Title:

COMPONENT ARCHITECTURE OF THE TECOLOTE FRAMEWORK

CONF-981207--

Author(s):

Mark E. Zander, John H. Hall, James W. Painter

Submitted to:

ISCOPE, Santa Fe, NM, December 8-11, 1998

MASTER

DISTRIBUTION OF THIS DOCUMENT IS UNLIMITED

Los Alamos
NATIONAL LABORATORY

Los Alamos National Laboratory, an affirmative action/equal opportunity employer, is operated by the University of California for the U.S. Department of Energy under contract W-7405-ENG-36. By acceptance of this article, the publisher recognizes that the U.S. Government retains a nonexclusive, royalty-free license to publish or reproduce the published form of this contribution, or to allow others to do so, for U.S. Government purposes. Los Alamos National Laboratory requests that the publisher identify this article as work performed under the auspices of the U.S. Department of Energy. The Los Alamos National Laboratory strongly supports academic freedom and a researcher's right to publish; as an institution, however, the Laboratory does not endorse the viewpoint of a publication or guarantee its technical correctness.

DISCLAIMER

This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor any of their employees, make any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

DISCLAIMER

Portions of this document may be illegible in electronic image products. Images are produced from the best available original document.

Component Architecture of the Tecolote Framework

Mark Zander¹, John Hall¹, Jim Painter¹, Sean O'Rourke¹

Los Alamos National Laboratory

RECEIVED

OCT 05 1998

OSTI

Abstract. Los Alamos National Laboratory's Tecolote Framework is used in conjunction with other libraries by several physical simulations. This paper briefly describes the design and use of Tecolote's component architecture. A component is a C++ class that meets several requirements imposed by the framework to increase its reusability, configurability, and ease of replacement. We discuss both the motives for imposing these requirements upon components and the means by which a generic C++ class may be integrated into Tecolote by satisfying these requirements. We also describe the means by which these components may be combined into a physics application.

1 Introduction

Los Alamos National Laboratory's Blanca project is part of the Department of Energy's Accelerated Strategic Computing Initiative (ASCI), which focuses on science-based nuclear weapons stockpile stewardship through the large-scale simulation of multi-physics, multi-dimensional, stockpile-relevant problems. Blanca is the only Los Alamos ASCI project written entirely in C++. Tecolote, the underlying framework for the development of Blanca physics codes, provides an infrastructure for combining individual component modules to create large-scale applications that encompass a wide variety of physics models, numerical solution options, and underlying data storage schemes, activating only essential components at run-time [3]. Tecolote maximizes code re-use and separates physics from computer science as much as possible. This allows physics model developers to use the Parallel Object-Oriented Methods and Applications (POOMA) framework, upon which Tecolote is layered, to write algorithms in a style similar to the problem's underlying computational physics equations [4].

POOMA contains architecture and parallelism abstractions that allow the user to write parallel physics codes without worrying about the underlying architecture or communications libraries. POOMA provides C++ fields that are similar to Fortran-90 arrays, but have additional features, including domain decomposition, load balancing, communications, and compact data storage. POOMA's unique capabilities provide the methods developer with powerful tools for expressing various mesh types and multiple dimensions; this allows application developers to write mesh- and dimension-independent physics code whenever possible. Combined with Tecolote, POOMA's flexibility allows us to keep pace

with the ever-changing ASCI environment, rapidly prototype ideas, and build on what others have done rather than using valuable time to reimplement the same basic models on different architectures.

Tecolote is portable to all ASCI-relevant hardware, making full use of its available parallelism. By supporting the rapid implementation of physics models and their immediate application to problems on the ASCI scale, Tecolote provides a powerful and flexible run-time environment that allows users to create and compose physics codes with varying capabilities "on the fly."

2 Approach

As we will discuss later, the Tecolote Framework supplies an application programmer interface that supports factorization of applications into components. Factorizing an application enhances the programmer's ability to cleanly separate interfaces from implementations, encapsulate conceptually independent subparts of a program, avoid code duplication, and maximize code reuse. After factorization, the user can integrate desired components using techniques supplied by Tecolote. Key concepts supported by this component architecture include separation of computer science from physics in simulations, implementation-independence of component interfaces, and increased run-time configurability (through an input-file scripting language). This flexible approach is made possible through the use of C++ inheritance and virtual function polymorphism. However, a Tecolote component's increased modularity comes at a price: creation of and communication between components can be more expensive than corresponding operations for ordinary C++ objects. Thus, the components' granularity must be large enough that these operations do not impact the code's efficiency.

Tecolote facilitates factorization through several mechanisms:

- uniform run-time data-sharing interface (the `DataDirectory`) with dynamic scoping rules;
- facility for the run-time description of components' type and inheritance relations (`MetaType`) and the registration of this information in a single type table (`MetaSet`);
- means of configuring both data values and control flow without recompilation (the input file scripting language); and
- separation of I/O and computation through the designation of "persistent" data intended for I/O.

The remainder of this paper illustrates the above-described mechanisms and their interaction through the extended example of a gamma-law equation of state (EOS) model.

3 Setting Component Parameters - Persistents

In object-oriented programming, I/O methods are generally encapsulated in application classes. To adopt a new I/O format, whether binary instead of ASCII

text or eight-digit instead of six-digit floating-point output, every application class must be modified.

In Tecolote, we separate what is needed for I/O from how I/O is executed. The "what" is specified by the application programmer in a persistent list that contains the class data members available for I/O. Another component, an I/O module, determines how I/O is actually executed. The I/O module knows how to extract persistent locations from objects' MetaTypes and how to perform some type of I/O operation. There may be several different I/O modules in a system, each corresponding to a different data format.

For example, GammaLaw class members are

```
REAL          pmin;      // minimum pressure
REAL          gamma;     // adiabatic gamma
```

The persistents are listed outside the class declaration:

```
template< class C >
BEGIN_PERSISTENT( GammaLaw< C > )
    PERSISTENT( REAL, pmin, "pmin" )
    PERSISTENT( REAL, gamma, "gamma" )
END_PERSISTENT
```

Persistents support factorization by separating I/O modules from application modules and by deferring decisions about data initialization until run-time. Factoring I/O out of application objects ensures consistent input and output formatting with little burden on the application programmer; it also localizes changes required to support new data formats.

4 Sharing Fields Between Components - DataDirectory

Two models may use different (or the same) fields to compute their respective results. However, because we want to use virtual-function polymorphism to call the two models interchangeably, the models must use the same calling sequence in their respective evaluation functions. To avoid passing different fields in the argument lists of the evaluation functions, we have developed another alternative: passing a single data structure to the EOS model constructor, which then holds all the fields needed for a material. This data structure is termed the **DataDirectory**.

A **DataDirectory** is actually just like any other Tecolote component, except that it may have any number of persistents with any names. In contrast, ordinary components may only contain the persistents specified in their persistent lists. Any object, including another **DataDirectory**, may be placed inside a **DataDirectory**. Thus the **DataDirectory** structure is hierarchical, much like a Unix directory structure. Two entries are automatically put in a newly created **DataDirectory** to allow traversal of the hierarchy: "Root," which points to the **DataDirectory** at the top of the hierarchy, and "Parent," which points to the immediate predecessor in the hierarchy of the current **DataDirectory**.

Fig. 1. The DataDirectory hierarchy

Figure 1 shows the DataDirectory hierarchy used for a multi-material simulation (for simplicity, only a few POOMA Fields are shown here). Unlike a Unix file structure, the DataDirectory structure has scoping rules similar to those of C++ inheritance. However, whereas C++ scope is determined by compile-time class inheritance relations, DataDirectory scope is determined by run-time object nesting. For instance, when GammaLaw attempts to get the PhysicsMesh from the "Material" DataDirectory, it fails to find it. Therefore, the search continues up the hierarchy, examining the "Material Set" and "Root" directories and terminating after it finds the requested PhysicsMesh in the "Root" directory.

The example below, from the GammaLaw class, illustrates the use of the DataDirectory macro GET:

```
ScalarField<C>& IntEnergy(
    GET("IntEnergy", Mat, ScalarField<C>, (Mesh))
);
```

The first argument is the name of the requested item; the second is the DataDirectory in which the search starts (Mat[erial] is a DataDirectory); the

third is the type of the `DataDirectory` item; and the fourth (if present) represents the constructor arguments that are needed if the item is not present and must be added to the `DataDirectory`. The `GET` macro returns a reference to the object found in the `DataDirectory`.

The `DataDirectory` improves code factoring because a single data structure is passed to methods that otherwise would use different calling sequences. By deferring the association of data with a particular model until its actual instantiation, the `DataDirectory` structure provides generalized parameter-passing without explicit parameter lists. It enhances integration by supplying a mechanism for transparent data sharing among independent modules.

5 Building Components from an Input File - Scripting

At the start of its execution, a Tecolote program must specify which modules to use, their initial data values, and the high-level control structure in which they are applied. Tecolote uses a different component for each option and employs persistents to fill in the data needed by the options. Therefore, a Tecolote program built from components must use a methodology that creates objects from its components and places persistent data in the objects. We incorporate a scripting language into the Tecolote framework to accomplish these tasks.

Each object is described by an object name, a `MetaType` name, and a list of persistent values. Object hierarchy (or nesting) is indicated by listing one object in the persistent list of another. The following example shows nested objects where a `GammaLaw` is created as the `Eos` persistent of a `Material`. An object's constructor is called before its persistents are loaded. Therefore, an optional initialize function can be called after an object's persistents have been loaded to perform further initialization that is dependent on persistent values.

```
gas = Material(  
  Eos = GammaLaw(  
    gamma = 0.5,  
    pmin = 0.001  
  )  
)
```

In both debugging and actual use, it is desirable to change the control flow of a program without rebuilding the entire code. Tecolote provides this flexibility by allowing the user to specify higher level function sequences in the input file. The necessary facilities are already provided by Tecolote and may be extended by an application programmer through a `MetaType` that maps methods and functions into function objects available from the input file.

The language, based on Backus' FP language [1] and Robison's IFP [5], includes program-forming operations (PFOs), basic objects, and elementary operations. The PFOs include control structures such as branching (If), looping (While), and sequences (Compose). Examples of basic objects are Lists, Numbers, and strings. Elementary operations that act on basic objects might

include arithmetic, comparison, and collection functions such as concatenation and length. In Tecolote, as in FP, users may define new objects and functions, but not new PFOs.

The input file describes the initial object hierarchy of a program. In addition, it selects which components will be used as well as the initial values of their persistents. The functional scripting language allows the programmer, by combining function objects using PFOs, to define an application's high-level behavior.

6 Registering a Component with Tecolote - MetaTypes

In most applications, a module must directly reference other modules with which it interacts. This requirement obstructs factorization and prevents the application programmer from deferring module interactions until run-time. In addition, it is tedious to find all references to a module when replacing a module referenced in many locations within the program. In contrast, Tecolote components are registered only once – in the **MetaSet**, a table containing all the program components (see Figure 2). Individual components interact only indirectly, through the **MetaSet**, promoting component independence.

A module is registered as a component with Tecolote by using a **MetaType** which, when invoked, automatically registers itself with the **MetaSet**. An object's **MetaType** is an object in its own right, much like Java's "Class" class [2]. The **MetaType** for a class:

- has a name in the **MetaSet**,
- holds the persistent list for that class,
- can create and initialize that class,
- carries its C++ type information to run-time, and
- can convert the **MetaType** from or to a single base class.

Many languages create a unified type hierarchy by having a single base type from which all other classes are derived (Java's Object class, for example). Tecolote classes, on the other hand, do not share a common base class. Eliminating this restriction allows Tecolote to incorporate classes not written for the framework (such as standard container classes and C++ basic types). Non-Tecolote classes are incorporated into the Tecolote type system by describing their basic features and their persistents in a **MetaType**.

In the example below, the **GammaLaw<Cell>** class is registered with the framework.

```
#include "GammaLaw.hh"
static MetaTecolote<GammaLaw<Cell>, Eos>
    GammaLawMeta("GammaLaw", MAKE_PERSISTENTS(GammaLaw<Cell>));
```

The generic **MetaTecolote** class is used to instantiate objects of classes with a constructor that takes **DataDirectory** and **string** arguments. Other **MetaTypes** are available that instantiate objects of classes with a constructor that takes no arguments. **MetaTecolote** is best used for classes that need more information

about their environment, while other **MetaTypes** are used for classes written without knowledge of Tecolote. The class **GammaLaw<Cell>** is given the name "GammaLaw" in the **MetaSet** and has the base class **Eos**. The **MAKE_PERSISTENTS** macro registers the persistent information defined for the **GammaLaw** class.

Fig. 2. Conceptual diagram of the Tecolote Framework

All **MetaTypes** in the system register themselves with the **MetaSet**, a table that may be searched either by the **MetaType** name or by the type of the class that the **MetaType** contains. Although all modules are registered identically, physics modules and I/O modules interact with the **MetaSet** differently. Physics modules do not make explicit use of the **MetaSet**; however, I/O modules use the **MetaSet** explicitly both to build object hierarchies from input and to output program data. One example of this interaction, mentioned above, is the use of persistent information to perform output. Input modules also use a **MetaType**'s constructor, initializer, and base class in creating the object hierarchy.

Two I/O modules shown in Figure 2 are the **Parser** and the **Printer**. The **Parser** reads data from the input file and then finds the **MetaType** by name in the **MetaSet**. After finding the **MetaType**, the **Parser** creates the object, loads its persistents, and calls its initializer. When an object is passed to the **Printer**, the **Printer** uses its C++ type to find the corresponding **MetaType** in the **MetaSet**. It then prints the **MetaType** name, scans the persistent list, and prints each sub-object in turn.

MetaTypes and the **MetaSet** are key elements for supporting factorization in Tecolote. **MetaTypes** turn C++ classes into components, and the **MetaSet** may

be searched for any component in an application program, which provides a level of indirection between components.

7 Conclusion

By using the various techniques in the Tecolote Framework, the Los Alamos National Laboratory's Blanca Project has been able to integrate a wide variety of physics packages into codes with relative ease. We add new physics models by modifying only the code in the `MetaSet`, without rewriting any of the code's I/O modules. Complex data sharing is accomplished by the `DataDirectory`, which allows us to avoid complex calling sequences or global variables. By deferring the choice of components until they are specified in the input file at run-time, we ensure maximum flexibility in the code. The combined benefits from this component architecture approach ensure simpler, faster, and less error-prone means for adding new physics modules to a program.

Our future work will address the effects of Tecolote's component architecture and functional scripting language. Potential areas of investigation include application programming methodologies in a functional scripting language, the high-level expression of parallelism in the component architecture (given functional-language guarantees), and the applicability of a functional language in specifying object hierarchies and application control flow.

References

1. J. Backus. Can Programming Be Liberated from the von Neumann Style? A Functional Style and its Algebra of Programs. In *Communications of the ACM*, pages 613-641, August 1978.
2. J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley, August 1996.
3. K.S. Holian, L.A. Ankeny, S.P. Clancy, J.H. Hall, J.C. Marshall, G.R. Mcnamara, J. W.Painter, and M.E. Zander. TECOLOTE, an Object-Oriented Framework for Hydrodynamics Physics. In *Proceedings of the Conference on Numerical Simulations and Physical Processes Related to Shock Waves in Condensed Media*, Oxford, England, September 1997.
4. J. V. W. Reynders, J. C. Cummings, P. J. Hinker, M. Tholburn, S. Banerjee, M. Srikant, S. Karmesin, Atlas S., K. Keahy, and W. F. Humphrey. *POOMA: A Framework for Scientific Computing Applications on Parallel Architectures*. MIT Press, 1996.
5. A. Robison. Illinois functional programming: A tutorial. *BYTE*, pages 115-125, February 1987.