99

LA-UR- 01- 6950

Approved for public release; distribution is unlimited.

c. 1

Title: AUTOMATED COMPONENT CREATION FOR LEGACY C-PLUS-PLUS AND FORTRAN CODES

Author(s): Matthew J. Sottile
Craig E. Rasmussen

Submitted to: Component Deployment 2002 Conference
Berlin, Germany
June 20-21, 2002

Los Alamos
NATIONAL LABORATORY

Automated Component Creation for Legacy C++ and Fortran Codes

Matthew Sottile and Craig Rasmussen

Los Alamos National Laboratory, Los Alamos NM 87545, USA**

Abstract. A significant amount of work has been spent creating component models and programming environments, but little support exists for automation in the process of creating components from existing codes. To entice users to adopt the component-based paradigm over traditional programming models, integration of legacy codes must be as simple and fast as possible. We present a system for automating the IDL generation stage of component development based on source code analysis of legacy C, C++ and Fortran codes using the Program Database Toolkit. Together with IDL compilation tools such as Babel, we provide an alternative to hand-written IDL code for legacy applications and libraries. In addition to generating IDL, we propose an XML-based method for specifying meta-data related to type mapping and wrapper generation that can be shared between our tools and IDL compilers. The component model of choice for this work is the Common Component Architecture (CCA) using the Scientific Interface Definition Language (SIDL), though the concepts presented can be applied to other models.

1 Introduction

Component architectures provide a rich conceptual model for the construction and maintenance of computer programs similar to more mature paradigms such as object-orientation. It is widely accepted that for a programming model to be adopted by users and application developers, a critical mass of existing codes in the form of object or component libraries must be reached. If it isn't, users of the model are forced to write significantly more code than they would in the older, existing models regardless of the elegance and potential of the new. To achieve this body of ready-to-use code to entice end-users and programmers to use component programming, it must be a reasonable task to "componentize" existing codes. This will achieve two important goals: first, a code base can be rapidly developed within an organization to entice users to adopt new methodologies, and second, legacy codes can be "wrapped" and integrated into modern environments. In this paper we present a tool that uses stable compiler technology to assist in the generation of interface definitions for existing codes, and methods for allowing IDL compilation tools to use this information for wrapper generation.

** This work was supported by the U.S. Department of Energy through Los Alamos National Laboratory Contract W-7405-ENG-36, LA-UR No. XXXXX

Our tool targets a specific component model currently under development within the scientific computing community: the Common Component Architecture (CCA)[1]. The goal of this tool is to automate the generation of interface definitions of existing codes using the Scientific Interface Definition Language (SIDL)[4]. Efforts have been undertaken many times in the past to automate code wrapping by creating parsers for the languages to be wrapped that can understand the interface presented by a code and generating the IDL equivalent. Unfortunately, few have been able to support codes that utilize all aspects of the language specifications. Examples include lack of C++ template support and oversimplified array representations that are either too restrictive or useless to scientific users and those using Fortran 90 in a mixed-language environment. By using the Program Database Toolkit (PDT)[8] for gathering and exploring interface information, we are able to take advantage of PDTs reliance on reliable compiler technology to extract interfaces directly from a standards-compliant front end. The front-ends that are used support each target language that we wish to support: C, C++, Fortran 77 and Fortran 90. The compiler front-end provides sufficient information about the interface of a given code for our tool to explore the entire interface and generate an equivalent IDL interface. Another tool that was developed using PDT for wrapping is SILOON[7], which allows users to automatically generate Python and Perl extensions so that scripts in either language can call the existing code.

Currently our tool does not provide IDL mappings for every type in the C++ or Fortran language specification. To do so would not only be limited by time and programmer-power, but by the lack of essential semantic information in the code itself. Some types cannot be automatically mapped to IDL, so we have chosen at this time to implement an initial version that can use accurate default mappings for many common data types. We are in the process of integrating an XML based mechanism to allow users to customize the behavior of the type mapper so that the SIDL that is generated will require minimal modification by the user. An example of where ambiguity in mapping types appears is with a simple C pointer - in some cases, the pointer is used to represent a single value, where in others the pointer could be referring to an array. Given solely interface information about a piece of code (without data or control flow analysis), we cannot determine how this type is to be interpreted until the user intervenes.

1.1 The Common Component Architecture

Before continuing, we will briefly introduce the goals and purpose of the CCA project. The CCA is a widespread effort within the high-performance computational science community to bring scientific programming models closer to ones found in the general computing industry today. It is similar to other industry component models such as CORBA[10] and JavaBeans[9], but with an emphasis on its use in high performance distributed and parallel environments where complex data structures will be frequently used. Existing component systems can be used in a scientific computing environment to build applications, but since their design requirements were biased toward business or Internet applications, they require additional work to fit well in the high performance computing environment. In addition to meeting scientists performance and usability requirements, the notion of a common component model emphasizes the additional hope that it will encourage a more free exchange of code without imposing implementation restrictions on users as we commonly find in existing scientific libraries. Conceptually, our tools that target the CCA and SIDL are capable of being re-targeted to other component architectures provided a proper implementation of the interface and type mapping modules. Since this is the case, we have designed the tool from the start to be modular so that we can explore this in the future with other popular component architectures. We chose C++ as the implementation language for our tools, which allows porting to new component architectures by using basic object inheritance to create specialized versions of important objects.

2 The Program Database Toolkit and IDL generation

In order to generate an accurate guess at the appropriate IDL mapping for a C++, C or Fortran code, one must have access to the signatures for each function in the interface. A simple script can retrieve this information at a basic level, but it is virtually impossible to derive the interface of complex functions (such as those using templates.) The only place this information is readily available is at the compiler front end after parsing the language into an intermediate language (IL). Instead of reinventing a full, standards-compliant compiler front end to gather this information, we use the Program Database Toolkit (PDT) to gather and access the interface information.

The PDT is based on industrial compiler front ends from the Edison Development Group (C and C++) and Mutek (Fortran 77 and 90). Once the program has been parsed to the intermediate language from the compiler front end, an IL analyzer creates a program database (PDB) that is easily traversed for programs wishing to use this information. A C++ library known as Ductape is used in conjunction with PDT to explore the PDB files using basic container objects and iterators. Ductape forms the layer of our IDL generation tool that explores the program or library interface and passes PDT type objects to our type mapper. The type mapping portion of the tool described later takes the type objects found within a function interface and maps them to the proper IDL type descriptor used later in IDL generation. An illustration of the tool architecture is shown in Figure 1.

Once a PDB file is available for a given legacy code, it is passed into our tools that then use Ductape to walk the data structures and decide what IDL representation is appropriate. To illustrate the process, we assume that we are working with a PDB file for a C++ class library. When the process starts, we determine the scope of the classes within the library, such as the C++ namespaces that the classes reside within. Once we have this information, we explore each class within the namespace. Each public method of the class is passed into a TypeMapper object that we have created to make decisions regarding the IDL equivalent for argument and return types. The result is a set of TypeDescriptor

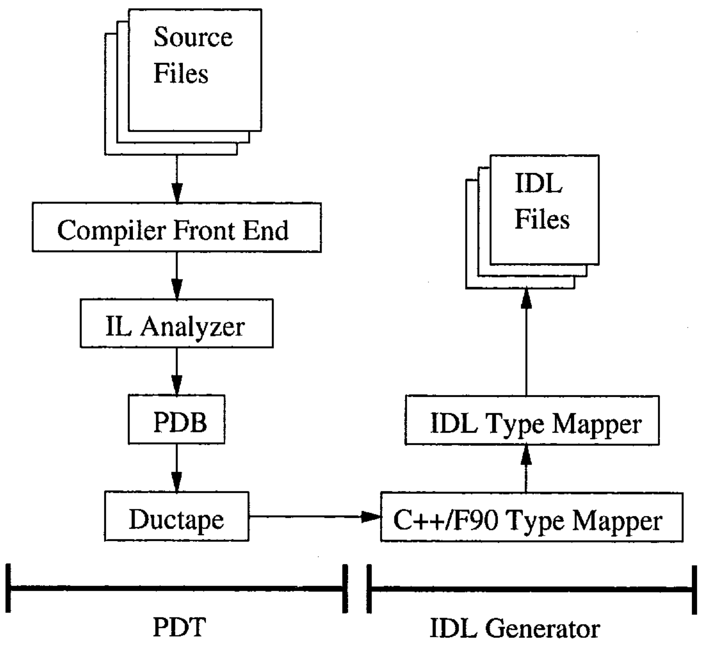

Fig. 1. Architecture of our IDL generator using PDT.

objects that represent the IDL types along with additional fields describing the characteristics of the argument behavior (such as its intent, as explained later.) The TypeMapper objects are the key to accurate IDL generation and are examined in more detail in the next section. The PDT representation in Ductape is language specific: a Fortran method signature is described by different types within Ductape than the equivalent signatures in C++.

3 The TypeMapper Object

Our tools were written in C++, which allows us to use inheritance to specialize classes such as the TypeMapper. The basic TypeMapper object is an abstract class that doesn't provide any functionality, but provides a simple method that takes a PDT type and returns a type descriptor reflecting the equivalent IDL. We do not simply return a type string or other simple structure since an IDL type can provide richer information about an argument beyond basic types. For example, in the CCA SIDL, a type must reflect the type of the argument and its intent - whether it is an "in", "out", or "inout" argument. This information is used in the component runtime system for deciding when and where to copy values, so that unnecessary copies of large data structures can be avoided. For example, a value that is used simply as input should be specified as an "in" argument so that it is not copied back to the caller on each function call. In distributed, remote method invocation situations, this will cut down on wasteful network traffic and memory consumption.

Currently two classes inherit from the TypeMapper interface: the CxxTypeMapper for C++ and the FortranTypeMapper for Fortran. Originally, an older version of PDT was used for building the initial version of our tools, and Fortran support was not complete yet. As a result, the majority of our code and testing has revolved around C++. The current version of the tools include type mappers for Fortran at a basic level, using the mappings described below. An additional type mapper for each language will also be available that can be customized to fit specialized mappings for specific codes. As mentioned earlier, the customizations will be expressed using an XML document that also contains meta-data for classes and methods useful during IDL compilation.

3.1 C++ to SIDL default type mappings

Creating a mapping from C++ to SIDL involved deriving a default mapping for types that do not exist within the SIDL. The most troublesome of these types are references and pointers. In particular, it is difficult to decide if a pointer such as `int *x` refers to a single value (`*x=4`) or an array (`x[2] = 4`) based solely on the interface of a function. Similarly, we cannot tell from an interface alone if a reference argument is modified within the method or not. These ambiguities make mapping arrays and argument intents difficult with limited information about data and control flow.

C++ Signature	*Default Intent*
TYPE *	inout
const TYPE * TYPE const *	in
TYPE * const	inout
const TYPE * const TYPE const * const	in

Table 1. Valid uses of the `const` modifier with C++ pointers and their default intent.

In addition to the trivial mapping of basic types, we provide mappings for the SIDL string type from a character pointer, and handle the different usages of the `const` modifier on arguments. A basic, non-pointer or non-reference argument is automatically an "in" argument. Indirection through a pointer or reference allows data to be modified, so the safest assumption is that data can be passed in and returned via these arguments. Therefore pointers and references with no `const` modifier are mapped to "inout" arguments. If a `const` modifier appears before the type, the language specification states that this argument is therefore treated as an "in" argument regardless of it being a reference, pointer or other mutable type. This interpretation of the `const` modifier is detailed in Table 3.1. The current default type mappings are presented in Table 3.1.

C++ Type (type)	*SIDL Type (stype)*	*Intent*
int	int	in
long	long	in
float	float	in
double	double	in
char	char	in
char *	string	inout
class *	class	inout
type * / type [ ]	array<stype,1>	inout
type $*^n$ / type $[]^n$	array<stype,n>	inout
type &	stype	inout
const type	stype	in

Table 2. Default C++ to SIDL type mappings

3.2 Fortran to SIDL default type mappings

Fortunately, mapping to Fortran is somewhat simpler than C or C++ when using PDT to examine the interfaces. This is due to the presence of an argument's intent in the IL from the Fortran 90 compiler. In the case of C or C++, this had to be inferred based on the type of the data or through user intervention. Currently the Fortran type mapper is less mature than the C++ version, but much of the code we have developed for mapping C++ types can be carried over with little modification to Fortran. As we will also see later, the XML format for specifying custom mappings is designed from the start with both C++ and Fortran in mind. We present the default type mappings for Fortran in the current TypeMapper object in Table 3.2.

Fortran Type (type)	*SIDL Type (stype)*
INTEGER*4	int
INTEGER*8	long
REAL	float
DOUBLE PRECISION	double
CHARACTER*1	char
CHARACTER*(*)	string
type(n)	array<stype,1>
type($d_1, d_2, ...d_n$)	array<stype,n>

Table 3. Default Fortran to SIDL type mappings

3.3 Custom type mappings and code meta-data in XML

In order to fulfill the needs of users and minimize the amount of modification required in the SIDL our tool generates, we have developed a method to allow users to define custom type mappings and other meta-data related to the IDL-generation and wrapping process. Currently, a draft DTD has been created to define the format of these XML files, and we are in the process of creating a TypeMapper class that can be parameterized based on this specification. We are using the Xerces[3] XML library for C++ to validate, parse, and interpret these XML files. The layout of the XML files is intended to mimic the structure of codes in all languages that we wish to support. A `TYPEMAP` tag can be defined within specific other tags to specify custom mappings for the scope in which they are to be applied. For example, the user can define type mappings to be applied in the scope of a library, namespace, module or class. Similarly, for languages (such as C) where scoping may simply be controlled based on nested braces, we allow the user to define these regions with general `SCOPE` tags containing custom mappings.

The XML specification allows users to customize mappings at the function level. A function can be specified by its name and signature, and individual argument types can be given. Additionally, if the user knows exactly what code they want the IDL generator to produce when creating wrapper code, they can place it in a `CODE` tag contained in the function tag. The following sample XML file illustrates the format we have developed.

```
<!DOCTYPE library SYSTEM "mapping.dtd" [
]>
<library name="my_library" lang="cxx">
<scope name="some_cxx_namespace">
<method name="mystatic" sig="void foo(int *x)">
    <arg name="x" intent="out" sig="int *x" sidl="int"></arg>
    <return sig="void" sidl="void"></return>
</method>
<class name="foo_class">
    <method name="foo" sig="void foo(int *x)">
        <arg name="x" intent="inout" sig="int *x" sidl="array int,1"></arg>
        <return sig="void" sidl="void"></return>
        <code>
printf("Hi.\n");
        </code>
    </method>
</class>
</scope>
</library>
```

A final point of interest regarding the XML is that the users are not required to generate the files by hand. We provide a tool which is also based on PDT that creates an XML representation of the interfaces contained within. This can be used as a starting point by the user to customize and pass through our tools and others that know how to interpret the data.

4 Limitations of automatic IDL generation

A fully automatic solution based simply on interface analysis is not possible. The actual meaning of primitive types such as pointers in C or C++ cannot be determined from a basic method signature. Similarly, the intent of an argument cannot be determined for some languages. More sophisticated analysis of the code within a compiler can extract most, if not all, of this information. Data flow analysis to determine the usage of a variable can reveal whether the data is modified or simply read by a method, which will dictate whether the data should have an "in", "out", or "inout" intent. If a variable is a pointer and we see it used as an array, we can assume that it is not simply a singleton value that is pointed to. Unfortunately, this would require a significant effort within the compiler at the IL analysis phase. Furthermore, languages such as C do not prevent programmers from using array syntax to access singleton values through a pointer (though this is considered horrible programming style.) Relying on the user to provide hints that allow a tool to look up the exact meaning of a data type in the context of a method or class will let the tool make the proper decision at the IDL mapping phase.

5 Current status and future directions

5.1 Current status

As has been mentioned above, our target component model is the CCA. Most of our test cases for the SIDL generation were C++ classes that are part of the CCA specification and related component specifications (such as the equation solver interface, ESI[6]). The SIDL for the CCA specification in C++ that was generated by our tool is equivalent to the hand-coded SIDL that has been used previously. The SIDL generated from the ESI headers is not currently accurate due to the lack of template support in our tool. The lack of template support in SIDL requires that the SIDL generator must find a legal SIDL "equivalent" that is sufficient for the users, but not as general as the original template code as we will show below in our discussion of template support in our future work. An example of a header file and the corresponding SIDL generated by our tool is included in Appendix A.

5.2 Creating an automatic component wrapper for legacy codes

In addition to providing a nearly complete mapping of the source language to IDL, we would also like to tie the SIDL generation and PDT knowledge into the IDL compilation process to build complete wrappers very rapidly. The IDL generation process is the first step, where the interfaces are defined in a language independent meta-language. Given these IDL interfaces, an IDL compiler creates code that interfaces with the original code, the client code that wishes to call it, and the support infrastructure (such as Babel[5], ILU[2], CORBA, etc...).

Currently, these steps in the CCA context are not connected. The IDL generator has no knowledge of the actual code that the IDL compiler will generate, and the IDL compiler has no knowledge of the original interface signatures of the code in its native language. This limits the compilers ability to generate the required code to call into the library being wrapped. By providing the information in the IDL generation phase related to the IDL and original signature mappings to the compiler, the compiler can generate the method calls and data conversions required for completing the wrapping process. Additional code can be provided through the `CODE` tags within method descriptions in our XML format and automatically inserted into the code generated by the IDL compiler. We illustrate the wrapper generation process in Figure 2, with the relationship between each step of the process shown.

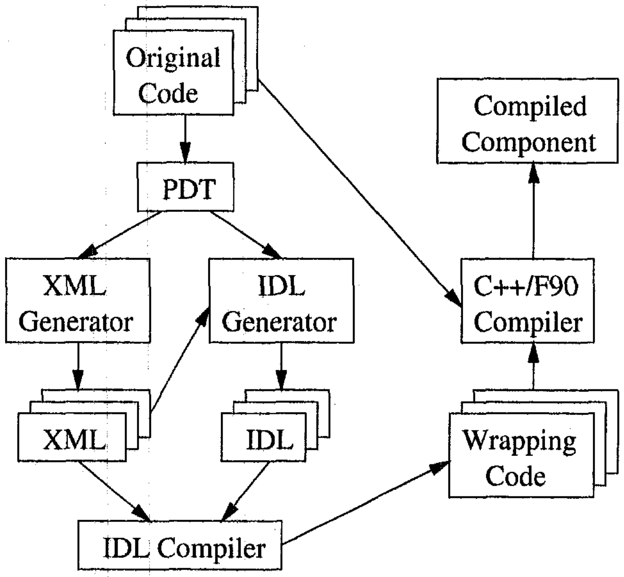

Fig. 2. The parts of the complete component wrapping process.

5.3 Supporting sophisticated types

A further limitation of our current tool is the lack of support for sophisticated types such as C++ templates. SIDL does not currently have a representation of templates, and will likely never incorporate one. To overcome this, we plan to allow users to specify which types they expect a template to be instantiated with. Given these types, we can create SIDL descriptions of the templated objects for each type and instantiate the templates appropriately in the code generated by the IDL compiler. Though it is not the most elegant solution, it will provide a functional solution that is significantly better in the author's opinion than simply not supporting templated types at all. We will add a tag to the XML DTD described above to allow users to define the types that they wish generated.

6 Conclusion

As many have shown, source code analysis opens many possibilities for automated code generation and manipulation. We have shown that by using standards-compliant compiler technology, we gain access to the full code interface and can support many language features not available in other packages. Unfortunately, inferring the exact behavior of code from the interface is not possible without additional information from either the user or more sophisticated analysis of the code control and data flow behavior. The XML format that we have developed will allow users to provide this data regarding the interface behavior, so our tools will be able to create an accurate SIDL representation. Furthermore, the XML will contain enough information to allow IDL compilation tools to generate much of the code required for functional wrappers around existing codes. Our tools have been successful thus far in using PDT to generate a significant amount of code that was previously written by hand.

Acknowledgements. We would like to acknowledge the following people for their assistance and feedback regarding this work. Sameer Shende, Kathleen Lindlan, and Bernd Mohr provided excellent assistance in learning how to use the PDT package. Scott Kohn supplied the required modifications to our C++ type mapper to allow us to wrap large interfaces such as the CCA specification headers. Finally, we would like to thank all members of the CCA Forum for their valuable feedback.

A An example of automatically generated SIDL

The following is a header file defining two basic C++ classes to illustrate how our tool will generate a SIDL file for it. The example shows how inheritance, `const` modifiers, and various uses of pointers and references are dealt with.

```
namespace Test {
class MyParent {
    public:
        int someFunction(char *astring);
};
class SIDL_Tester:MyParent {
    public:
        int min(int a, int b);
        void averager(const int *values, int size, float &average);
        int strlen(char *thestring);
};
};
```

Once the header has been passed through PDT and a PDB file has been generated, our tool is able to produce the following SIDL code.

```
version Test 1.0;
/**
 * C++ package: "Test"
 */
package Test {
  /**
   * C++ class: "Test::MyParent"
   */
  class MyParent {
    /**
     * int Test::MyParent::someFunction(char *)
     */
    final int someFunction(
      inout string astring);
  }
  /**
   * C++ class: "Test::SIDL_Tester"
   */
  class SIDL_Tester
    extends MyParent
 {
    /**
     * int Test::SIDL_Tester::min(int, int)
     */
    final int min(
      in int a,
      in int b);
    /**
     * void Test::SIDL_Tester::averager(const int *, int, float &)
     */
    final void averager(
      in array<int> values,
      in int size,
      inout float average);
    /**
     * int Test::SIDL_Tester::strlen(char *)
     */
    final int strlen(
        inout string thestring);
    }
}
```

References

1. R. Armstrong, D. Gannon, A. Geist, K. Keahey, L. Curfman-McInnes, S. Parker, and B. Smolinski. Toward a common component architecture for high performance scientific computing. In *Proceedings of the Eighth International Symposium on High Performance Distributed Computing*, 1999.
2. Doug Cutting, Bill Janssen, Mike Spreitzer, and Farrell Wymore. *ILU Reference Manual.* Xerox Palo Alto Research Center, 1993.
3. The Apache Software Foundation. *The Apache XML Project.* Available at `http://xml.apache.org/`.
4. Scott Kohn, Tammy Dahlgren, Tom Epperly, and Gary Kumfert. *The State of SIDL: Quarterly Status Report.* Common Component Architecture Forum Meeting, Indiana University, Bloomington, IN. October 2-3, 2001.
5. Scott Kohn, Gary Kumfert, Jeff Painter, and Cal Ribbens. Divorcing language dependencies from a scientific software library. In *Proceedings of the 10th SIAM Conference on Parallel Processing*, 2001.
6. Sandia National Laboratories. *Equation Solver Interface (ESI) Standards.* Available at `http://z.ca.sandia.gov/esi/`.
7. Los Alamos National Laboratory. *SILOON: Scripting Interface Languages for Object-Oriented Numerics.* Available at `http://www.acl.lanl.gov/siloon/`.
8. Kathleen A. Lindlan, Janice Cuny, Allen D. Malony, Sameer Shende, Bernd Mohr, Reid Rivenburgh, and Craig Rasmussen. A tool framework for static and dynamic analysis of object-oriented software with templates. In *Proceedings of SC2000: High Performance Networking and Computing Conference*, 2000.
9. Richard Monson-Haefel. *Enterprise Javabeans.* O'Reilly and Associates, 2000.
10. Alan Pope. *The* CORBA *Reference Guide: Understanding the Common Ob ject Request Broker Architecture.* Addison-Wesley, 1998.