Runtime automatic generation of template static binded code

Tiago Quintino*, **Andrea Lani***, **Herman Deconinck***

* Von Karman Institute for Fluid Dynamics, Aeronautics and Aerospace Dept.
Chaussee de Waterloo 72, B-1640, Sint-Genesius-Rode, Belgium
web page: http://www.vki.ac.be
e-mail: quintino@vki.ac.be

ABSTRACT

In the development of software for computational science and engineering, multiple objectives come into play such as flexibility, efficiency and usability. Often these objectives seem to be driven by opposing forces rather than being complementary and able to be achieved simultaneously. Typically code flexibility, provided by dynamic extension points (hooks) is seen as conflicting with statically bound highly efficient code. Also user-friendliness, both for developers and end-users is affected when complex architectures are designed to cope with high flexibility.

Frameworks are developed to instigate the re-use of design solutions for related families of applications within the domain of scientific computation, while component architectures are developed aiming at the re-use of implementations of specific numerical methods. When these frameworks and components are brought together to create a specific application or a family of applications, a system for binding these components has to be chosen. This choice can either be a dynamic binding or a static binding.

Dynamically bound code allows to build applications which delay configurations options to runtime, and so allow the end-user to control them. This is advantageous for a multi-purpose application, where for example, a user can choose multiple discretization methods or multiple physical models. But dynamic bound code relies on abstraction, virtual dispatches and mechanisms for runtime identification, all of which hinder the task of the compiler when optimizing the code and contribute to impacting on the overall efficiency.

On the other hand, statically bound code, which has seen recently a growing interest by the worldwide scientific computing community in the form of C++ templates using meta-programming techniques, is based on compile time configuration decisions binding components to each other without abstractions and thus promoting a multitude of optimizations by the compiler, leading to extremely efficient code. The limitations are in the flexibility and usability of the code, as each application is a static combination of components which can no longer be changed. This means that after deployment, the end-user has little maneuver in the configuration of the simulations.

Ideally, one would like to have the advantages of both and none of the disadvantages, in the form of a code that could delay to runtime the configuration options of which components are bound together, but removing the dynamic dependencies that affect the performance of the final product.

The C++ language, which has become one of the most used languages for the building of large frameworks and components on the domain of scientific computing [2,3], features both paradigms of development: dynamic binding (via language features like inheritance, virtual dispatch and overriding) and static binding (via template types, template meta-programming and overloading).

Making these paradigms work together would require that at runtime, the functionality present in the application would be enhanced with a new combination of components, statically bound together. But C++ features dynamic configuration only through dynamic polymorphism, via virtual functions and all functionality is predefined before runtime.

The dynamic creation and selection of new components requires language features for runtime modification of the code, which are missing in the C++ language but present in languages like Smalltalk and CLOS. Therefore it is not possible to defer to runtime decisions the configuration of static code, based purely on language constructs.

As an example, in the most inner loop of a finite volume solver lies the computation of a numerical flux based on certain characteristics of the physics, like the number of equations, the definition of physical flux and the eigen-structure of the jacobian of same flux. The physics to be modelled, as a point of variance, should be a end-user defined parameter, and the binding of the numerical flux, which is yet another point of variance related to the accuracy and stability of the computation, should be another end-user defined parameter. If we would want to statically parametrize, using C++ templates, the numerical flux with the physical flux, say a Roe numerical flux with the Navier-Stokes equations, this choice would have to be bound at compile time. If we want to keep the point of flexibility we must use an abstraction of the physics and introduce a virtual dispatched function that will disable code optimization from the compiler, e.g., the number of equations will not be known at compile time, therefore loop unrolling is disabled.

This research work focus on providing an automatic facility for generating C++ templated static binded code, according to user runtime requirements, with the aim of maximizing code efficiency without sacrificing flexibility of the application. With such facility, the problem described in the previous paragraph is overcome by deferring the binding decision to runtime, giving the user the choice which numerical flux to use with which physical model, and still allowing the compiler to perform all its optimizations, via static binding.

This is possible by creating a code generator, which will instantiate providers of concrete template static combinations which are compiled on demand and dynamically loaded into the automatic facility, in practice extending at runtime the available functionality of the running code. This is similar to Java class loading mechanism, based on the concept of ”runtime assembly of code”, but applied to C++ template code.

It must be pointed out that this does not equal to, nor the authors aim at, an idiom to provide the C++ language with pure reflection capabilities. This idiom does not allow class and method discovery and identification, for the class interface and its methods must be clearly known and preexisting.

On the other hand, this technique can be viewed as part of a domain engineering procedure, to be applied with the C++ language to overcome the limitation of C++ not supporting dynamic configuration such as described in the seminal work on generative programming and software domain engineering [1].

The current technique has been applied in the development of COOLFluiD, which is a component based object-oriented framework for high-performance scientific computing developed by the von Karman Institute for Fluid Dynamics.

REFERENCES

[1] K. Czarnecki, U. Eisenecker: Generative programming: methods, tools and applications. Addison-Wesley, 2000. ISBN 0201309777.

[2] OpenFOAM: The Open Source CFD Toolbox `http://www.opencfd.co.uk/openfoam/`

[3] W. Bangerth, R. Hartmann, G. Kanschat deal.II - a General Purpose Object Oriented Finite Element Library ACM Transactions on Mathematical Software, to appear

[4] A. Lani, T. Quintino, D. Kimpe, H. Deconinck The COOLFluiD Framework - Design Solutions for High-Performance Object Oriented Scientific Computing Software *International Conference Computational Science 2005*, LNCS 3514 vol 1, Springer-Verlag, 2005.