

Experiences Developing ALEGRA: A C++ Coupled Physics Framework*

Kent G. Budge[†]

James S. Peery[‡]

Abstract

ALEGRA is a coupled physics framework originally written to simulate inertial confinement fusion (ICF) experiments being conducted at the PBFA-II facility at Sandia National Laboratories. It has since grown into a large software development project supporting a number of computational programs at Sandia. As the project has grown, so has the development team, from the original two authors to a group of over fifteen programmers crossing several departments. In addition, ALEGRA now runs on a wide variety of platforms, from large PCs to the ASCI Teraflops massively parallel supercomputer. We discuss the reasons for ALEGRA's success, which include the intelligent use of object-oriented techniques and the choice of C++ as the programming language. We argue that the intelligent use of development tools, such as build tools (e.g. *make*), compiler, debugging environment (e.g. *dbx*), version control system (e.g. *cvs*), and bug management software (e.g. ClearDDTS), is nearly as important as the choice of language and paradigm.

1 Introduction

ALEGRA is a coupled physics framework whose roots go back to 1990, when the authors joined Sandia National Laboratories and began development of a shock physics code based on arbitrary Lagrangian-Eulerian finite element algorithms. This code was intended to support the inertial confinement fusion (ICF) program built around the PBFA-II facility by providing a way to simulate the implosion of ICF target capsules driven by light ion beams. Originally the code was a variant of the PRONTO finite element code [1], which had proven successful for Lagrangian transient dynamics analysis and was felt to provide the necessary framework. PRONTO was written in FORTRAN-77, plus a few C subroutines (mainly for memory management,) and was unusually well-structured and -documented.

As work on the new code (originally named RHALE) progressed, it became increasingly burdensome to add new features to the existing framework. The memory management scheme was difficult to work with and such minor nuisances as the 19-line continuation limit on FORTRAN statements became major stumbling blocks. The lack of type checking in FORTRAN (and our failure to obtain and use tools such as FLINT) was a constant source of deep bugs. We favored

*This work was supported by the United States Department of Energy under Contract DE-AC04-94AL85000. Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy.

[†]Senior Member of the Technical Staff, Sandia National Laboratories, Albuquerque, NM.

[‡]Manager, Computational Physics Research and Development (9231), Sandia National Laboratories, Albuquerque, NM.

switching development to C but were hesitant to suggest the change to our management because of concerns about the learning curve and the cost of rewriting legacy code.

Another growing concern was with the issue of parallelization. Our internal customers in the ICF program wanted 3-D simulations in which hydrodynamics, radiation transport, and ion energy deposition were all coupled. Such simulations would require at least an order of magnitude more memory and processor speed than would be available on any single-processor or shared-memory multiprocessor supercomputer under development at the time. It seemed likely (as was to prove the case) that future Sandia supercomputers would be massively parallel distributed memory machines. Therefore, we were expected to develop a code that could be run on such machines, but we were also expected to “hedge our bets” by ensuring that the code would also run well on vector supercomputers and on shared-memory multiprocessor supercomputers. These are rather incompatible goals, particularly when one considers that vectorization often requires exactly the opposite programming strategy from that required to make effective use of cache on RISC machines. However, we were obligated to do our best.

It was at about this time that C++ swam onto our radar screens. We were briefed on the language and its usefulness for massively parallel computing by Ian Angus, then of Boeing Corporation. Our manager also attended the briefing and was sufficiently impressed to authorize us to experiment with writing a version of RHALE in C++. Naturally, we dubbed this version RHALE++, thus falling into the first common C++ trap: **The decision to use a sexy new programming language is not sufficient to guarantee a sexy new code**, and many a programming team has engaged in dubious advertising by appending a “++” to the name of their product. Had we realized how expensive it would be to convert our programming environment to C++, we doubt our management would have approved the conversion. On the other hand, had we had such miraculous foresight, we would have avoided a number of pitfalls and thereby made the conversion much less expensive.

We over time have also considered using FORTRAN-90 or HPF. We rejected the former because of a discouraging lack of tools (including compilers) and the latter because it was still in an embryonic state. By contrast, C++ was widely available (via cfront) and could be compiled on all platforms of interest. Other development tools, such as integrated design environments with source checking, were becoming available as well.

We found programming in C++ to be a very exciting experience and wrote many enthusiastic articles on our work [2], [3], [4], [5], [6]. We particularly liked the idea of operator overloading on concrete data types, which seemed the natural way to express physics as computer code [7], [8], [9], [10]. Operator overloading is orthogonal to object-oriented programming, but our confusion was understandable given the amount of jargon being thrown around at the time. This leads to the rather obvious conclusion: **Anyone can throw around a buzz word. Some research is needed to actually understand its meaning.** This is not to deny the value of operator overloading on concrete types, a topic we discuss in more detail later, but by mistaking this for object-oriented programming (which we now understand to mean programming with polymorphic classes organized into an inheritance graph) we missed many opportunities for effective programming.

As we continued to develop numerical code in C++, we became aware of many efficiency pitfalls and other difficulties in using the language effectively. This led to considerable research and the identification of many of the bottlenecks [2], [5], [8], [10], [11], [12], [13], [14]. Some of these have been fully resolved; others are still being resolved as compiler technology improves [15].

Prior to 1996, the RHALE++ project, now renamed ALEGRA (on the theory that vaguely feminine Latin names had customer appeal) had grown quite slowly, with the equivalent of perhaps three developers working on the code full time. However, after 1996, the Advanced Strategic Computing Initiative (ASCI) and a number of other programs provided funding to rapidly expand the development effort. The ALEGRA team is now the equivalent of about 10 full-time developers. We soon learned that **using a modern language is not enough. One must also use modern development tools.** In particular, we were compelled to adopt the CVS version

control environment and to institute rigorous regression testing. We are also beginning to make use of the ClearDDTS Web-based bug tracking software.

The ALEGRA code is now approaching maturity, and we have begun to think ahead to the “next” code (or at least to a thorough rewrite of ALEGRA.) We are therefore in a position to pontificate on “lessons learned” and to share our hard-earned experience with other workers in the field.

2 Programming a C++ Multiphysics Framework

Here we describe the C++ programming idioms we consider most valuable for developing multiphysics frameworks. These consist of operator overloading on concrete types; polymorphism by inheritance; and polymorphism by genericity.

2.1 Operator Overloading on Concrete Types

One of the first things that attracted us to C++ was the concept of operator overloading on user-defined concrete types. We were enthralled with the expressiveness of code such as that illustrated in Figure 1:

```
void Decompose(const double delT, SymTensor& V,
               Tensor& R,  const Tensor& L)
{
    SymTensor D;
    AntiTensor W, Omega;
    Vector z, omega;

    D = Sym(L);
    W = Anti(L);

    z = Dual(V*D);
    omega = Dual(W) - 2.0 * Inverse(V - Tr(V) * One) * z;
    Omega = 0.5 * Dual(omega);

    R = Inverse(One - 0.5 * delT * Omega) *
        (One + 0.5 * delT * Omega) * R;
    V += delT * Sym(L * V - V * Omega);
}
```

FIG. 1 *Sample code using operator overloading on concrete types*

which updates the polar decomposition of the deformation tensor in a deformed solid. The equivalent code written in FORTRAN-77 code would be some seven pages in length [5]. This is a dubious statistic, of course: The supporting class definitions come out of a header file some 41 pages in length. Each of the objects declared in this chunk of code corresponds to a familiar mathematical object, such as a Cartesian vector or tensor. Vector and tensor arithmetic is expressed using the appropriate C++ operators and more sophisticated operations (such as taking the symmetric part of a tensor or the dual of a vector) are given their usual names. We thought this was pretty cool stuff, and to an extent we still do.

Unfortunately, most compilers translate this subroutine into sequences of machine instructions that are far from optimal. The biggest difficulty, which we identified in 1994, is that these compilers insist on storing all class objects in addressable memory rather than in registers [12]. This seems like an obvious thing to do, since class objects are a lot like C structs, but when an subexpression returns an object that will be used once, then discarded, it is a lot more efficient to

store the members of the temporary object in registers. Some of the latest C++ compilers perform this “disaggregation of structs” optimization and produce very efficient code for concrete expressions. A measure of this efficiency is that these compilers produce code for the standard `complex` class template that is as efficient as FORTRAN’s built-in `complex` type [15]. Unfortunately, with the scientific market becoming a smaller fraction of the total computing market all the time, support for such specialized optimizations may continue to be limited, although there is some reason to believe they will be important for efficient compilation of the Standard Template Library and therefore of more general interest.

From our perspective as developers of a large physics framework, the use of overloaded operators on concrete types seemed like a natural approach to developing reusable code. The various vector and tensor operations could be coded once and used indefinitely. In fact, our PHYSLIB vector/tensor library [7] has held up remarkably well over the years. We envisioned an environment in which classes would be developed to represent mathematical objects of ever-increasing complexity. Unfortunately, it proved extraordinarily difficult to extend this concept much beyond simple, fixed-size objects such as complex numbers or Cartesian vectors and tensors. The next level of complexity would be scalar, vector, or tensor fields, represented by smart arrays of scalar, vector, or tensor values, with appropriate operations, including calculus operations. Designing beautifully expressive class interfaces for such things was fairly easy. Making them efficient proved to be next to impossible.

The problem is that operator overloads are atomic. If one adds three scalar field objects in a naive way, one ends up generating at least one temporary object that remains alive (on most compilers) until the end of the block in which the statement is located. Intermediate results are once again being stored in memory rather than in registers. Furthermore, individual operations are performed across an entire array, quickly flushing the cache. All these things result in abysmal performance.

We never did satisfactorily resolve this problem, despite considerable effort. It appears that template expressions [16] may have overcome some of these difficulties, but template expression code (like all nontrivial template code) tends to severely tax the capabilities of most of today’s compilers. Our resolution of the problem was to drop the idea of building a hierarchy of increasingly sophisticated concrete classes and to approach reusability in a more traditionally object-oriented way — through the use of polymorphism. We now restrict our use of operator overloading on concrete data types to low-level, fixed-size entities such as the vectors and tensors discussed earlier.

Before we discuss polymorphism, we should describe the current structure of the ALEGRA database. The various scalar, vector, and tensor fields are no longer represented by homogeneous arrays. Instead, the database consists of lists of the various topological entities (nodes, edges, faces, and elements) making up the finite-element grid. Each entity contains its own array of local data. Fast iterators are provided to traverse each kind of list. Simple access functions then provide fast access to the data within each entity. Because the local data array is identical in size for each type of entity, specialized fixed-length allocators are provided that improve allocation time and increase main and cache memory efficiency.

In retrospect, the most fundamental problem with our concept of smart arrays for database storage was that we were trying to reconcile irreconcilable goals. On the one hand, we wanted large array objects that could be processed efficiently by a vector supercomputer. On the other hand, we wanted the objects to work well on massively parallel supercomputers. These two goals are incompatible. Vector supercomputers work best when they chain operations on very long sequences of data. Massively parallel machines, on the other hand, are usually based on RISC processors with hierarchical memories that begin with the register set and proceed through a large cache and main memory to disk storage (via virtual memory.) A distributed-memory massively parallel supercomputer can be thought of as the ultimate hierarchical memory system. Such systems rely heavily on data locality — the opposite of the situation with a vector supercomputer. Put another way, a vector supercomputer is best at applying the same operation to lots of

homogeneous data, while a cached massively parallel machine is at its best applying numerous operations to the same small set of data. Our list-based framework, where each topological entity owns its own local data array, does an excellent job of keeping the data being processed in cache for as long as possible. However, it vectorizes very poorly.

2.2 Polymorphism by Inheritance

Having concluded that operator overloading on concrete data types had its limits, we next turned to the most characteristic technique of object-oriented programming — programming with inheritance graphs and virtual functions. This is what many programmers are thinking of when they discuss polymorphism, though generic programming (templates) is also a form of polymorphism. We were inclined to view inheritance as a high-level mechanism, best applied to very large objects, but prohibitively expensive within innermost loops. This led to a fairly useful paradigm for object-oriented numerics in which a program consisted of a set of large polymorphic objects that communicated with each other and performed internal computations using overloaded concrete objects. We planned to add templates to the mix as containers of concrete objects, once we were confident that compilers were “up to snuff” on template code. (Incidentally, we still haven’t reached that point.) The current version of ALEGRA largely reflects this thinking.

An ALEGRA database consists of a single large object, of class `Body`, that owns an array of objects of class `Region`. The intent was that each region be operated on by a different set of physics packages. However, in the current version of the code, we have never used more than one `Region`, which makes the utility of this feature open to question. class `Region` is abstract. It contains the lists of topological entities together with methods for accessing and manipulating these entities, and provides support for h-adaptivity. `Region` also contains generalized methods for solving certain systems of equations, such as the discretized diffusion equation (useful for heat transport, magnetohydrodynamics, and radiation transport.)

Each class derived from `Region` represents a different physics package. `Region` is always inherited publicly, and as a virtual base class, so that there is no duplication of the actual database. The hierarchy for `Region` looks something like Figure 2. For example, our `RMhdCon` class, which is the ultimate physics package for Sandia’s ICF program, combines all the attributes of the `MhdCon` and `Radiation` classes, which in turn rely on the `Mag`, `Solid_Dynamics`, `Hydrodynamics`, and `Dynamics` classes, all of which are ultimately derived from `Region`. None of these classes add anything to the database in `Region`. They add a few control variables and the methods appropriate for simulating the physics they represent.

Only a few virtual functions of `Region` are called by its client, `Body`. These include initialization methods, a method to advance a step in time, and a method to determine the best time increment for the next time step. Most of the virtual functions in `Region` are used internally, by derived classes. For example, the time step algorithm for most classes derived from `Hydrodynamics` is based on the method, `Hydrodynamics::Lagrangian_Step`. Few classes redefine this method. However, many classes redefine certain virtual functions called within this method, such as `InternalForce`.

Another place in which polymorphism is used is in the element hierarchy. Because the ALEGRA framework is based on finite elements, almost all calculus operations are performed at the element level. ALEGRA supports a variety of elements, including 8-noded hexes, 4-, 8-, and 10-, and 11-noded tets, 6-noded wedges, and 5-noded pyramids. Each element is represented by a class derived from `Element`, which declares a set of virtual methods for the various calculus operations. It is worth noting that this arrangement, which has worked quite well, is in direct violation of our earlier rule that polymorphism and virtual functions are high-level concepts, not to be used in innermost loops. The calculus operations for descendants of `Element` are seldom used anywhere else. We have found that the data localization this provides is much more important than the modest cost of a virtual function `CALL`.

A dangerous but useful practice we have adopted is to allocate all elements as `Element` objects. These are individually passed to the placement new operator, as necessary, to set the

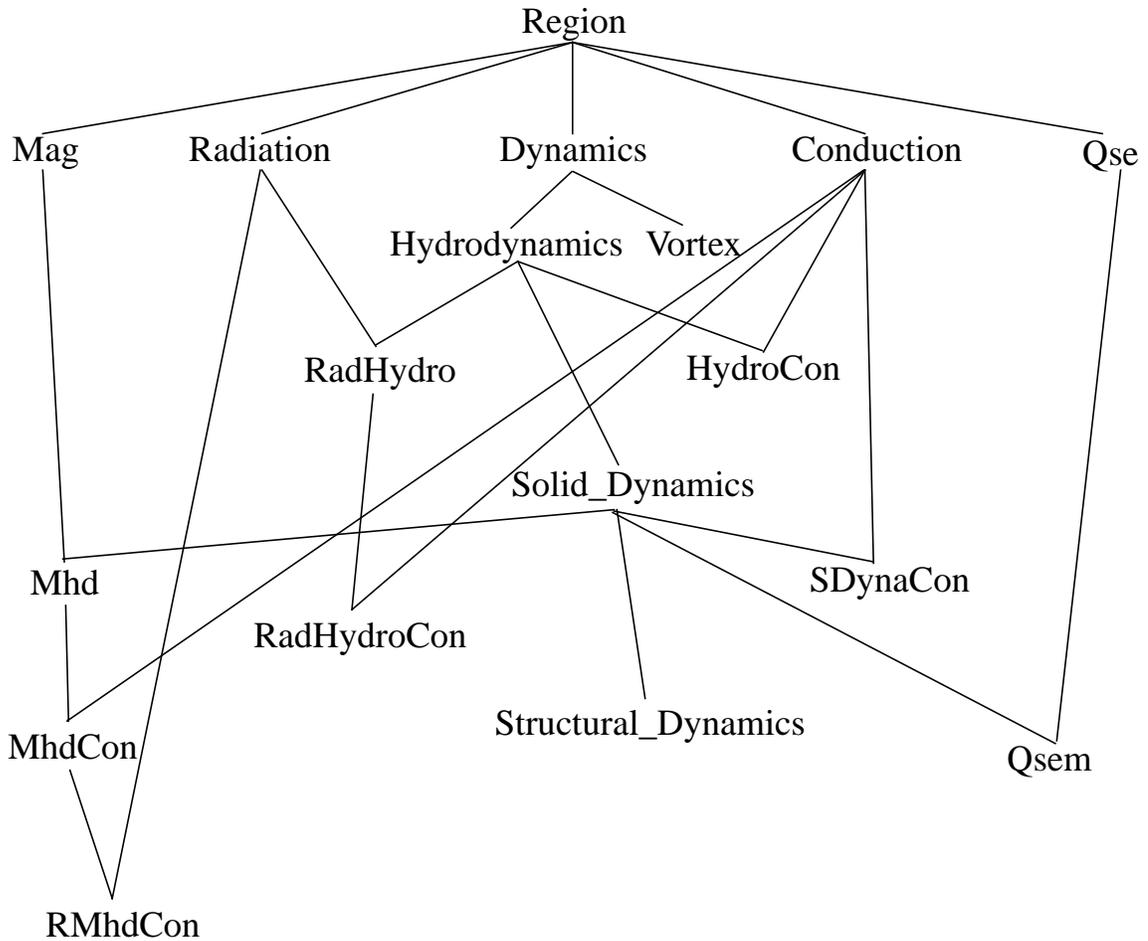


FIG. 2 *Region Class Hierarchy*

virtual function pointer to the correct value for each element. Thus, we have an homogeneous array of inhomogeneous element types. This provides some savings in space and other advantages at the cost of requiring programmers to strictly adhere to a few conventions. Among these are that no class derived from `Element` can add any new data members to the class, nor can they make any changes to the interpretation of existing members that would require redefinition of the destructor. We do not particularly recommend this dangerous practice and will probably avoid it in future code efforts.

Boundary conditions are also implemented using class hierarchies. For example, kinematic boundary conditions used by the `Dynamics` class and its descendants are all represented by descendants of class `Kinematic_BC`, which is an abstract class declaring a number of pure virtual functions that implement the boundary condition. At the appropriate point in the algorithm, class `Dynamics` (or its descendant) loops over the list of kinematic boundary condition objects, calling the virtual `Apply_BC` method for each to enforce the boundary condition.

Finally, material models have been implemented as a hierarchy of classes descended from `Material_Model`. At the appropriate point, each physics algorithm that requires material models to close its equations loops over the list of materials in each element, calling the virtual `Update_State` function to revise the state of the material based on the evolution of the simulation in the previous time step. Furthermore, each material model provides methods to inform the database how much storage is required for material history variables, and the database allocates additional storage for each element accordingly.

Has this resulted in reusable code? Yes and no. We have found ALEGRA a wonderful framework in which to develop and use new physics simulations packages. This has made it easier for us to expand the ALEGRA development team to its current ~15 members, and we have been fairly successful at not stepping on each other's toes. However, the true test of reusability is for a code to be reused outside its original development shop, and we have not seen this happen yet. One reason is that ALEGRA relies on a large suite of supporting codes, including preprocessing codes (mostly mesh generators) and postprocessing codes (for visualization) and a number of software tools. Many of these were developed at Sandia and are not easily available elsewhere. Another reason is the "not-developed-here" mental block that is much too common in the scientific and engineering communities. Finally, C++ is a relatively new language, and many would-be collaborators are deterred by the language barrier, FORTRAN continuing to be the predominant language in the community. However, even if these problems were solved, we suspect that ALEGRA would be difficult to introduce into a new environment, because its class interfaces are not sufficiently transparent to make the framework simple to use without some expert guidance.

2.3 Polymorphism by Genericity

Although many C++ programmers think of inheritance and virtual functions when they hear the word "polymorphism", generic programming (programming with class and function templates) is also a form of polymorphism.

A polymorphic object conforms to the expectation of an algorithm by adapting its type:

```
class Base { /* ... */ };
class Derived : public Base { /* ... */ };

void My_Algorithm(Base *);

/* ... */

Derived *object;
My_Algorithm(object);

// object is a Base so far as My_Algorithm is concerned
```

By contrast, a function template confirms its signature to the type(s) of the object(s) it operates on:

```
template <class T> void My_Algorithm(T *);

Any_Type *object;

My_Algorithm(object);

// My_Algorithm adjusts itself to make use of Any_Type
```

These two mechanisms are therefore complementary.

A function template does have some expectations of the objects it operates on, of course:

```
template<class T> T& max(T &a, T &b) { return a>b? a : b; }
```

In this example, the `max` function template expects that the ordering operator "`>`" is defined for the target type and that this operator returns either a `bool` or a type convertible to `bool`. These are rather loose requirements, making templates exceptionally flexible. We believe that generic

programming will prove to be more flexible than programming using inheritance for many numerical applications.

The existence of function and class templates gives a new significance to operator overloading. In the absence of genericity, operator overloading is cute but nonessential. It provides valuable clarity in writing expressions, assuming it is used intelligently, but one can always define a conventional function instead. In a generic programming environment, the picture changes rather drastically, since operator overloads provide a common spelling for operations on both user-defined and intrinsic types.

We have not made much use of generic programming in ALEGRA; current C++ compilers are just a little too rocky and portability is still a concern. However, we expect this situation to change in time for the next thorough revision of our framework.

2.4 C++ Is the Worst Programming Language, Except For All the Others

Finally, we conclude this section with some observations about the C++ language itself, independent of the issues of when and how to use the two forms of polymorphism. C++ is described by its critics as an ugly and overly complicated language, with an unnecessarily steep learning curve. Unfortunately, there is much truth to this description. The observation of one of the authors (Budge) as a member of X3J16, the ANSI committee that has produced the standard for C++, is that much of the ugliness and redundancy arose out of a desire to maintain as much backwards compatibility as possible. The committee also seemed to believe that supporting as many programming paradigms as possible was more important than orthogonality. These are important considerations, but the effort to satisfy them has made C++ a difficult language to master.

Another criticism that is still occasionally heard is that C++ is not as fast a language as (for example) FORTRAN-77. Our experience is that this is simply not the case if a decent compiler is used. There *is* some tendency for novice C++ programmers to use programming idioms that carry an excessive run-time overhead, but nothing in the language requires the use of these idioms, and there are almost always equally elegant ways to write the code that do not carry such an overhead. (Often, this consists of the use of genericity in place of inheritance.)

Finally, our group has been criticized for using a language that is not the most widely accepted in our community. It is still the case that a scientific or engineering program written in FORTRAN will be readable to a larger fraction of the community than one written in C++. This puts us in roughly the same situation as the first Western scholars to publish learned papers in a language other than Latin. It is difficult to know how to respond to this criticism. Outside the scientific community, C++ is probably more widely understood than FORTRAN, and it is certainly more widely taught in the universities. It is an unhappy reality that scientific programming no longer dominates the market. This will force some changes in how scientific programmers go about their business.

Our feeling is that C++, for all its faults, is currently the best choice of language for writing a large simulation framework. FORTRAN-77, and even FORTRAN-90, fail to provide many features we have grown accustomed to, including both kinds of polymorphism. Java fails to provide genericity or expanded types, which in our minds rules it out for serious programming. Eiffel is an attractive and powerful language, but it has an even smaller following in the scientific community than C++, and at times the structure of this language seems downright hostile to symmetric operations.

3 Lessons Learned

The most painful lesson we learned while developing ALEGRA is that one cannot ignore efficiency issues on the theory that getting the algorithm right and the interfaces clean comes before tuning the code. Our original database architecture, based on smart arrays of various data types (scalars, vectors, and tensors), allowed us to correctly implement our algorithms in a fairly

elegant manner in a relatively short time. Sadly, it proved nearly impossible to make this architecture efficient. The architecture that proved efficient was just as elegant, but respected certain realities about how the data was stored and processed by the computer.

This leads to the second lesson learned: Representation is important, and it cannot always be hidden behind the interface of a framework or class library. Smart arrays look different from lists, and it should not be surprising to learn that replacing our smart array database architecture with a list-based architecture required some fairly extensive rewriting of code. A corollary of this second lesson is that loops are not evil. In fact, the Standard Template Library has illustrated that loops (and iterators) can be beautiful.

Finally, much of the success we have had in adapting the ALEGRA framework to accommodate many developers has come from our use of development tools rather than the choice of language and programming paradigm. Regression testing has become indispensable to our development efforts, as has intelligent version control and formalized bug tracking. We would do even better to institute formalized documentation and code inspection procedures, but certain institutional barriers have made this difficult. Code inspection requires skilled programmers to sit in a room and talk for an hour or more rather than sit at their terminals and write code, and the benefits (though very substantial) are not always highly visible. This is unfortunate, considering the ample body of evidence in favor of code inspection.

4 Conclusions

Our experience with ALEGRA is that it is not always possible to hide data representations within a framework or class library interface. We therefore see a bright future for generic programming (templates) in scientific and engineering programming. As we see it, general-purpose algorithms can be implemented either by allowing the type of the objects operated on to conform to the expectations of the algorithm (via inheritance and virtual functions) or by allowing the signature of the algorithm to conform to the type of the objects (via function templates.) The former requires run-time mechanisms and at least some awareness of the data representation, while the latter does not. This suggests that templates will always be more efficient. We also believe that templates will prove to be more flexible. A function template makes no assumptions about its target other than how certain operations are spelled, whereas a function operating on an abstract base class assumes that the target is a child of the base class. This is problematic when class hierarchies are developed independently. It is more problematic when an algorithm is applicable both to user-defined types and to built-in types.

There is no magic solution to programming challenges. Elegant and reusable software is an expression of visionary thinking, and we don't know how to automate creativity. The best we can do is to provide a flexible framework in which to express flashes of insight when they come. We believe that the constructs provided by C++, including operator overloading, inheritance, and templates, provide such an environment.

References

- [1] L.M. Taylor and D.P. Flanagan, *PRONTO 2D: A Two-Dimensional Transient Solid Dynamics Program*. SAND86-0594 (1984). Sandia National Laboratories, Albuquerque, NM.
- [2] J.S. Peery, K.G. Budge, A.C. Robinson, and D. Whitney, *Using C++ As A Scientific Programming Language*, CRAY User's Group Conference, Santa Fe, NM, Sept. 23-27, 1991.
- [3] J.S. Peery, *RHALE++: A Next Generation Strong Shock Wave Physics Code Developed in C++*, Copper Mountain Conference on Iterative Methods, Copper Mountain, CO, April 9-14, 1992.
- [4] J.S. Peery and K.G. Budge, *Experiences in Using C++ to Develop a Next Generation Strong Shock Wave Physics Code*, ASCE 8th Conference for Computing in Civil Engineering, Dallas, TX, June 10-12, 1992.

- [5] K.G. Budge, J.S. Peery, and A.C. Robinson, *High-Performance Scientific Computing Using C++*, USENIX++ Technical Conference, Portland, OR, Aug. 10-14, 1992.
- [6] K.G. Budge and J. S. Peery, *RHALE: A MMALÉ Shock Physics Code Written in C++*, International Journal of Impact Engineering **14**:1-4 (1993).
- [7] K.G. Budge, *PHYSLIB, A C++ Tensor Class Library*, SAND91-1752 (1991), Sandia National Laboratories, Albuquerque, NM.
- [8] M.K. Wong, K.G. Budge, J.S. Peery, and A.C. Robinson, *Object-Oriented Numerics: A Paradigm for Numerical Object-Oriented Programming*. Computers in Physics **7**, 6 (1993).
- [9] K.G. Budge, J. S. Peery, A. C. Robinson, and M. K. Wong, *C++ as a Language for Object-Oriented Numerics*, presented to the European C++ Users Group, July 1993.
- [10] K.G. Budge, J.S. Peery, A.C. Robinson, and M.K. Wong, *C++ and Object-Oriented Numerics*, Journal of C Language Translation **5**, 32 (1993).
- [11] J.R. Weatherby, J.A. Schutt, J.S. Peery, R.E. Hogan, and S.W. Attaway, *An Object Oriented Finite Element Code Architecture for Massively Parallel Computers*, Supercomputing '93, Nov. 1993.
- [12] K.G. Budge, J. S. Peery, A. C. Robinson, and M. K. Wong, *Management of Class Temporaries in C++ Translational Systems*, Journal of C Language Translation **6**, 2 (1994).
- [13] M.K. Wong, J.S. Peery, and R.M. Summers, *Development of High Efficiency Finite Element Codes in C++*, presented at OONSCI '95.
- [14] A.C. Robinson, J.S. Peery, M.K. Wong, and R.M. Summers, *High Efficiency Parallel Production Code Development Using Finite Elements within the Framework of C++*, presented at POOMA '96 (December 1996).
- [15] A.D. Robison, *C++ Gets Faster for Scientific Computing*, Computers in Physics **10** (1996), pp. 458-462.
- [16]. T. Veldhuizen, *Expression Templates*, C++ Report **7**:5 (1995), pp. 26-31