Using Design Patterns and XML to Construct an Extensible Finite Element System*

J. Barr von Oehsen[1], Christopher L. Cox[2], Eric C. Cyr[2], and Brian A. Malloy[3]

[1] CAEFF, Clemson University, Clemson, SC 29634, USA
vonoehse@ces.clemson.edu

[2] Department of Mathematical Sciences, Clemson University
Clemson, SC 29634, USA
clcox@ces.clemson.edu

[3] Department of Computer Science, Clemson University
Clemson, SC 29634, USA
{ecyr,malloy}@cs.clemson.edu

Abstract. We present an object-oriented finite element software package which employs XML and design patterns in order to solve a problem which is fundamental to the simulation of viscoelastic flows. The XML format increases flexibility in data handling, while design patterns provide a high-level organizational structure. The application problem and the finite element methodology are described, along with the opportunities presented for incorporation of the software tools. Numerical experiments comparing results with and without XML and design patterns are also presented.

1 Introduction

The goal of the Center for Advanced Engineering Fibers and Films is to develop computational models which predict final fiber and film properties based on processing conditions. A primary goal of the CAEFF modeling group is to develop a widely-applicable finite element software package for modeling of viscoelastic flows that is easily maintainable and fully modular, so that the code can be readily customized to the user's choice of material parameters, governing equations, solution domain and computer platform. Most efforts to improve finite element codes have concentrated on the linear or nonlinear system solvers, because normally this stage dominates the overall CPU time. For applications such as ours, it is also worthwhile to make the assembly phase as flexible as possible with efficiency also a major consideration. This flexibility will be especially important when the code is ported to a parallel environment, where both assembly and solution phases are distributed over many processors. Ease of handling data in a variety of formats is a necessary aspect of the software package, because the target code will link the simulation routines with an experimental database and advanced visualization software.

* This work was supported primarily by the ERC program of the National Science Foundation under Award Number EEC-9731680.

In this paper, we present the design and implementation of a prototype system which will undergird a viscoelastic flow simulation package. Our model includes several design patterns that assure ease in maintenance and modification of the code. Design patterns help developers resolve software architecture bottlenecks and provide a common vocabulary for building a simulation package. Those familiar with the patterns used to develop a system, will find the code easier to read and maintain. The end-user, likely one who is not familiar with object-oriented programming, will see modules identified according to application.

In the next section we provide background about XML and the design patterns that we use in our system. In Section 3, we describe our finite element solution to a prototype viscous flow problem that we consider, and in Section 4, we provide an overview of the design of our system and the flow of data through the system. In Section 6 we describe our use of XML to present our output to any viewer for which an accompanying XSLT style sheet is provided. In Section 7 we describe the results of comparing our patterned approach with a non-patterned approach also written in C++. Finally, in Section 8, we draw conclusions and describe our ongoing work.

2 Background

In this section, we provide background about XML and design patterns, including an overview of the Factory Method and Singleton patterns that facilitate our modeling of a finite element solution of partial differential equations[13, 4].

2.1 XML

The extensible markup language, XML, has become one of the hottest concepts in computer science, web authoring and web programming. Like its HTML counterpart, XML is derived from the standard generalized markup language, SGML. However, HTML is a markup language used for displaying information content; XML, on the other hand, is a markup language for creating markup languages. HTML is a markup language for marking documents using tags for headings and paragraphs, whereas XML enables the creation of new tags for marking anything, such as mathematical formulas, molecular structure of chemicals, music scores and any other document. HTML limits the user to a fixed collection of tags, used primarily to describe the content that is displayed in a browser.

An XML document consists of a list of element types (tags), together with their attributes. The relationships of these elements, to each other, can be specified by an optional XML Schema. XML Schemas express shared vocabularies and allow machines to carry out rules made by people. They provide a means for defining the structure, content and semantics of XML documents. A schema is not required for a document but is recommended for document conformity. By combining an XML document with its corresponding XML Schema, an XML parser can determine the content and structure of an XML document. We have created our own fiber and film XML Schema - which we hope will become the industry standard.

For web authoring, HTML has emerged as the technology of choice for describing the content of an HTML document; cascading style sheets, CSS, has emerged as the technology of choice for describing the form of an HTML document. Similarly, the extensible style language, XSL, was developed as a technology for describing the form of an XML document. An XSL style sheet provides the rules for displaying or organizing an XML document's data. XSL also provides elements that define rules for describing how to transform one XML document into another XML document. For example, an XML document can be transformed into an HTML document. The facet of XSL that addresses the problem of transforming XML documents is called XSL transformations, or XSLT.

2.2 Design Patterns

A fundamental aspect to any science or engineering discipline is a common vocabulary that practitioners can use to express concepts, and a language for relating the concepts. The development of a catalog of design patterns provides such a vocabulary, together with a body of literature to help software developers resolve recurring problems encountered in software development. Patterns describe design practices that capture experience in a way that enables others to reuse this experience. The primary focus is not so much on technology as it is on creating documentation of sound software engineering design practices.

The *Factory Method* is a creational pattern that describes an extensible approach to the construction of objects. This pattern is typically used to create instances of objects described by a class hierarchy. The Factory Method eliminates the need for binding application-specific code about the construction of the objects into the system[4]; instead, the construction is incorporated into a function, or method, that abstracts the details of construction. This method typically accepts a parameter that encodes information about the type of object to create and then the Factory Method, after interpreting the information, constructs the object. If the Factory Method is a class constructor, than this pattern is sometimes referred to as the *Virtual Constructor Pattern*[4].

The *Singleton Pattern* is another creational pattern that permits the developer to restrict creation to a single instance, while providing global access to this instance. To do this, the Singleton class constructor is placed either in the private or protected section, and a *construction function* is placed in the public section of the Singleton class. Instantiation of the Singleton class must be made through a call to the *construction function*, which maintains a count of the number of instantances previously created, as well as a pointer to the single instance. If the Singleton class has already been instantiated, than the pointer to this instantiation is returned; otherwise, an instance of the Singleton class is constructed and the resulting pointer returned.

3 A Finite Element Solution of a Viscous Flow Problem

A mathematical model for the flow of a viscoelastic fluid consists of the standard momentum, mass, and energy balance equations, plus a constitutive equation representing the manner in which the stress depends on the velocity gradient. A common form of these equations for isothermal flow, with inertial terms dropped, is [2]

$$\rho \frac{\partial \mathbf{v}}{\partial t} - \nabla \cdot \boldsymbol{\tau} + \nabla p = \mathbf{f} \tag{1}$$

$$\nabla \cdot \mathbf{v} = 0 \tag{2}$$

$$\boldsymbol{\tau}_p + \lambda_1 \boldsymbol{\tau}_{p(1)} - \alpha \frac{\lambda_1}{\eta_p} \{\boldsymbol{\tau}_p \cdot \boldsymbol{\tau}_p\} = -\eta_p \dot{\gamma}, \tag{3}$$

In (1), ρ is the fluid density, $\mathbf{v}$ is the velocity vector, p is pressure, $\boldsymbol{\tau}$ is the extra stress tensor, and $\mathbf{f}$ is the forcing term vector.

The extra stress has been split according to $\boldsymbol{\tau} = \boldsymbol{\tau}_n + \boldsymbol{\tau}_p$ where $\boldsymbol{\tau}_n$, the Newtonian part, is a constant multiple of $\dot{\gamma} = \nabla \mathbf{v} + (\nabla \mathbf{v})^T$. The polymeric part, $\boldsymbol{\tau}_p$, satisfies a nonlinear differential or integral equation. For example, (3) is known as the Giesekus model. In this equation λ_1, α and η_p are fitted parameters and $\boldsymbol{\tau}_{p(1)} = \frac{d\boldsymbol{\tau}_p}{dt} - (\nabla \mathbf{v})^T \cdot \boldsymbol{\tau}_p - \boldsymbol{\tau}_p \cdot \nabla \mathbf{v}$.

Several finite element formulations for approximating the solution of (1)-(3) appear in the literature. Realistic simulations,in three dimensions, require the solution of systems containing hundreds of thousands or even several million unknowns. Therefore iterative methods are normally chosen over direct methods. One such iterative approach is the θ-method [10], which will be briefly described. For the general problem of finding the solution $\mathbf{U}$ for the system

$$M \frac{d\mathbf{U}}{dt} + A(\mathbf{U}) = 0 \tag{4}$$

with initial condition $\mathbf{U}(0) = \mathbf{U}_0$, the θ-method is based on splitting the nonlinear operator A into a sum of simpler (linear) operators A_1 and A_2. For equations (1)-(3), $\mathbf{U} = [\mathbf{v} \quad p \quad \boldsymbol{\tau}_p]^T$, or more precisely the coefficients in the finite element approximation for each function. The operator A is the nonlinear integral/differential operator which arises in the variational formulation of (1)-(3). One complete θ-method update for (4) on time interval $[t, \quad t + \Delta t]$ consists of three steps. For the operator A associated with the finite element approximation of (1)-(3), a splitting can be chosen so that the second step is effectively a transport equation for $\boldsymbol{\tau}_p$, while the first and third steps dominate the computational work and have the form

$$\begin{aligned} \beta \mathbf{v} - \nu \Delta \mathbf{v} + \nabla p &= \mathbf{f} \\ \nabla \cdot \mathbf{v} &= 0 \end{aligned} \tag{5}$$

which is the Stokes Problem varied by the addition of the $\beta \mathbf{v}$ term.

Because the solution of the Stokes Problem is fundamental to the θ-method for solving (1)-(3), and because the theme of this paper is software design for finite element approximation, we will concentrate on the Stokes Problem, i.e. (5) with $\beta = 0$), with simple Dirichlet boundary condition $\mathbf{v} = 0$ and computational domain the unit square in the first quadrant, with interior denoted Ω and boundary Γ.

The finite element solution is based on the variational formulation, i.e. find $(\mathbf{v},\ p)$ in an appropriate product space of functions (in this case $H_0^1(\Omega) \times L_2(\Omega)$) so that

$$\begin{aligned} \nu \textstyle\int_\Omega (\nabla \mathbf{v} : \nabla \mathbf{w} - p \nabla \cdot \mathbf{w}) &= \textstyle\int_\Omega \mathbf{f} \cdot \mathbf{w} \\ - \textstyle\int_\Omega q \nabla \cdot \mathbf{v} &= 0 \end{aligned} \tag{6}$$

for all $(\mathbf{w},\ q)$ in a similar product space [5]. In (6), $\boldsymbol{\sigma} : \boldsymbol{\tau} = \sigma_{ij}\tau_{ij}$, with summation on repeated indices.

A suitable choice of finite dimensional subspace-pair for the approximate solution of (6) is the Taylor-Hood element on a triangular mesh [12], comprised of continuous piecewise quadratic functions for $\mathbf{v}$ and $\mathbf{w}$, and continuous piecewise linears for p and q. For uniqueness, an additional condition must be imposed on the pressure space. We impose the condition $\int_\Omega p = 0$.

The finite element solution is computed in three steps: mesh generation, assembly, and solution of the matrix system. For mesh generation we use the object-oriented package QMG [9]. An efficient, modular formulation of the assembly stage is especially important for this code because it must eventually be applicable to a range of fluid characterizations (i.e. (3) will change significantly) in various physical settings (e.g. a variety of boundary conditions). Assembly on each triangular element is carried out with the standard technique of mapping to a canonical element, using isoparametric elements to allow for curved boundaries [5]. The use of this mapping has a significant impact on data structures, allowing the use of *singletons* for basis functions and quadrature rules. For this test problem the linear system was solved using banded Gaussian elimination with partial pivoting. A benchmark analytical solution was used to check convergence with respect to mesh refinement, in order to assure that the code was working properly.

4 System Overview

Figure 1 summarizes our system to provide a finite element solution to a viscoelastic flow problem. The two circles on the left of the diagram illustrate input to the system: the top circle on the left represents the mesh, and the bottom circle on the left represents a polymer read from an existing database. Both inputs to our system are in XML format, using a schema that we have developed for viscoelastic flow. Our input is accepted by the `Finite Element Factory`, represented by the square on the left of Figure 1. The `Finite Element Factory` builds the necessary objects, used in the assembly of the linear system.

The `Assembly Routine` is represented by a circle in Figure 1, and uses methods in two other factories to facilitate assembly. These methods are part of our `Basis Function Factory` and `Quadrature Factory`, represented by the

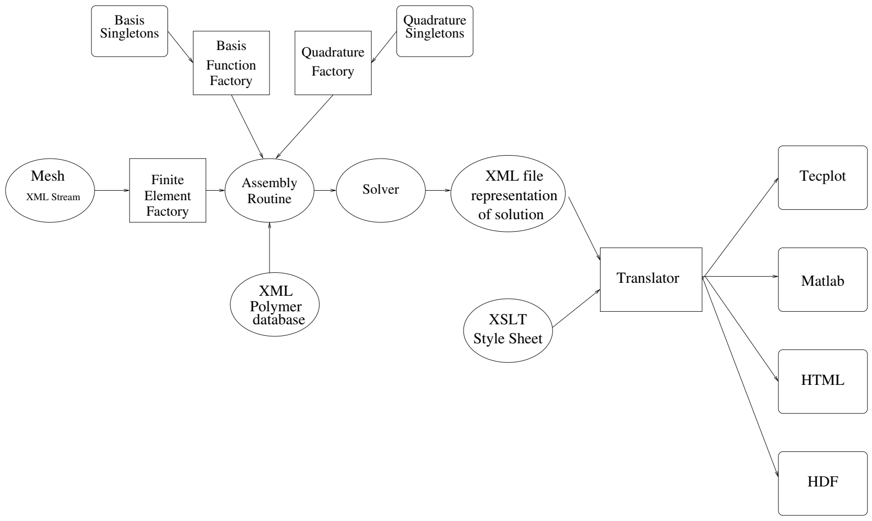

Fig. 1. *System Overview.*

squares at the top of the figure. Once assembled, the mesh is input to the `solver`. The output of the solver is an XML representation of the solution, which together with an `XSLT Style Sheet`, is input to our `Translator`. Using an appropriate XSLT style sheet, the output can be formatted for any viewing tool, such as *Tecplot*, *Matlab*, [1, 6], or any of the other viewing tools illustrated on the right side of Figure 1.

In the next section, we provide details about the `Finite Element Factory and Singleton`, that is a pivotal class hierarchy in our design.

5 Finite Element Factory and Singleton Patterns

The creation of objects (in our case 1D, 2D, and 3D elements, basis functions, and quadratures) occur through common factories (finite element factory, basis element factory, quadrature factory) rather than allowing the creation of these objects to be distributed throughout the system. The advantage of this approach is that if adding a new object to our design is necessary, we need only modify the factory - not the entire system. For example, if we need to add another type of basis function (which we will do when we include 3D models) or constitutive equation to our system, the factory pattern allows this flexibility without having to modify the whole code. Not only does this design allow for easy maintenance, but enables us to make our code modular.

The singleton pattern allows the creation of one and only one instance of an object. This design works particularly well with our system since we are pushing all of our finite element analysis onto a canonical element. There is no need to have multiple copies of a particular basis function object stored in memory if

```
( 1)  < ?xml version="1.0"? >
( 2)  < FINITE_ELEMENT
( 3)     xmlns : xsi = "http : //www.w3.org/2001/XMLSchema − instance"
( 4)      xsi : noNamespaceSchemaLocation = "finite_element.xsd" >
( 5)        < NODES NUMBER_OF_NODES = "1681" >
( 6)        < NODE GLOBAL_NODE_NUMBER = "0" >
( 7)        < VECTOR DIM = "2" >
( 8)        < COORD > 0 < /COORD >
( 9)        < COORD > 0 < /COORD >
(10)        < /VECTOR >
(11)        < VELOCITY >
(12)        < VECTOR DIM = "2" >
(13)        < CRD > 0 < /CRD >
(14)        < CRD > 0 < /CRD >
(15)        < /VECTOR >
(16)        < /VELOCITY >
(17)        < PRESSURE >-104.249< /PRESSURE >
(18)        < /NODE >
(19)        < /NODES >
(20)        < ELEMENTS NUMBER_OF_ELEMENTS = "800" >
(21)        < ELEMENT GEOMETRY = "TRIANGLE_SIX_POINT"
(22)          GLOBAL_ELEMENT_NUMBER = "0" >
(23)        < VECTOR DIM = "6" >
(24)        < COORD > 0 < /COORD >
(25)        < COORD > 2 < /COORD >
(26)        < COORD > 84 < /COORD >
(27)        < COORD > 1 < /COORD >
(28)        < COORD > 43 < /COORD >
(29)        < COORD > 42 < /COORD >
(30)        < /VECTOR >
(31)        < /ELEMENT >
(32)        < /ELEMENTS >
(33)  < /FINITE_ELEMENT >
```

Fig. 2: *XML Example.* This figure illustrates our use of XML to describe the data structures that we use in our modeling of a viscous flow problem. Our approach can be applied to most mathematical models where data needs some form of visualization.

all instances are exactly the same. Thus, the singleton pattern allows us to save on the memory used and also increase the overall performance of the assembly routine. Moreover, we allow global access to the singleton object exclusively through a static member function.

6 XML Representation of the Data

Figure 2 illustrates our use of XML to describe the finite element solution of a viscous flow problem. The finite element XML Schema includes XML tags to describe nodes and elements used in finite element analysis. These tags are represented by `NODES` and `ELEMENTS` on lines 5 and 20 in the figure. The children of `NODES` are one or more `NODE` elements, which are composed of coordinate vectors, velocity vectors and pressure, illustrated on lines 6 through 18 in the figure. We have abbreviated the XML representation but in the actual system these XML elements completely describe our finite element solution. Unfortunately, the solution is almost meaningless if one can not use other tools to view the results. Here lies the problem - different tools require different file formats. It is precisely this reason that flexibility in translation is absolutely essential.

Depending on the solution of the linear system and the visualization tool that we use, we sometimes need to create more than one formatted file. By exploiting the XML tools made available by the Apache XML project (Xerces and Xalan), the Mozilla Organization (Rhino - JavaScript for Java), and the Simple API for

XML (SAX) we have designed our system so that most decisions of this type are transparent to the user[7, 3, 8, 11]. For example, we accomplish the transparency of creating multiple formatted files by first parsing the XML file to check tags and then set certain values dependent upon the parsed outcome. Once completed, our system then reads in a given XSLT style sheet and substitutes our set values into parameterized conditional statements. The translation only takes place where the conditional statements are true (see Figure 3).

```
( 1)  < xsl:when test="$param='velocity_pressure'" >
( 2)  variables="x","y","u","v","p"
( 3)  < /xsl:when >
( 4)  < xsl:when test="$param='no_pressure'" >
( 5)  variables="x","y","u","v"
( 6)  < /xsl:when >
( 7)  < xsl:when test="$param='no_velocity'" >
( 8)  variables="x","y","p"
( 9)  < /xsl:when >
```

Fig. 3: *Parameterized Conditional Statements.* This figure summarizes how one uses parameterized conditional statements within an XSLT style sheet. If the variable "param" is set to "velocity_pressure," then only the text on line (2) will be used in the translation. Likewise for the other conditional statements.

As shown in Figure 1, once the system of equations has been solved, the results are output into an XML file. The oval to the right of the solver represents these results. The XML representation of the solution, together with an XSLT style sheet, becomes input to a translator, represented by the rectangle toward the right of Figure 1. An XSLT style sheet is tailored to a particular viewing tool and is used to translate the XML representation into a file for that particular tool. Finally, the translated file is used as input to the tool and can be viewed by the user in a variety of file formats.

Using XML has several advantages: (1) ease of maintenance and less code bloat (we do not have to hard code different formats into our system), (2) the user does not need to wait for a software upgrade just so that he or she can use their favorite visualization/computer algebra system tool, (3) writing an XSLT style sheet is not very difficult, (4) the user can save output in multiple formats, and (5) if the user decides later that he or she would rather use a different file format after the fact, then (as long as the user has the proper style sheet) getting a new translation can easily be managed without having to run the finite element code from the start.

7 Results

In this section, we present the results of our object-oriented design and implementation, written in C++, that uses the Factory Method and Singleton patterns. To show the effects of our use of the patterns, we compare our patterned approach to an object-oriented approach without patterns (Fortran legacy code rewritten to be object oriented), also coded in C++.

All of the experiments in this section were executed on a Dell Optiplex GX1 workstation, equipped with a 500 MHz PIII processor and 768 Megabytes of memory. Our operating system is Red Hat Linux 7.1. The C++ compiler is gcc version 3.0.2.

Number of Unknowns	**Experiments with Assembly Routine (sec)**			
	C++ No Patterns		Patterned C++	
	mean	std dev	mean	std dev
5,477	27.76	0.077	8.63	0.01
22,202	113.82	0.210	34.82	0.12
89,402	697.50	6.670	382.43	27.93

Table 1: **Results of Experiments.**

This table illustrates the results of our experiments with two versions of the assembly routine that builds the linear system. Each column reports the results of 50 executions.

Table 1 illustrates the results of our experiments with two versions of the assembly routine that builds the linear system. There are three rows of data in the table, representing the number of unknowns to be solved within the linear system. There are three columns of data in the table, including headings for the experiments with the C++ code that did not use patterns, `C++ No Patterns`, and a heading for the C++ code that used patterns, `Patterned C++`. For each approach, we report the `mean` and `standard deviation` for fifty executions of the assembly routine, making note of the number of unknowns for each experiment. For example, for the experiment with 5,477 unknowns, the C++ code without patterns required an average of 27.76 seconds, and the C++ with patterns required an average of 8.63 seconds.

The difference between the C++ code without patterns and the C++ code with patterns is that the unpatterned code distributes object creation throughout the application; in the patterned code, object creation is localized in the factory method for the respective class framework. For example, the factory method for the finite element framework accepts a dimension parameter and then builds the elements and sets up the bookkeeping necessary for solving a particular problem, simplifying code maintenance.

In comparing the results of the C++ code without patterns to the C++ code with patterns, we see a clear performance advantage to the organization and modularity provided by the patterns. For example, for the test case with 89,402 unknowns, the C++ code with no patterns required on average 697.50 seconds, while the patterned C++ code required on average only 382.43 seconds. This performance benefit is due, in part, to our use of the singleton pattern, which obviates the need to repeatedly create and destroy objects during execution. Instead, we create a singleton of each of the objects and then use this singleton when it is needed.

8 Conclusions and Future Work

In this paper, we have described an object-oriented finite element software package that incorporates XML formatting and design patterns to provide ease of maintenance and extensibility. Numerical experiments indicate no additional cost in CPU time with the new design. Rather, the CPU time for the assembly stage of the finite element process decreased when the code was modified to include XML and design patterns. In the immediate future, we will benchmark the code against a conventional C-language code and Fortran. We are also extending the code to simulate non-Newtonian and viscoelastic flows through complex geometries. An effort is underway to develop a software database of experimental results that will provide input parameters for the code. The technique presented in this paper will be especially useful for each of these next phases.

References

1. Amtec Engineering Incorporated. *Tecplot, Version 9.0.* http://www.amtec.com/, 2001.
2. F. P. T. Baaijens. Mixed finite element methods for viscoelastic flow analysis: a review. *J. Non-Newtonian Fluid Mech.*, 79:361–385, 1998.
3. Fluent. *Fluent Flow Modeling Software.* http://www.fluent.com/, 2001.
4. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.
5. C. Johnson. Numerical solution of partial differential equations by the finite element method. *Cambridge University Press*, 1992.
6. Mathworks Incorporated. *Matlab, Version 6.1.* http://www.mathworks.com/, 2001.
7. Oasis. *SAX developed collaboratively by the members of the XML-DEV mailing list.* http://www.oasis-open.org/, 2001.
8. Mozilla Organization. *Rhino: JavaScript for Java.* http://www.mozilla.org/, 2001.
9. QMG. 2.0 mesh generation software. *www.cs.cornell.edu/home/vavasis/qmg2.0/qmg2_0_home.html*, 1999.
10. P. Saramito. A new θ-scheme algorithm and imcompressible fem for viscoelastic fluid flows. *Math. Modelling Numer. Anal.*, 28(1):1–34, 1994.
11. The XML Apache Organization. *The Apache XML Project.* http://xml.apache.org/, 2001.
12. F. Thomasset. Implementation of finite element methods for navier-stokes equations. 1981.
13. J. B. von Oehsen, R. C. Jenkins, C. L. Cox, and B. A. Malloy. Exploiting xml to provide a uniform interface for graphical representation of finite element analysis. *Proceedings of the International Conference on Computer and Information Systems*, pages 181–185, Oct 2001.