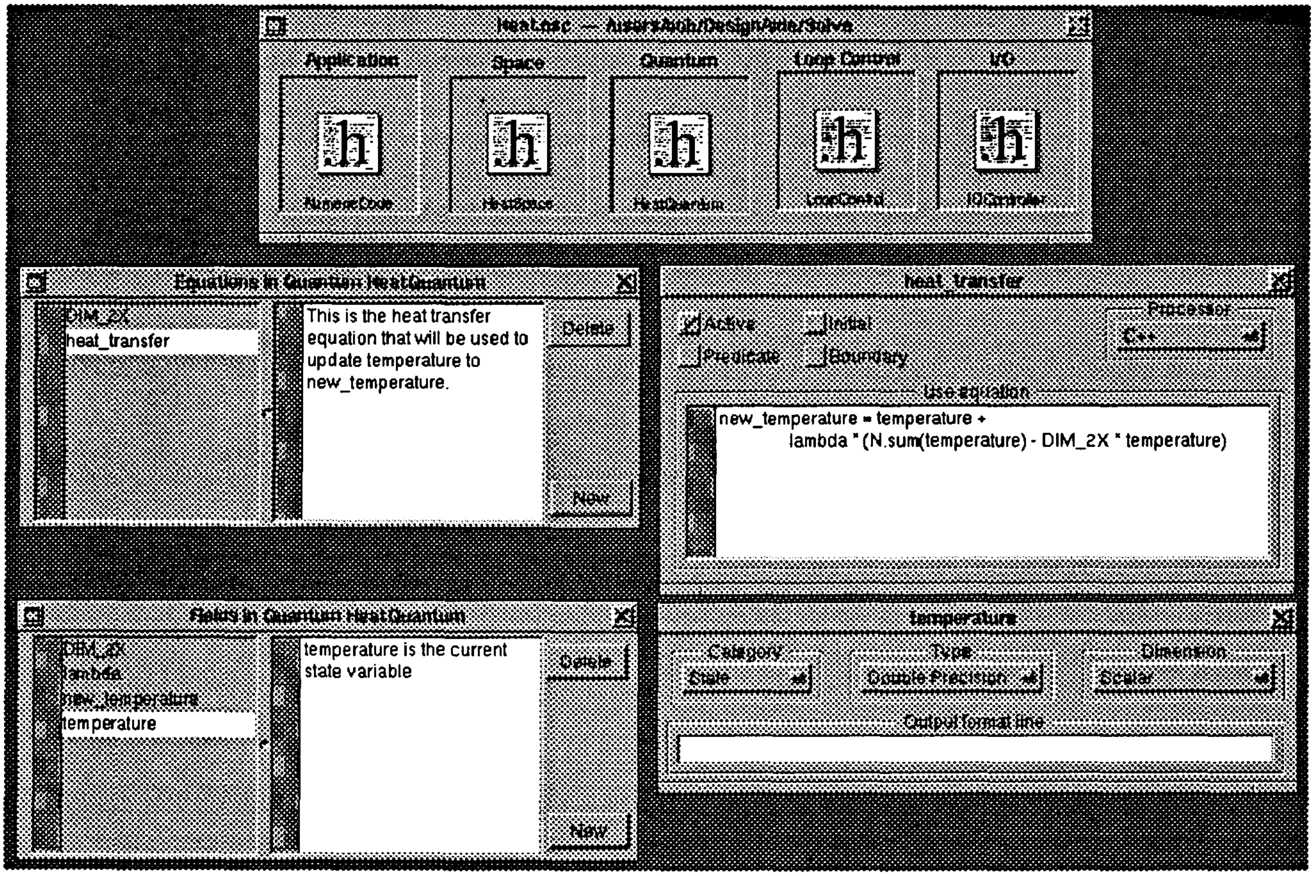


FIGURE 5 Using the design aide to specialize a computation.



DIM_2X
heat_transfer

This is the heat transfer equation that will be used to update temperature to new_temperature.

☑ Active ☐ Initial
☐ Predicate ☐ Boundary

```
new_temperature = temperature +
        lambda * (N.sum(temperature) - DIM_2X * temperature)
```

DIM_2X
lambda
new_temperature
temperature

temperature is the current state variable

FIGURE 6 Elaborating a Quantum component.

Reuse of translator components occurs at two levels: by embedding calls to the translator itself, and by linking directly to phases and other facets of the translator. For example, the design aide must avail itself of the analysis phases of the translator in order to better support the user. These phases, and their results, must be sharable within the environment. Similarly, the class and project browsers require information derivable from the source modules.

5.1 Language: C++

C++ is an evolving object-oriented language based on C [10]. C++ differs from C by adding classes, inheritance, user-defined overloaded operators, dynamic function binding, reference variables, and run-time exceptions. Optimizing (and vectorizing) C++ requires a combination of techniques drawn from both conventional high-quality C compilers as well as from optimizers for object-oriented languages [11]. As with any new technology, existing techniques can be extended to handle new situations. However, besides extending the known to handle the novel, new techniques are emerging. In particular, techniques now being developed in program annotation and partial evaluation show promise of more effective optimization strategies.

Table 1 briefly enumerates some of the features of C++ and their impact on compiling frameworks. In the table, a "+" denotes a feature helpful to an optimizer and a "−" denotes a feature detrimental to optimization.

Locality of reference assists an optimizing compiler, because locality makes it more likely for a compiler to be able to determine the effect of operations. Classes help the optimizer because they support fine-grained encapsulation. On the other hand, classes can complicate the work because they tend to proliferate scopes. Similarly, member functions operate in encapsulated spaces, but tend to be numerous and small. Although small functions are "good" programming practice, they present problems to optimizers which want larger sections of program text to work on. One advan-

Table 1. Impact of C+ + Features on Optimization

C++ Facility	Impact on Optimization
Classes	± Proliferation of scopes
Exceptions	− Inhibits/complicates optimization
Member functions	± Proliferation of functions
Overloaded operators	− Management of temporaries
References	+ Well-behaved pointers
Static members	+ Localization of global variables
Templates	± Proliferation of code
Virtual functions	− Dynamic dispatch

tage of procedure integration (or "inlining") is that it exposes a larger range of code for an optimizer to work on. Function calls also inhibit optimizations.

Templates have much the same effect as classes, but have the additional benefit that different template instantiations can be optimized in different ways. Consider a matrix template that can be instantiated using integers or doubles as the elements in the matrix; the compiler will be able to generate better code because the integer and the double versions are distinct.

References and static members tend to help the optimizer by allowing a programmer to better control encapsulation and to better indicate the use of pointers. This contrasts well with C, in which pointers are used to implement both dynamic data structures and to effect call-by-reference.

Exceptions cause two difficulties. First, they may incur either storage or execution time overhead when a try-block (the block that specifies a possible handler) is entered. Second, because an exception may cause control to leave a block immediately after a function is called, the optimizer cannot assume that control will continue normally after a call. In particular, the optimizer cannot leave values in temporary memory (registers, in particular) unless it is prepared to restore those values when an exception is thrown. Fortunately, techniques exist to handle this problem.

Finally, overloaded operators and virtual function dispatch make good coding and optimization more difficult. Overloaded operators are problematic because they give the programmer the appearance of being primitive operations without giving the compiler sufficient information to manage their resources properly. Virtual function calls are problematic because they complicate the call graph and can hide possible optimization. Methods for dealing with both overloading and virtual calls are discussed in the next section.

5.2 Optimizing Object-Oriented Scientific Programs

A translator in a framework-based, object-oriented, scientific code environment must provide a wide range of optimization techniques to assure that the final programs achieve necessary performance standards. The needed optimizations include both the usual intraprocedural and interprocedural techniques as well as new techniques specific to object-oriented programs.

Especially for framework-based scientific codes, interprocedural analysis is essential. To be completely effective, interprocedural optimization demands full knowledge of the entire program, not just a function, a file, or a class. On the surface, this is contrary to the notion of object-oriented programming as iterative enhancement, in which encapsulation is used to hide the bulk of the details of the program from the programmer. However, it is the translator that is violating encapsulation boundaries, not the programmer. Tools such as project browsers serve as useful intermediaries between the user and the translator by assisting the user in designating the actual configuration of the program and by transferring that information to the translator. The end result is that the user may not even be aware that she is providing the compiler with the information needed to complete interprocedural analysis.

5.2.1 Intraprocedural Optimization

All of the usual intraprocedural optimizations, such as strength reduction, constant folding, code motion, copy propagation, common subexpression elimination, and subscript analysis for uncovering potential vectorization [12] are needed for object-oriented codes. These techniques are not specific to object oriented codes, and are presented in most textbooks that cover optimization of imperative languages [13].

In C++, the ability to redefine operators encourages developers to create new concrete data types. A concrete data type appears to be a primitive type: it can be declared, assigned, and passed as an argument just like a primitive type [14]. For example, a C++ programmer is free to define and use infix expressions such as

A = B + (A * C);

where A and B represent vectors, C is a matrix, and "+" and "*" have user-defined meanings.

As presently defined, C++ does not provide sufficient information about user-defined operators to the translator to effect even common code improvements. Without extralinguistic information, a translator may not be able to bring its full range of techniques to bear on the overloaded operators.

To a programmer, the appearance of the user-defined operators suggests that they will work just like the standard operators. These appearances are deceiving, but nowhere more than in the arena of resource management. It is well known that the management of temporary objects is a difficult problem in C++ class libraries [15, 16].

One method for assisting the translator is to provide annotations on class and method definitions. Annotations, in the form of compiler directives, are already used in many C and C++ translators. Properly expressed, annotations can provide control over optimization, can convey semantic information to the compiler, can be portable, and do not require language extension.

One approach to annotations uses algebraic equations together with cost estimates to express potential transformations. Although not yet fully implemented, our goal is to use the equations as rewrite rules in the same way that other rewrite rules are applied in advanced compilers [17].

Consider again the matrix expression

A = B + (A * C);

If the compiler can determine that A, B, and C do not share storage, this code can be rewritten as

T1 = A * C
A = B + T1

Suppose that both a *= operator and a += operator are defined. Given the rewrite rules $A + B \equiv B + A$, $A += B \equiv A = A + B$, and $A *= B \equiv A = A * B$, an optimizer can easily rewrite this code to

A *= C
A += B

for a savings of at least two function calls in C++, and possibly saving the creation and destruction of two temporary values along with copying. Adding such rewrite rules requires two mechanisms in a compiler: the ability to attach annotations to symbols or other language elements, and the ability to perform rule-directed rewrites. The trick, then is to be able to detect when A, B, and C do not share storage. This requires some form of interprocedural alias analysis, along with the information about storage management.

5.2.2 Interprocedural Analysis

The interprocedural analysis and optimization techniques used for imperative languages, such as Fortran [18], are also needed for statistically typed object-oriented languages like C++. Such techniques include interprocedural alias analysis [19, 20], constant propagation [21], data flow analysis [22, 23] and control dependence analysis [22, 24, 25] for determining potential parallelization strategies.

As an example, consider a Vector class having operations * and +, and consider the Vector expression

A = B * C + D

The simplest translation of this code from C++ into C results in several nested function calls:

```
Vector::operator=(A,
   Vector::operator+(
      Vector::operator*(B, C), D))
```

By linearizing and naming temporaries, a compiler can easily achieve

Vector T1, T2
T1 = B * C
T2 = T1 + D
A = T2

Given a reasonable procedure integration (inlining) mechanism, the code can now be expanded to something resembling

```
// set up *
for (i = base1; i < size1; i++)
{ t1[i] = b[i] * c[i];}
// set up +
for (j = base2; j < size2; j++)
{ t2[j] = t1[j] + d[j];}
// set up copy
for (k = base3; k < size3; k++)
{ a[k] = t2[k];}
```

With interprocedural constant propagation, it may be possible to determine that *base*1 = *base*2 = *base*3 and *size*1 = *size*2 = *size*3. In this case, the loops can be jammed:

```
for (i = base1; i < size1; i++) {
    t1[i] = b[i] * c[i];
    t2[i] = t1[i] + d[i];
    a[i] = t2[i];
}
```

Straightforward transformation then yields a nicely vectorizable loop using vector chaining. Again, this depends upon the presence of adequate analysis: interprocedural constant propagation to allow the loop jamming,* alias analysis along with subscript analysis in the target compiler to detect that the loop is vectorizable.

5.3 Function Specialization

Framework-based programs make heavy use of virtual (dynamic) function dispatch. It is a virtue of object-oriented languages that any specific object A can be used wherever a more general object B can be used, so long as A is derived from B. This virtue, of course, has a cost in the form of virtual functions. Virtual functions have two costs: they inhibit optimization and they incur cycles during execution.

In a framework, most of the apparent classes are abstract; they will be replaced by specific classes and objects quite uniformly in the resulting code. For example, although a component may be defined in terms of a matrix object, the final code might use only one specific class of matrices. In this case, the virtual function calls could be eliminated in the optimized code in favor of direct calls to the class being used.

Function call specialization [26, 27] is the elimination of run-time procedure dispatch by determining at compile time the actual function being invoked. Specialization requires interprocedural type propagation, along with the ability to examine the entire program being compiled.

For example, consider a C++ fragment in which the doSomething member function is invoked on anObject.

```
anObject.doSomething()
```

The actual function invoked depends on the type of anObject: it may be inherited or it may be a member function defined in some class derived from anObject's class. In the presence of a virtual function, a basic translator will generate a dynamic procedure call that consults the table of functions associated with anObject's class during execution.

However, whenever the actual class of the recipient (anObject in this case) can be determined at compile time, the dynamic call can be replaced by a static call. Better, by determining the exact definition of doSomething, the translator can integrate the body of doSomething directly into the loop. Procedure integration may create new opportunities for improvement. Integrated configuration management also supports this optimization by providing access to all of the sources needed for procedure integration, whether or not a programmer has specified the functions to be "inline" using the nominal C++ directive.

Many C++ translators already generate direct calls to virtual functions provided that the type of the recipient can be derived locally. However, interprocedural type propagation will enable a compiler to fully deploy this optimization. In its simplest form, type propagation is just a form of constant propagation on type values.

Specialization does not require that only a single function be invoked. For example, consider a code fragment in which an iterator over a list of Shapes invokes member functions on each individual shape.

```
for (p = firstShape; p != 0;
p = p->nextInList()) {
    p->resize(...);
    p->move(...);
    p->redraw(...);
}
```

Each iteration involves three dynamic function dispatches. Now suppose that a variable p can

* Full interprocedural constant propagation is not essential to this example; a compiler can generate multiple versions of the overall procedure by dynamically testing the sizes, and if they are all equal, executing the jammed and optimized loop.

refer to any of several different classes derived from Shape. If the derived classes are known at compile time (or even if only a subset is known), a compiler can factor out the dynamic dispatch and then use statically compiled calls:

```
for (p = firstShape; p!= 0;
    p = p->nextInList()) {
  if( isASquare(p)) {
  // Call Square functions
       Square::resize(p, ...);
       Square::move(p, ...);
       Square::redraw(p, ...);
  } else if (isACircle(p)){

       ...

  }
```

The details of run-time type determination are omitted. In this case, the tests isA... can be generated by the compiler. The proposed run-time type identification facility being considered by the C++ standards committee also is sufficient [28].

Performing specialization effectively conflicts with the separate compilation model embraced by C and C++. Ideally, an optimizing compiler for C++ will have access to all of the source code for an application. Full access to the source code does not, however, compromise the object-oriented programming model. When full access is unavailable, the compiler must fall back to conservative assumptions and produce correct output.

5.4 Partial Evaluation

Partial evaluation results from combining a (possibly empty) subset of a program's data with a program to produce a new, simpler, program [29]. In a sense, conventional optimizations such as constant folding are just forms of partial evaluation using an empty input data set.

Berlin [30, 31] has shown that partially evaluated scientific codes can show significant speedups. Although it is unclear how to extrapolate these results from small Scheme programs to large scientific codes, the results are encouraging.

Koo and Sundaresh [32] have recently shown that partial evaluation can be used to implement function call specialization. Their work is based on a high-level semantic model, but the results confirm the following observation: by using partial evaluation and by tracking the types of the objects created, a system can determine the actual types of objects involved in virtual function calls. Thus, rather than implementing a static analysis to propagate data types, the system simply performs an abstract interpretation on the program and tracks the results. Krishna [33] is currently working on the general problem of partial evaluation in C and C++ for scientific codes.

On its own, using partial evaluation would be too expensive to apply to most programs. However, an optimizer can use results from the partial evaluation to simplify other optimization passes such as interprocedural constant propagation. In the long term, partial evaluation may become an important technique in optimizing large scientific codes because many scientific codes operate on relatively fixed data sets for which a partially evaluated program would be appropriate.

6 CONCLUSION

This article has presented a framework-based environment for object-oriented scientific programming and has examined the impact of a framework-based approach upon programming environments for object-oriented scientific codes. The use of a framework simplifies the creation of domain-specific, intelligent tools that can assist in the elaboration of programs from the framework. These same tools can be used to support project browsing and configuration management. With integrated configuration management, the environment is able to provide the interprocedural analysis needed to fully optimize scientific and numerical programs. Finally, the article has briefly touched upon optimization of object-oriented codes, including function specialization, partial evaluation, and the need for interprocedural analyses.

REFERENCES

[1] F. Dearle, "Designing portable application frameworks for C++," *C++ J.*, vol. 1, pp. 55–59, 1990.

[2] L. P. Deutsch, *Software Reusability: Volume II, Applications and Experience.* New York: ACM Press, 1989, pp. 57–71.

[3] R. Johnson and R. Wirfs-Brock, *OOPSLA Conference on Object-Oriented Programming Systems, Languages, and Applications.* New York: ACM Press, 1991.

[4] B. Stroustrup, *The C++ Programming Language* (2nd ed.). Reading, MA: Addison Wesley, 1991.

[5] R. A. Ballance, S. L. Graham, and M. L. Van De Vanter, "The Pan language-based editing system," *ACM Trans. Software Eng. Methods*, vol. 1, pp. 95–127, 1992.

[6] M. L. Van De Vanter, R. A. Ballance, and S. L. Graham, "Coherent user interfaces for language-based editing systems," *Int. J. Man-Machine Systems*, vol. 37, pp. 431–466, 1992.

[7] T. J. Ross, G. F. Luger, P. Morrow, and L. Wagner, *Proceedings of the ASCE 8th Conference on Computing*. New York: ASCE, 1992, pp. 535–542.

[8] T. J. Ross, G. F. Luger, and L. Wagner, "Object oriented programming for scientific codes: Thoughts and concepts," *ASCE J. Comput. in Civil Engineering*, vol. 6, pp. 480–496, 1992.

[9] T. J. Ross, G. F. Luger, and L. Wagner, "Object oriented programming for scientific codes. II: Examples in C++," *ASCE J. Comput. in Civil Engineering*, vol. 6, pp. 497–514, 1992.

[10] ANSI, *X3J16, Working Paper for Draft Proposed International Standard for Information Systems—Programming Language C++*. Washington, DC: The American National Standards Institute, CBEMA.

[11] C. Chambers, *The Design and Implementation of the SELF Compiler, an Optimizing Compiler for Object-Oriented Programming Languages*. Report Number STAN-CS-92-1420, Department of Computer Science, Stanford University, March 1992.

[12] R. Allen and S. Johnson, *Proceedings of ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*. New York: ACM Press, 1988, pp. 241–249.

[13] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*. Reading, MA: Addison-Wesley Publishing Company, 1986.

[14] J. O. Coplien, *Advanced C++: Programming Styles and Idioms*. Reading, MA: Addison Wesley, 1991.

[15] K. G. Budge, J. S. Perry, and A. C. Robinson, *Workshop Proceedings*. Berkeley, CA: USENIX Assoc., 1992, pp. 121–150.

[16] D. Forslund, et al., C++ Workshop Proceedings. Berkeley, CA: USENIX Association, 1990.

[17] E. Pelegri-Llopart, *Rewrite Systems, Pattern Matching, and Code Generation*. Report Number UCB/CSD 88/423, Computer Science Division, University of California, Berkeley, June 1988.

[18] F. Allen, et al., "An overview of the PTRAN analysis system for multiprocessing," *J. Parallel Distrib. Processing*, vol. 5, pp. 617–640, 1988.

[19] W. Landi and B. Ryder, *Proceedings of the ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation*. New York: ACM Press, 1992.

[20] W. E. Weihl, *Seventh Annual ACM Symposium on Principles of Programming Languages*. New York: ACM Press, 1980, pp. 83–94.

[21] M. N. Wegman and F. K. Zadeck, "Constant propagation with conditional branches," *ACM Trans. Programming Languages Systems*, vol. 13, pp. 181–210, 1991.

[22] R. A. Ballance and A. B. Maccabe, *PDGs for the Rest of Us*. Technical Report 92-10, Department of Computer Science, University of New Mexico, Revised October 1992.

[23] B. G. Ryder and M. C. Paull, "Elimination algorithms for data flow analysis," *ACM Comput. Surv.* vol. 18, pp. 277–316, 1986.

[24] R. A. Ballance, A. B. Maccabe, and K. J. Ottenstein, *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation*. New York: ACM Press, 1990, pp. 257–270.

[25] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The program dependence graph and its use in optimization," *ACM Trans. Programming Languages*, vol. 9, pp. 319–349, 1987.

[26] C. Chambers and D. Ungar, *Proceedings of the ACM SIGPLAN 1989 Conference on Programming Language Design and Implementation*. New York: ACM Press, 1989, pp. 146–160.

[27] D. Lea, *C++ Workshop Proceedings*. Berkeley, CA: USENIX Association, 1990, pp. 301–314.

[28] B. Stroustrup and D. Lenkov, "Runtime type identification for C++," *C++ Report*, vol. 4, pp. 32–42, 1992.

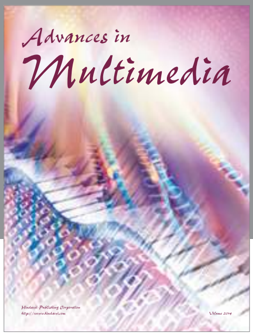
[29] D. A. Bjorner, A. P. Ershov, and N. D. Jones, *Partial Evaluation and Mixed Computation*. Amsterdam: North-Holland, 1988.

[30] A. Berlin, *Proceeding of the 1990 ACM Conference on Lisp and Functional Programming*. Loc: Publ, 1990, pp. 139–160.

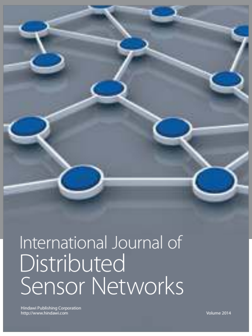
[31] A. Berlin and D. Wiese, "Compiling scientific code using partial evaluation," *Computer*, vol. 23, pp. 25–37, 1990.

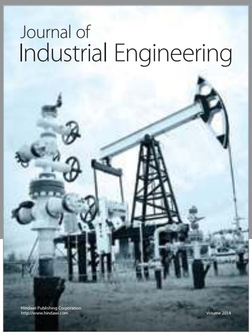
[32] S. C. Koo and R. S. Sundaresh, *Proceedings of the Symposium on Partial Evaluation and Program Manipulation PEPM*. New Haven, CT: Yale University, 1991, pp. 211–222.

[33] K. Krishna, "Program specialization via partial evaluation," Dissertation Proposal, Department of Computer Science, The University of New Mexico, Albuquerque, New Mexico, July 1992.

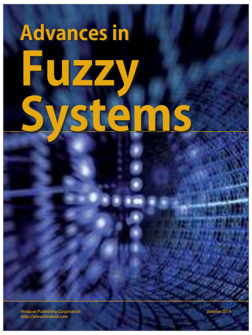

Advances in
Multimedia

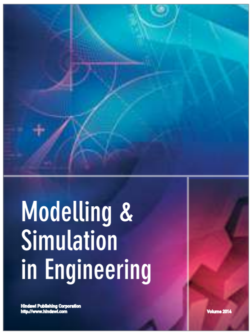
The Scientific
World Journal

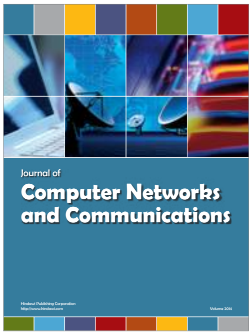

International Journal of
Distributed
Sensor Networks

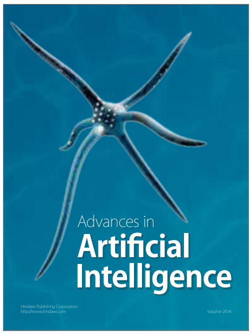

Journal of
Industrial Engineering

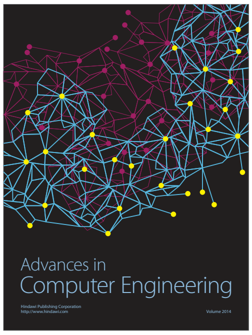
Applied
Computational
Intelligence and Soft
Computing


Advances in
Fuzzy
Systems

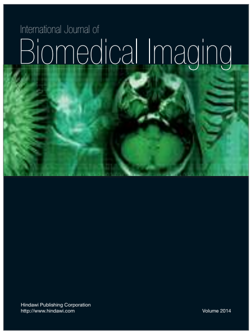

Modelling &
Simulation
in Engineering

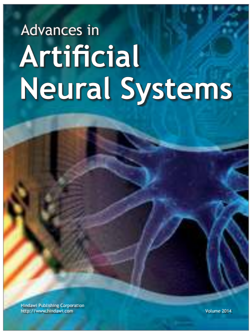

Journal of
Computer Networks
and Communications

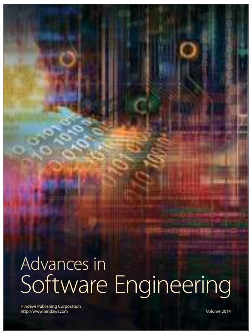

Advances in
Artificial
Intelligence


Advances in
Computer Engineering

International Journal of
Computer Games
Technology

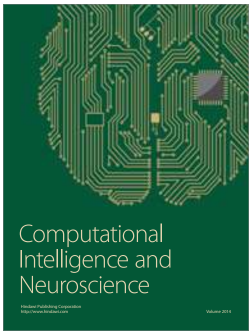

International Journal of
Biomedical Imaging


Advances in
Artificial
Neural Systems


Advances in
Software Engineering

Journal of
Robotics

Advances in
Human-Computer
Interaction


Computational
Intelligence and
Neuroscience

International Journal of
Reconfigurable
Computing

Journal of
Electrical and Computer
Engineering