

Supporting Generic Programming in a Multi-Language Component-Based Environment

Wael R. Elwasif *, Gregory Brown †, Thomas C. Schultheiss *, David E. Bernholdt *

{elwasifwr, browngp, schulhesstc, bernholdtde}@ornl.gov

* Computer Science & Mathematics Division
Oak Ridge National Laboratory

† School of Computational Science and
Information Technology
Florida State University

Overview

We examine some of the issues and possible solutions involved in supporting generic programming in a multi-language component-based environment which also includes languages that do not support generic programming.

This work stems from an effort to make the Ψ -Mag toolset available in componentized form. The Ψ -Mag toolset is a C++ library for computational materials sciences that makes extensive use of generic programming principles through the use of C++ templates. Our goal is to export the functionality of the Ψ -Mag toolset in componentized form that follows the specification of the Common Component Architecture (CCA). The CCA is an open component environment specifically designed to support high-performance scientific computing.

Accommodating components written in the most commonly used HPC programming languages is one of underlying design goals of the CCA. Language interoperability in CCA is achieved through the use of the Scientific Interface Definition Language (SDL) and its associated Babel compiler. SDL defines an object model that is closely related to the one defined in the Java programming language, which has no support for template-based generic programming. This object model is mapped into implementation objects in the most commonly used HPC languages (C, C++, Fortran, and Python) via the Babel compiler.

In this poster, we present preliminary results in exporting the functionality embedded into the C++, template-based Ψ -Mag toolset libraries to other languages using the SIDL/Babel language interoperability tools. Early results suggest that with proper design of the SIDL object layer that wraps Ψ -Mag objects, the library can be used in a multi-language environment *without the need to alter library internals*.

Goals

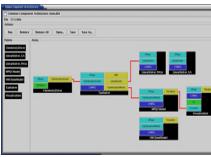
This work aims at exporting the functionality embedded in the Ψ -Mag toolset in componentized form that is compliant with the CCA specification. Towards that end, we address the general problem of accessing C++ template-based generic code in a multi-language environment via the SIDL/Babel language interoperability tools. We work within the boundaries of the existing SIDL object model to seek design patterns and programming techniques that facilitates such access without the need for extensive modifications to the underlying generic library design and/or code. We look for a combination of manual and automated solution that simplify the process whenever possible.

Technical Issues

Support for generic programming in a multi-language component-based environment is particularly challenging for two reasons. First, many languages do not support generic programming. While the language interoperability layer can be used in many cases to augment the target language, for example, by providing an object model to non-object oriented languages, generic programming requires more extensive modification to the base language. Second, C++'s compile-time mechanism for resolving templates based on how they are used is at odds with the idea that components need have no "knowledge" of how they will be used until the application is assembled, at runtime. This later reason is further broken down into two main issues, type mapping and object instantiation. The type mapping challenge stems from the need for template parameters to be *native* C++ classes with interfaces (contracts) that are well defined in the underlying library. Any attempt to extend these interfaces would require (sometimes prohibitively) extensive modifications to the underlying library that may render such an endeavour impractical. The issue of instantiation stems from the ability of the C++ compiler to determine *at compile time* the concrete types used to parameterize a class template. This information is used to instantiate a version of the class template that uses those concrete types.

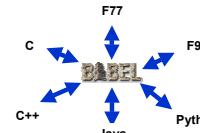
Common Component Architecture - CCA

- A component model and tools for HPC codes.
- Supports both parallel and distributed component applications.
- Takes a minimalist approach that facilitates migrating legacy codes.
- Support for major HPC languages through Babel interoperability tool



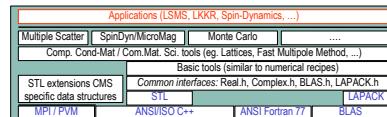
Language Interoperability using SIDL/Babel

- SIDL interface definition language supports Java-like object model
- Babel compiler generates glue code that binds servers to clients written in other HPC languages.
- Servers implement SIDL objects and interfaces (sometimes as lightweight wrappers).



The Ψ -Mag Tool Set

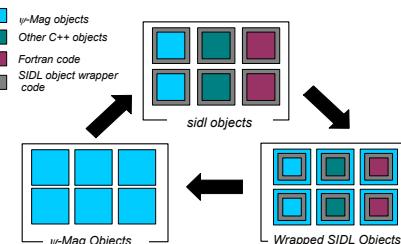
The Ψ -Mag tool set is a C++ library for computational magnetism from which scientific applications can be efficiently built. Using generic programming techniques and C++ templates, it provides common data structures, e.g. vectors, multidimensional histograms, and hierarchical trees, as well as numerical algorithms, e.g. special functions, integration, and random number generators. It also provides tools specific to computational magnetism, from methods for calculating the energies and fields of particular configurations of the magnetization to the Fast Multipole Method for calculating dipole-dipole interactions in large systems. These tools can be incorporated into applications ranging from first-principles quantum mechanics calculations, to Monte Carlo simulations of semiclassical spin models, to phenomenological micromagnetics simulations.



Conceptual organization of the Ψ -Mag tool set (black) along with the application layer (orange) and support layer (blue).

Approach

- Multiple SIDL interfaces are instantiated that support all possible types used in a generic Ψ -Mag interface.
- A single SIDL class implements all possible interfaces, using a factory design pattern to select which interface (and underlying Ψ -Mag class) is instantiated.
- SIDL objects wrap underlying Ψ -Mag objects.
- SIDL Method arguments are wrapped *on the fly* into Ψ -Mag compliant interfaces that are passed down to native Ψ -Mag code (this allows Ψ -Mag to be extended by code written in other languages).



The Methodology explained

```
class Gauss {
public:
    template <class Random, class Real>
    Real evaluate(Random &urand, Real variance)
    { return std::sqrt(variance)*next(urand); }

private:
    .....
};
```

Templated C++ Ψ -Mag class public Interface. This class generates a Gaussian random distribution using a random number generator (RNG) that is used as a template parameter.

The template parameter `Real` is used to specify the base type for the class. It can specify single precision (`float`) or double precision (`double`). An alternative method is to globally define `Real` in a header file, however such solution would not work in a mixed language environment.

The SIDL interfaces

```
package psimagConcept {
    interface RDistD {
        double evaluate[d](in RNGD u,in double p);
    }
    interface RDistF {
        float evaluate[f](in RNGF u,in float p);
    }
}
package psimagModel {
    class Gauss implements-all RDistD, RDistF {
        static RDistD createRDistD();
        static RDistF createRDistF();
    }
}
```

SIDL interfaces and class declaration for wrapping the Ψ -Mag class `Gauss`. SIDL code is processed using the Babel compiler to generate implementation skeletons for SIDL classes. Such skeletons are completed by the user to provide the logic behind the interfaces.

Since SIDL does not support templates, multiple instances of the same interface are needed to accommodate the use of multiple base types (`float` and `double` in this case). One SIDL class implements both interfaces. A factory design pattern is used to mediate the export of one of the two interfaces to clients. It should be noted also that the same argument applies to the Random Number Generator (RNG) interface, with interface `RNGD` defined in terms of double precision data type and `RNGF` for single precision data type. The `static` class methods allow a factory to instantiate and return one of the two interfaces implemented, without direct client access to the `Gauss` class itself.

Implementing the SIDL wrapping object layer

```
/** 
 * Method: evaluate[]
 */
double
psimagModel::Gauss::evaluate (
    /*in*/  ::psimagConcept::RNGD urng,
    /*in*/  double variance
) throw ()
{
    // DO-NOT-DELETE
    splicer.begin(psimagModel.Gauss.evaluate)
    // insert implementation here
    ::psimagWrap::RNG<double> uWrap(urng);
    return wrapped_obj.evaluate(uWrap,variance);
    // DO-NOT-DELETE
    splicer.end(psimagModel.Gauss.evaluate)
}
```

Implementation of the `evaluate(RNGD urng, double variance)` interface. The code in *italics* is automatically generated from the SIDL specification via the Babel compiler. The code builds a Ψ -Mag compliant object (`uWrap`) around the SIDL object `urng`. This new object is then used as an argument (along with the double precision parameter) in the call to the underlying wrapped Ψ -Mag object.

Every object argument is wrapped on-the-fly into a Ψ -Mag compliant object interface. The use of the namespace `psimagWrap` separates those wrapper classes from the native Ψ -Mag classes

Wrapping SIDL objects

```
namespace psimag {
template <class SIDLType>
class SIDLWrap {
public:
    SIDLWrap() { unbind(); }
    SIDLWrap(SIDLType* ptr) { bind(ptr); }
    void bind(SIDLType* ptr) { objPtr=&ptr; }
    protected:
        SIDLType* objPtr;
};
```

Base class for wrapping SIDL objects into Ψ -Mag compliant interfaces. It contains all "wrapping-related" functionality needed by descendant classes.

```
namespace psimagWrap {
template <class Real> class RNG {
public:
    ::psimag::PSIMAGConcept::RNG<Real> >
    {
        public:
            RNG(::psimagConcept::RNG& sidlobj);
            bind(sidlobj);
            void seed(int iseed) { objPtr->seed(iseed); }
            Real evaluate() { return objPtr->evaluate(); }
    };
}
```

Class that wraps the Ψ -Mag `RNG` class. Public methods match those in the native Ψ -Mag class.

Wrapped SIDL objects are used in place of native Ψ -Mag objects transparently. Adapting the SIDL interface to the one expected in native Ψ -Mag code is accomplished without changing the underlying Ψ -Mag code.

Conclusions and Future work

Incorporating template-based generic libraries in a multi-language component environment is possible through the use of a thin wrapping layer that adapts native templated interfaces to SIDL-based interfaces. Wrapped SIDL objects are used in place of native Ψ -Mag objects transparently. Adapting the SIDL interface to the one expected in native Ψ -Mag code is accomplished without changing the underlying Ψ -Mag code. Accommodation for base type template parameters is achieved through the explicit instantiation of all possible interfaces and the use of a factory design pattern.

Future work will focus on the automation of the wrapping process and automatic generation of the interfaces that it might entail. Support for generic array arguments is another area where we need to improve support. This is particularly tricky since the SIDL object model does not currently support generic array types.

References

1. M.H. Austern, *Generic Programming and the STL* (Addison-Wesley, Reading, Mass., 1999).
2. B. Stroustrup, *The C++ Programming Language* (Addison-Wesley, Reading, Mass., 2000).
3. [shhttp://www.ccs.ornl.gov/mri/psimag](http://www.ccs.ornl.gov/mri/psimag)