Babel
Users' Guide

Tamara Dahlgren   Thomas Epperly
Gary Kumfert   James Leek

Disclaimer

This work was performed under the auspices of the U.S. Department of Energy by University of California Lawrence Livermore National Laboratory under contract No. W-7405-Eng-48.

Release Information

Babel Users' Guide (this document)	UCRL-SM-205559
Babel Source Code (associated software)	UCRL-CODE-2002-054

Babel
Users' Guide

TAMARA DAHLGREN THOMAS EPPERLY
GARY KUMFERT JAMES LEEK

Center For Applied Scientific Computing
Lawrence Livermore National Laboratory
P.O. Box 808
Livermore, California, USA

July 26, 2005

Preface

This document applies to Babel 0.10.8. It, like the software it documents, is a work in progress.
– The Babel Development Team

Babel in a Nutshell

Babel is a tool that enables software written in different languages to communicate. It accomplishes this task by using an Interface Definition Language (IDL) similar to COM and CORBA. Babel relies on the Scientific Interface Definition Language (SIDL) that is specifically tuned for scientific applications. By expressing software interfaces, or APIs[1], in SIDL the appropriate glue code stubs and skeletons can be generated to facilitate language interoperability. Features unique to SIDL are:

- Dynamic multi-dimensional arrays
- Complex numbers (e.g. $2 + 3i$)
- In-process optimizations
- Special directives for large-scale parallel distributed programming (future)
- Syntax for specifying interface behavior (future)

Babel enables true object-oriented techniques even in non object-oriented languages. The object model that SIDL supports is similar to Java and Objective C where a class can extend at most one class, but implement many interfaces. In C++ speak, an interface is simply a class of all pure-virtual methods. Furthermore, if library developers want object-oriented features but are required to be 100% ANSI C compliant, Babel can meet those constraints. Although the Babel code generator is implemented in Java, the runtime libraries and generated files for C bindings are 100% ANSI C compliant.

Babel can be used as the basis for a component framework, but it is *not* a complete framework by itself. We've added a tiny CCA-compliant framework, called *Decaf*, in our examples/ directory. Decaf demonstrates how Babel can be used to implement a component framework.

SIDL is also a useful communications tool for code development teams since it only expresses the public API. That is, implementation details, which often prove distracting during collaborative design, can be safely avoided by restricting discussions to the interfaces described in SIDL. Furthermore, since SIDL is simple and clean it can be used by Computer Scientists, Math Programmers, and Application Scientists to debate APIs even using only email.

Scope of this Manual

This document is intended as an introduction and tutorial on the use of Babel tools for the generation and use of component software. The Babel tools were designed specifically for scientific applications, therefore most of the examples and exercises here also deal with scientific applications.

This manual assumes the reader is a programmer who is proficient in two or more of the following languages: C, C++, FORTRAN 77, FORTRAN 90, Java, or Python. Furthermore, this manual assumes the reader is familiar with the

[1] Application Programming Interfaces

SPMD[2] programming model that pervades the scientific computing community. Knowledge of and experience with MPI programming is helpful, but not strictly required.

Getting the Software

Babel source is available free of charge on the web. Developed by the Components Project at the Lawrence Livermore National Laboratory Center for Applied Scientific Computing (CASC), it is licensed under the Lesser GNU Public License (LGPL). See the source distribution for details.

The homepage for the Components Project is

http://www.llnl.gov/CASC/components

Conventions

The following typographic conventions are used throughout this manual.

Italic is used for file and command names. It is also used to highlight comments in examples and to define terms the first time they appear in a document.

`Constant Width` is used in examples to show the text that is generated, and in regular text to show operators, variables, and the output from commands or programs.

`Constant Slanted` is used for displaying for SIDL source code. We use a separate font to distinguish SIDL code from generated code.

`Constant Bold` is used to show user's modifications to generated code and in examples to show user's actual input at a terminal.

Sans Serif Slanted is used in examples to show variables for which a context-specific substitution should be made. The variable *filename*, for example, would be replaced by the actual filename.

Additionally, we may use specific blocks of text as sidebars to call the readers attention to particular information. Here's one kind.

> **Rationale:** *Often when listing restrictions or requirements, we find it helpful to also explain and document the rationale behind a design decision. In time, the context in which the rationale was based may become irrelevant, making the rationale blocks very useful for understanding when to change a decision.*

We Appreciate Your Feedback

We have tested and verified the information in this manual. Nonetheless, features may have changed or oversights may exist. Please contact us with any issues, corrections, or suggestions for future versions of this manual through snail mail at:

Components Project
Center for Applied Scientific Computing
Lawrence Livermore National Laboratory
P.O. Box 808, L–365
Livermore, CA 94551

[2] Single Program Multiple Data

or through email to:

`components@llnl.gov`

To find out more about Babel, feel free to subscribe to one or more of the associated distribution lists given below.

- `babel-announce@llnl.gov` is a moderated email forum to which anyone can subscribe (though no-one can post). This is a low-volume alternative for people who want to know about releases and major announcements.
- `babel-dev@llnl.gov` is an open discussion forum about Babel for serious babel users who want to talk about the internal workings of the tools. Anyone can subscribe or send email to this list.
- `babel-users@llnl.gov` is an open discussion forum about Babel for users. Anyone can subscribe or send email to this list.

To subscribe, simply send email to `majordomo@lists.llnl.gov` with the appropriate line(s):

`subscribe babel-announce` *[email-address]*
`subscribe babel-dev` *[email-address]*
`subscribe babel-users` *[email-address]*

where you can explicitly state your email address in *email-address* or, if you leave *email-address* blank, majordomo will use your email ReplyTo: field.

Acknowledgments

Project Alumni: Nathan Dykman, Scott Kohn, and Brent Smolinski

Interns: Melvina Blackgoat, Kirk Kelsey, Sarah Knoop, and Nija Shi

Alpha Testers: Andy Cleary, Jeff Painter, Cal Ribbens

Contributors (Ideas, Bug Reports, Patches, & Code): Rob Armstrong, Ben Allan, Wael Elwasif, Matt Knepley, Boyana Norris, Barry Smith, Jody Winston, and many more.

Sponsors: Babel development originally started as a Strategic Initiative (SI) in the LDRD (Lab Directed R&D) portfolio of Lawrence Livermore National Laboratory.

Current funding is from the DOE/Office of Science SciDAC program as part of the Common Component Technology for Terascale Scientific Simulation (CCTTSS). Also known as the Common Component Architecture.

Software Notices

Babel depends on a great deal of third-party software.

- **JavaCC** is used to generate the SIDL Parser. This is a java.net community project. JavaCC is available under a BSD-style license here: `https://javacc.dev.java.net/`).
- **gnu.getopt** is an implementation of GNU Getopt in Java and is distributed with Babel as a JAR file. It can be downloaded (along with sourcecode) from either the GNU website

  `http://www.gnu.org/software/java/packages.html`

  or the author's website

  `http://www.urbanophile.com/arenn/hacking/download.html.`

The following is the copyright notice for gnu.getopt:

```
/**************************************************************************
/* Getopt.java -- Java port of GNU getopt from glibc 2.0.6
/*
/* Copyright (c) 1987-1997 Free Software Foundation, Inc.
/* Java Port Copyright (c) 1998 by Aaron M. Renn (arenn@urbanophile.com)
/*
/* This program is free software; you can redistribute it and/or modify
/* it under the terms of the GNU Library General Public License as published
/* by  the Free Software Foundation; either version 2 of the License or
/* (at your option) any later version.
/*
/* This program is distributed in the hope that it will be useful, but
/* WITHOUT ANY WARRANTY; without even the implied warranty of
/* MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
/* GNU Library General Public License for more details.
/*
/* You should have received a copy of the GNU Library General Public License
/* along with this program; see the file COPYING.LIB.  If not, write to
/* the Free Software Foundation Inc., 59 Temple Place - Suite 330,
/* Boston, MA  02111-1307 USA
/**************************************************************************/
```

The text for the GNU Library GPL is available at `http://www.gnu.org/copyleft/library.html`.

Contents

Chapter 1

Introduction

Contents

1.1 Babel Facilitates Language Interoperability

Babel was conceived, designed, and built to solve a problem; namely, to make scientific software libraries equally accessible from all of the standard languages. Hence, its goal is language interoperability. The vision goes far beyond calling BLAS[1] implemented in FORTRAN 77 from a C program. At its heart, Babel lets programmers use their tool of choice in developing complete applications using components implemented in one or more distinct programming languages.

For instance, let us say that an application scientist is running a sophisticated C++ code from a Python scripting environment. This can already be easily accomplished with technologies like SWIG. Now let's say that the simulation is showing some erratic behavior and the application scientist wants to extend the `ConvergenceCheck` class to also report some information to a log file. Let's also assume that this application scientist doesn't want to write a new C++ class much less rewrite the current application. What this individual wants to do is derive and utilize a new class in Python from the C++ `ConvergenceCheck` class. Thus, the C++ simulation code will now have to invoke a method on a class implemented in Python, which then dispatches back to the C++ base class after doing its additional logging. This cannot be done in SWIG because SWIG does not support calls from C++ to Python, only from Python to C++. This is an example of a capability that Babel provides that is outside the scope of SWIG.

Figure 1.1 lists many of the primary languages that are of interest to scientific simulation software developers and users. The good news is that there is a path from each language to every other; meaning that calling from one to another is possible. However, the technologies to get from one language to another vary widely, are fraught with pitfalls, and may require calling through a completely different language.

Babel works by providing the technology to define and support the multi-language interoperation of a common subset of functionality through programming language-neutral interface specifications. See Fig. 1.2 to see a graphical representation of the supported languages. It is important to note that this common functionality subset is *far* from a lowest common denominator solution in that Babel actually adds functionality when it is lacking in the host language.

[1] BLAS: Basic Linear Algebra Subroutines

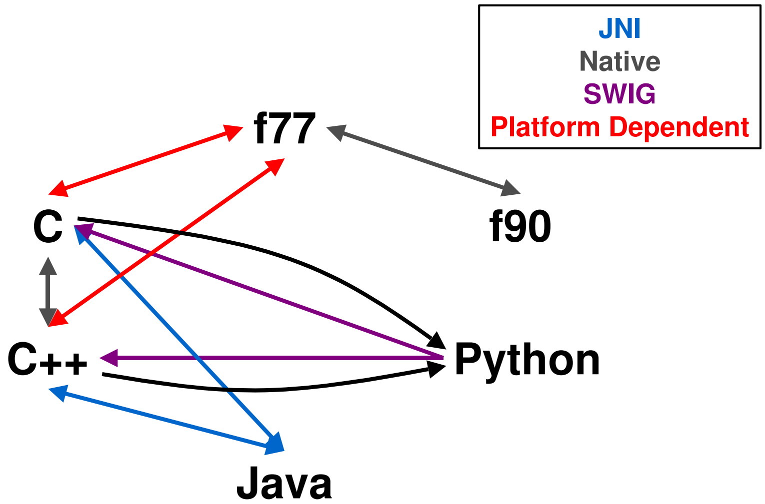

Figure 1.1: Language Interoperability Using Current Technology.

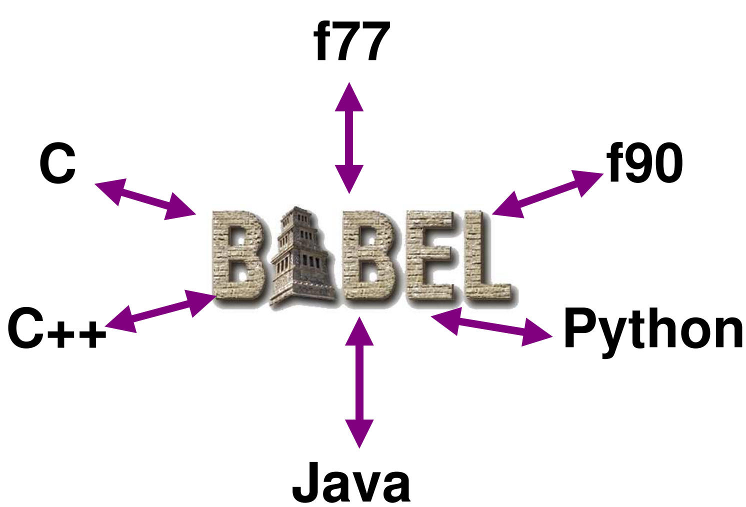

Figure 1.2: Language Interoperability Using Babel.

1.2 Scientific Interface Definition Language (SIDL)

In order to support multi-language interoperability, Babel relies on the specification of interfaces in the Scientific Interface Definition Language (SIDL) (pronounced "SIGH-dull"). SIDL is similar to COM and CORBA IDLs, but was designed with an emphasis on scientific computing. Specifically, SIDL currently supports dynamic multi-dimensional arrays and has built-in complex numbers. It will acquire a set of directives to aid in the description of massively parallel distributed objects and additional syntax for specifying interface behavior.

When it comes to deciding what programming idioms to support across all languages and which ones to reject, SIDL strikes a careful balance between minimalism and completeness. It is *not* a lowest common denominator solution. SIDL is minimal to keep the learning curve as low as possible. It is complete so developers do not feel constrained in how to express their solutions.

SIDL is object-oriented. Its object model closely resembles that of Java and Objective C. In this model there is single inheritance of implementation and multiple inheritance of interfaces. It supports the typical notions of virtual, static, and final methods. SIDL also provides a basic set of features by defining and implementing the basic types for interfaces, classes and exceptions. All types implicitly inherit from these basic types.

The most important concept to grasp about SIDL is that SIDL only defines a public interface that other programs may use to access your code. As a result, all methods defined as part of a SIDL file are public, if you do not want a method to be globally useable, simply do not define it in your SIDL file. Furthermore, all object and class data is implicitly private. There is no way to declare or define data in a SIDL file. Instead, any data required for your code should be declared in the implementation language files. This way, the languages that use your code through Babel may create your objects and pass them around just like any normal piece of data, but they may only access the data through the provided interface.

SIDL also has a complete set of fundamental data types, from booleans to double precision complex numbers. It also supports more sophisticated types such as enumerations, strings, objects, and dynamic multi-dimensional arrays.

SIDL is still a work in progress. Of particular research interest are directives that will be added for parallel distributed object interaction and features to specify behavioral semantics associated with the interfaces.

1.3 Benefits to Customers

Babel has two types of customers: *developer* and *user*. The developer implements a library that will be used by one or more users. Since one goal of the developer is to increase their customer base, the developer writes a SIDL file that effectively publishes the interface to their software in a platform and language neutral manner. The user, on the other hand, may not care or even know that they are interacting with a library through Babel.

Babel provides some features that benefits user and developer alike. The most important aspect to note here is that all Babel objects are reference counted. This feature is critical to encapsulate the memory allocation library (e.g. C's malloc/free or C++'s new/delete) used in the implementation of the object. Users never need concern themselves with when to free up a resource, they only declare when they're done with their reference to that resource. Developers are free to use different memory allocation subsystems in different parts of their code if need be.

1.4 Beyond Babel's Scope

The language interoperability problem is a large one, and though the Babel tools address much of it, there is still a lot that is beyond the scope of our tool. Babel is at its heart a code generator and a runtime library. Consequently, the following features are currently limitations of the Babel tool kit:

Reverse engineering is not supported. That is, there is no support for inspecting or modifying compiled code. In addition, scanning existing software to generate SIDL wrappers is not supported. There are other groups who are pursuing a C++ to SIDL converter. Since SIDL contains different information than what is in a C++ header file, however, such a converter cannot be fully automated without additional help.

Library compatibility is limited. Since Python and Java dynamically load libraries into their virtual machines, using these languages requires the ability to build shared libraries. In general, building shared libraries (particularly from C++) is difficult and error prone. This is compounded by the fact that compiler vendors have no standard way of doing this, and many tools that help building shared libraries don't support C++. One can build a legitimate shared library that still won't work because there are unresolved symbols, or the library was loaded in the wrong mode.

Compiler compatibility is limited. Since the C++ standard does not specify a binary interface and uses a lot of hashing in their symbol tables, there have been no attempts to get libraries from dissimilar C++ compilers to work together. Similarly, although we support FORTRAN 77 and FORTRAN 90, all libraries of Fortran code must be compiled with the same compiler... again because of the lack of a standard binary interface.

Despite the aforementioned limitations, Babel does facilitate the development of language interoperable software. However, issues of robust packaging, building, and deployment of language interoperable software still loom on the horizon.

1.5 Summary

Babel consists of a set of tools that are intended to be used for facilitating language interoperability in the scientific computing community. Using interfaces for libraries or components specified in Scientific Interface Definition Language (SIDL) files, Babel can generate corresponding XML representations as well as the source code for the corresponding stubs, intermediate object representations, and implementation skeletons. The generated source code then becomes the foundation for the glue code that is used for language interoperability between callers of libraries and components.

In addition to providing generated code that automatically handles mapping fundamental data type parameters associated with calls between different languages, Babel has built-in support for complex numbers and multi- dimensional arrays. Additional benefits include object reference counting to facilitate memory management.

Finally, Babel's primary goal is to facilitate the development of language interoperable libraries and components. Hence, support for reverse engineering is not provided. Given that Babel has been developed by a research team, there are also limitations associated with shared library and programming language-specific compiler interoperability support that have been looked into but probably will not be addressed in the foreseeable future. Regardless, Babel has proven to be useful to its stakeholders to the point that it is becoming an integral part of the Common Component Architecture (CCA). Refer to papers and presentations on our web site for more information.

1.6 Organization

The remainder of this document is separated into two parts; namely, foundations and supported language bindings. Part I is devoted to describing the SIDL and the Babel tools. It starts with a tutorial to gently introduce the reader to the development of glue code from both the implementation (or server) and user (or client) sides. The following chapter introduces SIDL and Babel basics. Finally, a chapter on advanced topics, such as linking options, is provided.

Part II describes the language bindings currently supported by Babel. At this point, most of the bindings are programming languages. In which case, most have both client- and server-side bindings. However, Babel also supports textual language backends. At this time, Extensible Markup Language (XML) and Scientific Interface Definition Language (SIDL) are the only to textual backends that are supported.

Appendices are included to provide more information on topics such as acronyms, the SIDL Grammar, and SIDL XML. In addition, sections are included that provide advice and tips on troubleshooting.

Part I

Foundations

Chapter 2

Installation

Ideally, Babel will configure and make "out-of-the-box" on most Unix-like machines. If the configuration process detects that certain resources are unavailable, it will correctly disable support for languages or features needing those resources. If this instance of correct behavior is not the intended behavior, then the installer is left to install the external resources and then re-configure, make, and install Babel. This chapter is intended to provide help and reassurance that Babel is indeed configured and installed correctly.

Contents

2.1 Simple Installation

These instructions assume you have a "tarball" (e.g. *.tar.gz file). We have volunteers who put together and manage RedHat RPMs and Debian *.deb distributions of Babel. If you have one of these distros, read their documentation first as it may have details that supersede our own.

A typical build is a simple sequence of

```
% ./configure
# lots of stuff
...
Fortran77 enabled.
C++ enabled.
Java enabled.
Python enabled.
Fortran90 enabled.
```

```
% make
# lots more stuff
...
% make install
# not so much stuff
...
```

There are many circumstances where the configuration step will properly terminate with an error, but if the configuration works, the build and installation shouldn't terminate abnormally.

2.1.1 Configure

There are two main choices to be made at configure time: "Where does the software get built?" and "Where does the software get installed?". The mechanisms for effecting these choices are quite different.

If you want to build software in a separate directory from where the tarball was untarred, this is called a "VPATH build". VPATH builds are useful if you want to build Babel multiple times with various compilers, flags, or you have a shared filesystem across multiple platforms. It separates the code you generate from things that you were given. The downside is that its more complex to remember where to edit what since original sources will be in the source directory tree and the generated sources and compiled assets will be in the build directory tree.

If you run configure in the directory it appears, (i.e. you typed **`./configure`**) you are performing an "non-VPATH build". To do a VPATH build, simply cd to the directory you want to be the build directory root, then launch configure from there. The following sequence demonstrates a vpath build

```
% tar zxvf babel-x.x.x.tar.gz
% mkdir babel-linux-build
% cd babel-linux-build
% ../babel-x.x.x/configure
```

Note that the directory where you build Babel should be different from the directory where you install Babel. The default install directory is /usr/local, but can be set to any directory that you have read/write access to. To change the install directory, run configure with the **`--prefix`** option. Since many people do not have root access on their machine (or prefer to install in a local directory when dealing with unfamiliar software), this option is probably the second most heavily used option for configure (first being **`--help`**, which is a good one to try also.)

At the time of this writing (0.9.3), there are two configure scripts in Babel, about 40K lines of shell script each. These configure scripts will then propagate the information they acquire to Makefiles by perform approximately 190 sed substitutions (per Makefile), to the source code by setting approximately 170 preprocessor macros in babel_config.h, and various bits of shell script in the build that do not get propagated to the install directory. The configure script does not modify any source code in Babel's runtime system or code generator. This means that source code generated by a different Babel installation is usable as long as it gets compiled against the local babel_config.h and linked with the local Babel runtime libraries.

2.1.2 Make

The makefiles are generated by the configure script from Makefile.in templates. The configure script is generated by a tool called autoconf. The Makefile.in's are generated from Makefile.am files by a separate, but related tool called automake. We also use a tool called libtool to help with libraries. Libtool is written in shell, automake in perl, and autoconf in m4.

After a successful configuration step, if your build fails it is most likely that there is a bug in Babel, autoconf, libtool, or a library of m4 macros from any of the above. It is less likely to be an issue with automake, but possible. Perl and m4 themselves are no longer involved in the process after the configure script is produced, so while there may be a nascent bug in the files they generated, it is unlikely.

2.1.3 Make Check (Optional)

re This is an exhaustive check that can take hours on an average workstation. The number of actual tests run depends on the number of languages that are enabled. In general a driver and an implementation of each test is generated in each enabled language. Then each combination of driver and implementation are run (both statically linked libraries and dynamically loaded libraries, as appropriate) and tested. A test script can actually launch multiple tests, and tests can have multiple parts. At the time of this writing (babel-0.9.3) there are over 13,000 parts tested when all languages are enabled.

2.1.4 Make Install

This transfers built software to the final installation directory. Examples and tests are not installed, nor are Makefiles or dozens of other types of files. Make install also builds javadoc documentation for Babel's code generator. Since some libraries are built with install paths in mind, libtool uses a lot of scripts to make things work in their build directory with binaries actually hidden in .lib subdirectories. Make install strips this extra scaffolding away as well.

2.1.5 Make Installcheck (Optional)

This is the same test suite as with make check. The only difference is that it is run against the code in the install directories, not the build directories.

2.2 External Software Requirements

Babel builds on a lot of available software; some optional, some required. Some we ship in our tarball, some we require users to install separately.

2.2.1 Required & Included

- **Java GetOpt:** This is a Java rewrite of GNU GetOpt available at http://www.urbanophile.com/arenn/hacking/download.html. The Babel code generator uses this to parse command line arguments. The JAR file, download information, and licensing details are in the lib/ subdirectory of the Babel distribution.

- **Xerces-J:** Xerces-J is a Java implementation of SAX and DOM XML parsers available from the Apache Software Foundation at http://www.apache.org. The Babel code generator uses this for XML I/O. The JAR file, download information, and licensing details are in the lib/ subdirectory of the Babel distribution.

2.2.2 Required but Separate

- **Unix shell & bintools:** On early 64bit Linux boxes, we found it necessary to rebuild even these basic tools with all 64bit options enabled. Apparently they were originally installed with less attention to detail than necessary. Bintools includes things like cp and mv.

- **C/C++ compiler:** The Babel runtime library and much of the code generated by the Babel code generator will be ANSI C. So that must be available. The C++ compiler should be optional, but at the time of this writing the configure and makefiles didn't reliably support disabling C++.

- **Java:** The Babel code generator is implemented in Java. One can disable the support for Java language bindings, but a working Java would still be needed for just about everything else. We generally stick with Sun's java developer kits (available at http://java.sun.com). Others have run Babel with Kaffe and GJC.

- **libxml2:** This is the Gnome C library for parsing XML files (see http://xmlsoft.org). The Babel runtime library needs version 2.4 or above to parse SCL files for dynamic loading.

2.2.3 Recommended

- **Python:** Needed for the python language binding (obviously) and for the testing harness. Since the Linux kernel is often configured with a Python-based tool, its hard to find a Linux without python already installed. Python can be downloaded from http://www.python.org.

  One important gotcha is a special case where non-python applications create Babel objects implemented in python. In this case, the Babel runtime needs to dynamically load the python virtual machine (libpython.so). Unfortunately, python does not always build a dynamically loadable version of this library by default. If the Babel configure script cannot find a libpython.so, it will disable server-side Python support.

  At the time of this writing, Python cannot be coerced to build a libpython.so on AIX.

- **Numeric Python (NumPy):** This is a scientific array python extension module. It provides native C arrays (and the ability to manipulate very big arrays) similar to python lists. Babel's python language binding requires this extension module available at http://www.pfdubois.com/numpy.

- **Python Meta Widgets (Pmw):** This is a library of GUI widgets built on top of Python's native tcl/tk interface (tkinter). Its available on SourceForge http://pmw.sourceforge.net Pmw is only needed by the GUI in the babel-life supercomputing demo. This Babel implementation of Conway's Game of Life is a separate tarball found in the contrib/ directory of the Babel distro. There is no test for Pmw in Babel's configuration script.

- **Chasm:** Babel uses the Fortran array descriptor library available in Chasm (see http://chasm-interop.sourceforge.net). Chasm is a language interoperability tool in its own right, but as of version 1.0.1, only the array library is considered complete. Without Chasm, the configuration script will disable Fortran 90 support.

- **pthreads:** Needed for Java language binding.

2.2.4 Optional

These packages are used by Babel maintainers in the course of normal development. You'll need these only if you start rewriting code in Babel's distribution.

- **Automake:** Part of GNU Autotools (see http://www.gnu.org/software/automake). Check the configure.ac file to determine exactly which version we use. The configure script will disable autoconf if it detects the slightest variation from the version we prescribe.

- **Autoconf:** Part of GNU Autotoolssee http://www.gnu.org/software/automake). Check the configure.ac file to determine exactly which version we use. The configure script will disable autoconf if it detects the slightest variation from the version we prescribe.

- **Libtool:** Part of GNU Autotools (see http://www.gnu.org/software/libtool). Note that we often find need to make minor tweeks to ltmain.sh so a fresh download may generate slightly worse results on some platforms.

- **m4:** Contact us for a patched version that we use (we overflow buffers in the distributed version).

- **JavaCC:** This Java Compiler Compiler is what we use to generate the SIDL parser in Babel. If you are interested in experimenting with changing the SIDL grammar, then edit the compiler/gov/llnl/babel/parsers/sidl/sidl.jj file and rebuilt the parser with this tool. Information available at https://javacc.dev.java.net.

- **LaTeX2HTML:** This is used to generate HTML the HTML version of our manuals.

- **perl:** Needed by automake, LaTeX2HTML and other bits and pieces.

Chapter 3

Basic Babel Code Generation

This chapter describes the Babel code generator and its command line options.

Contents

3.1 Babel is a Compiler

Babel is a compiler. It takes symbols and their interfaces as input and generates either code or a given textual representation. These interfaces may be specified in either Scientific Interface Definition Language (SIDL) or Extensible Markup Language (XML). The form the output takes depends upon the options specified on the command line. Refer to the Section 3.2 for details on command line options. More information on the supported bindings can be found in Part II of this document.

3.2 Command Line Options

The entire Babel code generator is written in Java and compiled into a jar file. For convenience, a small script called **`babel`** is provided that *should* set the appropriate environment variables and invoke the Java Virtual Machine on the jar file. To test that the script and jar file are working together properly, simply type **`babel --help`**.

Using Babel

Babel requires exactly one of the following mutually exclusive arguments on the command line unless you use the **`--multi`** option.

- **`--help`** : Print options to stdout.
- **`--version`** : Print version of Babel.
- **`--text=`*`form`*** : Generate text equivalent ("sidl" or "xml") of associated package(s) or generate interface documentation with "html".
- **`--client=`*`lang`*** : Generate client, or proxy, classes to access library.
- **`--server=`*`lang`*** : Generate the server and client classes to implement the library.
- **`--parse-check`** : Check the SIDL file only.

By far, the three most common uses of Babel will be to generate the Client-side proxies, Server-side implementations, and XML associated with the SIDL file. The last option is essentially used internally when the Babel runtime library is being developed.

The **--multi** option lets you generate multiple targets for a given set of files in a single run. Put it first on the command line, each **--client** or **--server** can have a different set of settings.

Additionally, there are a few supplemental arguments that complete the picture.

- **--output-directory=*dir*** : Specifies the root directory associated with the generated files. The default setting is the current working directory.
- **--generate-subdirs** : Generates files in a directory tree matching the packaging scope of the SIDL file. This is on by default for languages that have this requirement, such as Java and Python, but off by default for languages that have no such requirement. Hence, code generation for only the latter languages (e.g. C, C++, F77, F90) is effected by this option.
- **--short-file-name** : When the --generate-subdirs and --short-file-names options are used simultaneously, the generated file names will not include package names, just the class or interface symbol. Thus, either long or short names must be used in all clients or servers that have interdependencies; mixing short and long names will result in compile and/or runtime errors.
- **--repository-path=*path*** : Specifies a semicolon separated list of directories, or URLs[1] to search for XML Type descriptions. The need for these XML types is to resolve references in the SIDL file. This option can be used multiple times on the same command line. If appropriate, the Babel script adds the default repository path to the command line before dispatching to the Java Virtual Machine.
- **--no-default-repository** : Prohibits the use of the default repository in resolving symbols.
- **--suppress-timestamp** : Suppresses the insertion of meta-information that could result in generated files that would otherwise not differ from prior executions on the same, unchanged input file. Typically Babel inserts meta-information such as creation time into files it generates. Although this information is useful, it does result in the creation of excessive changes when using version control systems.
- **--exclude=*regex*** : This options can be used multiple times. Each time you add a regular expression that will be used to exclude symbols from code generation. No code or XML will be generated for any symbol matching the user provided regular expression. This command line option requires version 1.4.0 or later of the Java runtime environment.
- **--comment-local-only** : This option reduces the amount of comments in stub C header files. It will only include the doc comments for locally defined method. It will not include doc comments for inherited methods.
- **--hide-glue** : This option causes all non-impl files to be generated in a `glue/` subdirectory. This reduces the "clutter" in the current directory.
- **--language-subdir** : This options causes all generated files to be stored in a language-dependent subdirectory; if the **--generate-subdirs** option is also used, the language directory will be at the bottom of the hierarchy.
- **--exclude-external** : This option causes code to be generated only for the symbols specified on the command line. No code is generated for symbols on which the users symbols depend.
- **--cxx-ior-exception** : Earlier versions of the Babel C++ bindings checked the IOR pointer in a given stub before making any calls on it. If the IOR was null, a NullIORException was thrown. It was later found that in certain cases these checks were taking an inordinent amount of time, and since C++ does not normally check pointers before dereferencing them, it was decided that this feature was out of line with the spirit of C++. However, since some code had already been written that used this feature, we could not completely eliminate the checks. Therefore, this command line option was added. Calling babel with it will generate C++ stubs with the checks in them. This option has no effect on other languages.

[1] URLs have colons in them, so this path has to be semi-colon separated, even though UNIX paths are traditionally colon separated.

Table 3.1: Command Line Arguments.

SHORT FORM	LONG FORM	NOTES
-h	--help	Print options to stdout.
-v	--version	Print version of Babel.
-t*form*	--text=*form*	Generate text.
-c*lang*	--client=*lang*	Generate client classes.
-s*lang*	--server=*lang*	Generate server and client classes.
-p	--parse-check	Only check parsing of the SIDL file.
	--generate-sidl-stdlib	Regenerate the Babel runtime library.
-o*dir*	--output-directory=*dir*	Root directory to contain generated files.
-g	--generate-subdirs	Generate sources in directory tree matching SIDL packaging.
-R*path*	--output-directory=*path*	Use specified XML repository(ies) to resolve symbols.
-e*regex*	--exclude=*regex*	Do not generate output for matching symbol(s).
	--no-default-repository	Do not use the default repository to resolve symbols.
	--suppress-timestamp	Suppress time-related metadata generation.
	--comment-local-only	Reduce doc comments in C stub header.
-E	--exclude-external	Do not generate code for dependencies.
-u	--hide-glue	Put glue code in a subdirectory.
-l	--language-subdir	Put code in a language dependent directory.
-x	--cxx-ior-exception	Include Null IOR checks in C++ Stubs.
-V	--vpath	Set the impl (splicer block) root directory.

- **`--vpath=`*`dir`*** : This option sets the root directory Babel searches first when trying to load implementation files to preserve splicer block contents in the hand edited implementation files. If you are generating server-side C for a concrete class `x.y.z` and you used **`--vpath=/tmp`**, Babel would try to read splicer blocks from `/tmp/x_y_z_Impl.h` and `/tmp/x_y_z_Impl.c`. If it does not find either file in `/tmp`, it also checks the current directory. If you are using **`--generate-subdirs`** with **`--vpath`**, the vpath directory is the root of the tree, so for the example, Babel would search for `/tmp/x/y/z_Impl.h` and `/tmp/x/y/z_Impl.c`. When appropriate, Babel inserts `#line` directives to refer debuggers to the original file. As its name suggests, this option is useful when making vpath builds using make. Some people also use it to avoid spurious changes to the files managed by their revision control system.

Long and Short Forms

So far, we've shown described the long forms of command line arguments, starting with two hyphens "`--`". There are also short forms for many of the more frequently used commands. See Table 3.1 for details.

Examples

To create a new XML version of a SIDL file, use the following command:

```
% babel -tXML -omydepot mystuff.sidl
```

To exclude code generation for types whose name begins with "MPI.", use the following command:

```
% babel -sC++ --exclude='^MPI\.'  mystuff.sidl
```

Now suppose a developer wants to implement a library in C++ that corresponds to these types in the SIDL file.

```
% babel -sC++ mystuff.sidl
```

Alternatively, the developer could also create C++ implementation files based on the XML repository. In this case, a list of symbols to be implemented would need to be specified. Assuming that all of the types are in a package called "mystuff", the following command can be issued:

```
% babel -sC++ -Rmydepot mystuff
```

Now suppose a second developer wants to extend this software. A second SIDL file is created then the implementation files in FORTRAN 90 are generated with the following command:

```
% babel -sf90 -Rmydepot newstuff.sidl
```

A user now can download both SIDL files and create their Python bindings to use both libraries with the following command:

```
% babel -cPython -Rhttp://localhost/mystuff/mydepot;
 http://www.otherhost.com/newstuff mystuff newstuff
```

Finally, to generate SIDL files for each package based on the XML stored in the repository, the following command is used:

```
% babel -tSIDL -Rhttp://localhost/mystuff/mydepot;
 http://www.otherhost.com/newstuff mystuff newstuff
```

Chapter 4

Hello World Tutorial

Contents

4.1 Introduction

This tutorial guides you through the process of writing the classic "Hello World!" example using the Babel tools. In the process, it attempts to teach you how to write a Scientific Interface Definition Language (SIDL) interface description file, generate the library implementation in C++, and write a C main program to call the library. It also illustrates the process for writing a Makefile to compile and link the library and program.

4.2 Writing the SIDL File

The "Hello World!" program will be written in a directory called hello/ and place the client library in a subdirectory hello/lib/:

```
% mkdir hello
% cd hello
% mkdir lib
```

The first step is to write a SIDL file. Recall that SIDL is an interface definition language (IDL) that describes the calling interface for a scientific library. It is used by the Babel tools to generate glue code that hooks together different programming languages. A complete description of SIDL can be found in Chapter 5.

For this particular application, we will write a SIDL file that contains a class World in a package Hello. Method getMsg() in class World returns a string containing the traditional computer greeting. Using your favorite text editor, create a file called hello.sidl in the hello/ directory containing the following:

```
package Hello version 1.0 {
  class World {
    string getMsg();
  }
}
```

The package statement provides a scope (or namespace) for class World, which contains only one method, getMsg(). The version clause of the statement identifies this as version 1.0 of the Hello package.

4.3 Writing the Implementation

We will write the implementation in the lib/ subdirectory of hello/. The first step is to run the Babel shell script to generate the library implementation code for the SIDL file. We will implement the library in C++. The simplified command to generate the Babel library code (assuming Babel is in your PATH) is [1]:

```
% babel -sC++ -olib ../hello.sidl
```

In this Babel command, the "`-sC++`" flag, or its long form "`--server=C++`", indicates that we wish to generate C++ bindings for an implementation[2]. The "`-olib`" flag, or its long form "`--output-dir=lib`", defines the root directory of where the generated code should be placed.

This command will generate a large number of C and C++ header and source files. It is often surprising to newcomers just how much code is generated by Babel. Rest assured, each file has a purpose and there is a lot of important things being done as efficiently as possible under the hood.

Files are named after the fully-qualified class-name. For instance, a package *`Hello`* and class *`World`* would have a fully qualified name (in SIDL) as *`Hello.World`*. This corresponds to file names beginning with `Hello_World`[3]. For each class, there will be files with `_IOR`, `_skel`, `_stub`, or `_impl` appended after the fully qualified name. *IOR files* are always in ANSI C (source and headers), containing Babel's Intermediate Object Representation. *Impl files* contain the actual implementation, and can be in any language that Babel supports, in this case, they're C++ files. Impl files are the only files that a developer need look at or touch after generating code from the SIDL source. *Skel files* perform translations between the IORs and the Impls. In some cases (like Fortran) the Skels are split into a few files: some in C, some in the Impl language. In the case of C++, the Skels are pure C++ code wrapped in `extern "C"` {} declarations. If the file is neither an IOR, Skel, nor Impl, then it is likely a *Stub*. Stubs are the proxy classes of Babel, performing translations between the caller language and the IOR. Finally, the file babel.make is a Makefile fragment that will simplify writing the Makefile necessary to compile the library. You may ignore the babel.make file if you wish.

The only files that should be modified by the developer (that's you since you're implementing Hello World) are the "Impls", which are in this case files ending with `_Impl.hh` or `_Impl.cc` Babel generates these implementation files as a starting point for developers. These files will contain the implementation of the Hello library. Every implementation file contains many pairs of comment "splicer" lines such as the following:

```
std::string
Hello::World_impl::getMsg()
throw ()
{
    // DO-NOT-DELETE splicer.begin(Hello.World.getMsg)
    // Insert code here...
    // DO-NOT-DELETE splicer.end(Hello.World.getMsg)
}
```

Any modifications between these splicer lines will be saved after subsequent invocations of the Babel tool. Any changes outside the splicer lines will be lost. This splicer feature was developed to make it easy to do incremental development using Babel. By keeping your edits within the splicer blocks, you can add new methods to the hello.sidl file and rerun Babel without the loss of your previous method implementations. You shouldn't ever need to edit the file outside the splicer blocks.

For our hello application, the implementation is trivial. Add the following return statement between the splicer lines in the `lib/Hello_World_Impl.cc` file:

```
std::string
Hello::World_impl::getMsg()
throw ()
{
```

[1] For information on additional command line options, refer to Section 3.2.

[2] You can also try the "`--help`" flag to list all of the Babel command-line options.

[3] Note: dots are converted to underscores for file naming.

```
    // DO-NOT-DELETE splicer.begin(Hello.World.getMsg)
    return std::string("Hello World!");
    // DO-NOT-DELETE splicer.end(Hello.World.getMsg)
}
```

To keep the Makefile simple, we will use some GNU Make features. This Makefile may not work with other make implementations. The GNU gcc and g++ compilers are used in this example. The following Makefile in the lib/ subdirectory will compile the library files and create a shared library named `libhello.so`:

```
# Assumes babel-config is in the current path
.cc.o:
    g++ -fPIC 'babel-config --includes' -c $<
.c.o:
    gcc -fPIC 'babel-config --includes' -c $<

include babel.make
OBJS = ${IMPLSRCS:.cc=.o} ${IORSRCS:.c=.o} \
       ${SKELSRCS:.cc=.o} ${STUBSRCS:.cc=.o}

libhello.so: ${OBJS}
    g++ -shared -o $@ ${OBJS}

clean:
    ${RM} *.o libhello.so
```

You do not necessarily need to create a shared library for this example; you may generate a standard static library (e.g., libhello.a). However, in general, you must generate a shared library if you will be calling your library from Python or Java. To create the shared library archive libhello.so, simply execute make as follows:

```
% cd lib/
% make libhello.so
```

4.4 Writing the Client

We will write the client in the main hello/ subdirectory. The main program will be written in C. File hello.c is as follows:

```
#include <stdio.h>
#include "Hello_World.h"

int main(int argc, char** argv)
{
  Hello_World h = Hello_World__create();
  char* msg = Hello_World_getMsg(h);
  printf("%s\n", msg);
  Hello_World_deleteRef(h);
  free(msg);
}
```

This code creates the Hello_World object, calls the getMsg() method, prints the ubiquitous saying, decrements the reference count for the object, and frees the message string.

There are a few details worth noting here. The C bindings generate function names by combining packages, classes, and method names with underscores (e.g. `Hello_World_getMsg()`. Whenever you see double underscores in Babel generated symbols, they indicate something built-in to (and sometimes specific to) the language binding. The `_create()` method is built-in to every instantiatable class defined in SIDL, triggering the creation of Babel internal data structures as well as the constructor of the actual object implementation.

To generate the C glue code necessary to call the library, we run the Babel tool again, this time specifying C as the target language:

```
% babel --client=C hello.sidl
```

or simply

```
% babel -cC hello.sidl
```

The "-cC" flag, or its equivalent long-form "--client=C", tells the Babel code generator to create only the C stub calling code, not the entire library implementation. The library libhello.so already contains the necessary IOR, skeleton, and implementation object files. We compile the hello program using the following GNU Make Makefile:

```
.c.o:
    gcc `babel-config --includes` -Ilib -c $<

include babel.make
OBJS = hello.o ${STUBSRCS:.c=.o}
LIBDIR=`babel-config --libdir`
hello: ${OBJS}
    gcc ${OBJS} -o $@ \
      -Wl,-rpath -Wl,lib -Llib -lhello \
      -Wl,-rpath -Wl,$(LIBDIR) -L$(LIBDIR) -lsidl

clean:
    ${RM} *.o hello
```

Note that the "-R" flags tell the dynamic library loader where to find the hello and sidl shared libraries. You could achieve the same behavior through environment variables such as LD_LIBRARY_PATH. On some machines and compilers (notably linux-gcc-3.0) the -R flag is no longer supported, so you will have to modify the appropriate environment variable to find the shared library.

Finally, we make the executable and run it:

```
% make hello
% ./hello
Hello World
```

4.5 Final Remarks

Congratulations! You are now ready to develop a parallel scalable linear solver package.

The preceding process may seem to be the most complicated way to write the world's simplest program but, of course, the same process will also work for significantly more complex applications. "Hello World" is small enough to experiment with in the language of your choice. Parallel, multithreaded, scientific simulation codes are another matter entirely.

Chapter 5

SIDL Basics

Contents

5.1 Introduction

This chapter describes the basics of the Scientific Interface Definition Language (SIDL). The goal is to provide sufficient information to enable most library and component developers to begin using SIDL to wrap their software. It begins with an overview of SIDL files followed by an introduction to the fundamental data types. More complex topics such as the object arrays, exceptions, objects, and the XML repository are then addressed.

5.2 SIDL Files

SIDL files are human-readable, language- and platform- independent interface specifications for objects and their methods. SIDL allows you to specify classes, interfaces, and the methods therein. All methods defined in SIDL are public, since the developer is writing them as part of an interface description. Any data you wish a SIDL object to hold is not declared in the SIDL file, and is private. Data should be placed in the implementation skeleton files, and cannot be publicly exported.

Babel reads the SIDL files to generate the appropriate programming language bindings. These bindings, in the form of stub, intermediate object representation (IOR), and implementation skeleton sources, provide the basis for language interoperable software using Babel. In addition, SIDL files are used to populate the XML symbol repository that can serve as an alternate source of interface specifications during the generation of programming language bindings.

Basic Structure

The basic structure of a SIDL file is illustrated below.

```
package <identifier> [version <version>]
{
  interface <identifier> [ <inheritance> ]
  {
```

```
    [<type>] <identifier> ( [<parameters>] ) [throws <exception>];
      .
      .
      .


    [<type>] <identifier> ( [<parameters>] ) [throws <exception>];
  }


  class <identifier> [ <inheritance> ]
  {
    [<type>] <identifier> (<parameters>) [throws <exception>];
      .
      .
      .


    [<type>] <identifier> ( [<parameters>] ) [throws <exception>];
  }


  package <identifier> [version <version>]
  {
      .
      .
      .
  }
}
```

The main elements are *packages*, *interfaces*, *classes*, *methods*, and *types*. For a more detailed description, refer to Appendix B.

Packages provide a mechanism for specifying name space hierarchies. That is, it enables grouping sets of interface and/or class descriptions as well as nested packages. Identified by the *package* keyword, packages have a *scoped* name that consists of one or more identifiers, or name strings, separated by a period (”.”). A package can contain multiple interfaces, classes and nested packages. By default, packages are now re-entrant. In order to make them non-re-entrant, they must be declared as `final`.

Interfaces define a set of methods that a caller can invoke on an object of a class that implements the methods. Multiple inheritance of interfaces is supported, which means an interface or a class can be derived from one or more interfaces.

Classes also define a set of methods that a caller can invoke on an object. A class can extend only one other class but it can implement multiple interfaces. So we have single inheritance of classes and multiple inheritance of interfaces.

Methods define services that are available for invocation by a caller. The signature of the method consists of the return *type*, identifier, arguments, and exceptions. Each parameter has a *type* and a *mode*. The *mode* indicates whether the value of the specified *type* is passed from caller to callee (*in*), from callee to caller (*out*), or both (*inout*). Each exception that a method can *throw* when it detects an error must be listed. These exceptions can be either interfaces or classes so long as they inherit from `sidl.BaseException`. For a default implementation of the exception interfaces, the exception classes should extend `sidl.SIDLException`. Methods and parameter passing modes are discussed in greater detail in Section 5.6.

Types are used to constrain the the values of parameters, exceptions, and return values associated with methods. SIDL supports basic types such as `int`, `bool`, and `long` as well as strings, complex numbers, and arrays.

Comments and Doc-Comments

SIDL has the same commenting style as C++/Java and even has a special documentation comment (so called *doc-comment*) similar to those used in Javadoc. One can embed comments anywhere in their SIDL file. Documentation comments should immediately precede the class, interface, or method with which they are associated. Babel replicates documentation comments in the files it generates. It does not replicate plain comments.

```
/*
 *  1. This is a multi-line comment.
 *
 */

//  2. This comment fits entirely on a single line.

/*  3. This comment can fill less than a line. */

/** 4. This is a documentation comment. */

/**
 *  5. Documentation comments can span
 *     multiple lines without the beginning
 *     space-asterisk-space combinations
 *     getting in the way.
 */
```

Consider the above SIDL file fragment.

1. This comment is a regular multi-line comment that is delimited by a slash-star , star-slash (“`/*`”, “`*/`”) pair.

2. This is a single-line comment that starts with a double slash “`//`” and continues to the end of the line.

3. This comment is the same as # 1 except that it is completely contained on a single line. It can be embedded in the middle of a line anywhere a space naturally occurs.

4. This is a documentation comment. In keeping with Javadoc, Doc++, and other tools, it is delimited by slash-star-star and star-slash (“`/**`”, “`*/`”) combinations. Documentation comments are important because their contents are preserved by Babel in the corresponding generated files. Doc-comments must directly precede the interface, class, or method that they document.

5. This is a multi-line variant of a doc-comment. Note that initial asterisks on a line are assumed to be for human readers only and are discarded by Babel when it reads in the text. The multi-line doc-comment is the preferred way of documenting SIDL.

Packages and Versions

SIDL has both a packaging and versioning mechanism built in. Packages are essentially named scopes, serving a similar function as Java packages or C++ namespaces. Versions are decimal separated integer values where it is assumed larger numbers imply more recent versions. All classes and interfaces in that package get that same version number. If subpackages are specified, they can have their own version number assigned. If a package is declared without a version, it can only contain other packages. If a package declares interfaces or classes, a version number for that package is required.

```
package mypkg {

}
```

This SIDL file represents the minimum needed for each and every SIDL file. The package statement defines a scope where all classes within the package must reside. Since no version clause is included, the version number defaults to 0.

Packages can be nested. This is shown in the example below. The version numbers assigned to all the types is determined by the package, or subpackage, in which it resides. In the design of the SIDL file, remember that some languages get very long function names from excessively nested packages or excessively long package names.

```
package mypkg version 1.0 {

  package thisIsAReallyLongPackageName {
  }

  package this version 0.6 {
    package is {
      package a {
        package really {
          package deeply version 0.4 {
            package nested {
              package packageName version 0.1 {
              }
            }
          }
        }
      }
    }
  }
}
```

External types can be expressed in one of two ways. The fully scoped external type can be used anywhere in the class description. Alternatively, an `import` statement can be used to put the type in the local package-space. `import` statements can request a specific version of the package, if that version is not found, Babel will print an error. If no version is specified, Babel will take whatever version it is being run on. Babel can not be run on two versions of a given package at the same time, even if you only import or require one of them.

Another way to restrict the package version you use is the `restrict` statement. `restrict` does not import the package, but if you do later import the package or refer to something in that package by it's fully scoped name, Babel will guarantee that the correct version of the package will be used. Also note that all restrict statements must come before the first import statement.

Below is a sample SIDL file, that should help bring all of these concepts together.

```
require pkgC version 2.0; // restrict pkgC to version 2.0, not imported

import pkgA version 1.0; // restrict pkgA version 1.0. Includes class pkgA.A

import pkgB;      // import pkgB regaurdless of version.  Includes class pkgB.B

package mypkg version 2.0 {
  class foo {
    setA( A ); // imported from pkgA, must be pkgA.A-v1.0
    setB( B ); // imported from pkgB, must be pkgB.B, no version restriction
    setC( pkgC.C ); // must be pkgC.C-v2.0
    setD( pkgD.D ); // no version restriction
  }
}
```

Re-entrant Packages

By default, SIDL packages are re-entrant. This means that Babel allows sub-packages to be broken into separate files, but you'd still have to run Babel on all the files at the same time. Here's how it works.

Table 5.1: SIDL Types

SIDL TYPE	SIZE (BITS)
bool	1
char	8
int	32
long	64
float	32
double	64
fcomplex	64
dcomplex	128
opaque	64
string	varies
enum	32
interface	varies
class	varies
array<Type,Dim>	varies
rarray<Type,Dim> (*index variables*)	varies

First define the outermost package in a file.

```
package mypkg version 2.0 {

}
```

Then define a sub-package in a second file.

```
package mypkg.subpkg version 2.0 {

}
```

Note that both files begin with the identical version statement. Now as long as you run Babel on both SIDL files at the same time (with the outermost one first on the commandline), all is fine.

This works because the package statement takes a scoped identifier as an argument. As long as Babel knows that a package *mypkg* exists, it can handle a new package called *subpkg*. (This would also work if *subpkg* were a class. Version statements require an identifier for the outermost package. Since packages cannot have dots "." in their names, the only dots in version statements should appear at the numbers, not the package names.

Running the second file without the first will (and should) generate an error since the enclosing package was not declared. Re-entrance should be used judiciously. This feature may be disabled by labeling a given package as *final*.

5.3 Fundamental Types

Table 5.1 briefly shows the different data types that are supported in Babel. Refer to each chapter for the language specific bindings for each SIDL type. The "S" in SIDL stands for "Scientific." This emphasis is reflected in the fundamental support for complex numbers (*fcomplex* and *dcomplex*) and dynamic multidimensional arrays (*array<Type,Dim>*).

C++ developers looking at the SIDL syntax for arrays, might think that SIDL is a templated IDL, but this is not so. Although the syntax for SIDL arrays looks like a template, it is specific only to the array type. Developers cannot create templated classes or methods in SIDL.

> **Rationale:** *Although C++ templates are a very powerful programming mechanism, they apply only to C++. For Babel to implement similar hashing routines, method names in languages other than C++ would become prohibitively (thousands of characters) long. Moreover, this C++ template hashing mechanism is compiler specific so while C++ is very good at hiding the expanded template names (unless there is an error to report) we would have to add babel C++ bindings on a compiler by compiler basis.*

Discussion of the various types is broken up into sections. Numeric types such as *bool*, *char*, *int*, *long*, *float*, *double*, *fcomplex*, *dcomplex*, *string*s, as well as information about enumerated types and the opaque type are all covered in this Subsection 5.3.

Information about extended types such as Interfaces and Classes along with the methods they contain are described in Section 5.6, and Section 5.4 covers Array.

Numeric Types

The SIDL types *bool*, *char*, *int*, *long*, *float*, *double*, *fcomplex*, and *dcomplex* are the smallest and easiest data types to transfer between languages transparently. They all have a fixed size and can just as reasonably be copied as passed by reference.

Most languages natively support all of these data types (though perhaps less so with complex types). There are a few notable exceptions that may be of interest.

ANSI C does not define the size of `int` and `long`, only that the latter be at least as big as the former. As of the C99 standard, there are types `int32_t` and `int64_t` that are signed integers that explicitly support a fixed number of bits. Most compilers already have these symbols defined appropriately in `sys/types.h` (pre C99 standard) or `inttypes.h`.

Python defines its `int` and `long` to be equivalent to C, and therefore suffers the same platform dependent integer size problem with less flexibility for a workaround. It is not uncommon for regression tests involving longs and Python to fail on certain platforms. Python 2.2 has a patch to make SIDL long support better.

Strings

Strings are an interesting datatype because they are fundamental to many pieces of software, but represented differently by practically every single programming language. Strings can have a high overhead to support language interoperability because there is invariably so much copying involved.

FORTRAN 77 and 90 support for strings is limited to a predetermined buffer size. Since the results of a string assignment into that buffer in FORTRAN does not propagate the length of the string, trailing whitespace is always trimmed for any string begin passed out from a FORTRAN implementation.

Opaque

The *opaque* type is dangerous and rarely useful. However, there are particular times when an opaque type is the only way to solve a problem; for example, it is one of the few portable ways to implement an object with state in Fortran 77 (see Section 8.8). When a SIDL file uses an *opaque* type, Babel guarantees only bits will be relayed exactly between caller and callee. If there is a need to pass more information than an opaque provides, than the developer can simply pass a pointer to that information.

Use of a *opaque* carries a heavy penalty. When Babel matures enough to support distributed computing, any method calls with *opaque* in the argument list (or return type) will be restricted to in-process calls only.

> **Rationale:** *Since `opaque` is typically used for a pointer to memory, this sequence of bits has no meaning outside of its own process space.*

Enumerations

An enumeration is typically used in programming languages to specify a limited range of states to enable dealing with them by names instead of hard-coded values. For language interoperability purposes — especially to support this concept on languages with no native support — we've had to create specific rules for the integer values associated with enumerated types.

```
package enumSample version 1.0 {

  // undefined integer values
  enum color {
    red, orange, yellow, green, blue, violet
  };
```

```
    // completely defined integer values
    enum car {
      /**
       * A sports car.
       */
      porsche = 911,
      /**
       * A family car.
       */
      ford = 150,
      /**
       * A luxury car.
       */
      mercedes = 550
    };

    // partially defined integer value
    enum number {
      notZero,// This non-doc comment will not be retained.
      notOne,
      zero=0,
      one=1,
      negOne=-1,
      notNeg
    };
  }
```

Above is a sample of enumerations taken directly from our regression tests. It defines a package `enumSample` that contains three enumerations. C/C++ developers will find the syntax very familiar. When defining an enumeration, the actual integer values assigned can be undefined, completely defined, or partially defined.

SIDL defines the following rules for adding integer values to enumerated states that don't have a value explicitly defined.

1. Error if two states are explicitly assigned the same value
2. Assign all explicit values to their named state.
3. Assign smallest unused non-negative value to first unassigned state in enumeration.
4. Repeat 3 until all states have assigned (unique) values.

To verify the application of these rules, the `enumSample.number` enumeration will have the following values assigned to its states: `NotZero`=2, `NotOne`=3, `zero`=0; `one`=1, `negOne`=-1, `notNeg`=4.

5.4 Arrays

One of the features that separates SIDL and BABEL from Microsoft's COM/DCOM and the OMG's CORBA is support for multi-dimensional arrays. SIDL is designed to serve the high performance computing community, so we anticipate that both SIDL object developers and object clients may require direct access to the underlying array data structure to try to optimize instruction pipelining or cache performance. The purpose of this document is to describe the functional API to the SIDL array data structure and the underlying data structures. This presentation will focus on the C API for arrays because it is the basis for the other language APIs, so they will likely reflect its idiosyncrasies.

There are two main kinds of arrays in SIDL: normal arrays and r-arrays. R-arrays are a specialized form of array for numeric types that has a simpler interface from C, C++, FORTRAN 77 and FOTRAN 90. Normal arrays are used for all SIDL types.

The SIDL array API and data structure can be used in client code to prepare argument for passing to a SIDL method, and it is used inside the implementation code to get data and meta-data from incoming arguments.

Normal SIDL arrays can be "row-major" or "column-major". They are not parallel array classes, and not particularly sophisticated, but they are very, very general. These are meant to generalize the array types built into many languages, not to provide a general array component that everyone will use. It is expected for parallel array libraries to build on top of the array type presented into SIDL.

R-arrays

There are two kinds of SIDL arrays: normal SIDL arrays and raw SIDL arrays called r-arrays. Normal SIDL arrays provide all the features of a normal SIDL type. They can be passed as `in`, `inout`, or `out` parameters, and they can be returned as a method return value. Normal SIDL arrays can be allocated or borrowed, and they are reference counted. You can also pass `NULL` as a normal SIDL array.

SIDL r-arrays exist to provide a lower access to numeric arrays from C, C++, Fortran 77, Fortran 90 and future languages as appropriate. For example, a one-dimensional r-array in C appears as a double pointer and a length parameter. To highlight the contrast, normal SIDL arrays appear as a struct in C, a template class in C++, an 64-bit integer in Fortran 77 and a derived type in Fortran 90.

R-arrays are have more restrictions in how they can be used. Here is how r-arrays are more constrainted:

1. Only the `in` and `inout` parameter modes are available for r-arrays. R-arrays cannot be used as return values or as `out` parameters.

2. R-arrays must be contiguous in memory, and multi-dimensional arrays must be in column-major order (i.e., Fortran order).

3. `NULL` is not an allowable value for an r-array parameter.

4. The semantics for `inout` r-array parameters are different. The implementation is not allowed to deallocate the array and return a new r-array. `inout` means that the array data is transferred from caller to callee at the start of a method invocation and from callee to caller at the end of the a method invocation.

5. The implementation of a method taking an r-array parameter cannot change the shape of the array.

6. The lower index is always 0, and the upper index is $n-1$ where n is the length in a particular dimension. This is contrary to the normal convention for Fortran arrays.

7. It can only be used for arrays of SIDL `int`, `long`, `float`, `double`, `fcomplex`, and `dcomplex` types.

Rationale: *The way r-arrays are passed to the server-side code, particularly Fortran 77, makes it impossible for them to be allocated or deallocated. This makes* `out` *and return values impossible. Because the data has to be accessible directly from Fortran 77 without any additional meta-data, the array data must be in column-major order.*

Arrays of char are not currently supported for r-arrays because in some languages characters are treated as 16-bit Unicode characters.

The advantages of r-arrays include:

- Arrays appear more "natural" in C, C++, Fortran 77, Fortran 90 and future low level languages.
- Developers need less or no code to tranlate between their array data structure and SIDL's array data structure.
- SIDL generated APIs can have signatures very similar if not identical to well known legacy APIs.
- Less performance overhead because r-arrays can avoid a call to `malloc` and `free`.

When you declare an r-array, you also declare the index variables that will hold the size of the array in each dimension. For example, here is an method to solve one of the fundamental problems of linear algebra, $Ax = b$:

```
void solve(in    rarray<double,2> A(m,n),
           inout rarray<double>   x(n),
           in    rarray<double>   b(m),
           in    int              m,
           in    int              n);
```

In this example, `A` is a 2-D array of doubles with `m` rows and `n` columns. `x` is a 1-D array of doubles of length `n`, and `b` is a 1-D array of doubles of length `m`. Note that by explicitly declaring the index variables, SIDL takes avoid using extra array size parameters by taking advantage of the fact that the sizes of `A`, `x` and `b` are all inter-related. The explicit declaration also allows the developer to control where the index parameters appear in the argument list. In many cases, the argument types and order can match existing APIs.

The mapping for the solve method will be shown for C, C++, Fortran 77 and Fortran 90 in the following chapters. In languages that do not support low level access such as Python and Java, r-arrays are treated just like normal SIDL arrays, and the redundant index arguments are dropped from the argument list. The indexing information is available from the SIDL array data structure.

SIDL Language Features

As of release 0.6.5, interface definitions can specify that an array argument or return value must have a particular ordering for a method. The type `array<int, 2, row-major>` indicates a dense, [1] two-dimensional array of 32 bit integers in row-major order; and likewise, the type `array<int, 2, column-major>` indicates an dense array in column-major order. Some numerical routines can only provide high performance with a particular type of array. The ordering is part of the interface definition to give clients the information they need to use the underlying code efficiently. The ordering specification is optional.

For one-dimensional arrays, specifying `row-major` or `column-major` allows you to specify that the array must be dense, that is stride 1. Otherwise, for one-dimensional arrays row-major and column-major are identical.

If you pass an array into a method and the array does not have the specified ordering, the skeleton code will make a copy of the array with the required ordering and pass the copy to the method. This copying is necessary for correctness, but it will cause a decrease in performance. The implementor of the method can count on an incoming array to have the required ordering.

For `out` parameters and return values, an ordering specification means that the method promises to return an array with the specified ordering. The implementation should create the `out` arrays with the proper ordering; because if it does not, the skeleton code will have to copy the outgoing array into a new array with the required ordering.

For `inout` parameters, an ordering specification means the ordering specification will be enforced by the skeleton code for the incoming and outgoing array value.

At the time of writing this, the ordering constraints are enforced for Python implementation because Python uses Numeric Python arrays, so BABEL cannot control the array ordering as fully. The Python skeletons do force outgoing arrays (i.e., arrays passed back from Python) to have the required ordering.

Independent and borrowed arrays

From a memory perspective, there are two main kinds of arrays: independent and borrowed. The independent arrays owns and manages its data. It allocates space for the array elements when the array is created, and it deallocates that space when the array is finally destroyed.

The borrowed array does not own or manage its data. It borrows its array element data from another source that it cannot manage, and it only allocates space for the index bounds and stride information. The rationale for borrowed arrays is to allow data from another source to temporarily appear as a SIDL array without requiring data be copied.

If you `slice` an independent array, the resulting array is also considered independent even though it borrows data from the original independent array. The resulting array can still manage its data by retaining a reference to the original array; hence, its element data cannot disappear until the resulting array is destroyed. If you `slice` a borrowed array, the resulting array is also borrowed because like its original array, it doesn't manage the underlying data.

In the Babel generated code, r-arrays are converted to borrowed arrays. These borrowed arrays are allocated on the stack rather than on the heap to improve performance of r-arrays.

The Life of an Array

The existence of borrowed arrays causes the arrays to deviate from the normal reference counting pattern. You may recall that all arrays are reference counted, and an array's resources are reclaimed when the reference count goes to zero. However, a borrowed array's array element data will disappear whenever the source of the borrowed data

[1] meaning non-strided

Table 5.2: SIDL types to array function prefixes

SIDL TYPE	ARRAY FUNCTION PREFIX	VALUE TYPE
bool	sidl_bool	sidl_bool
char	sidl_char	char
dcomplex	sidl_dcomplex	struct sidl_dcomplex
double	sidl_double	double
fcomplex	sidl_fcomplex	struct sidl_fcomplex
float	sidl_float	float
int	sidl_int	int32_t
long	sidl_long	int64_t
opaque	sidl_opaque	void *
string	sidl_string	char *

determines that it should regardless of the reference count in corresponding the SIDL array. This behavior means that developers should consider any SIDL array that they did not create themselves, for example incoming arguments to methods, as potential borrowed arrays. When a method wants to keep a copy of an array that might be a borrowed array, it should use the `smartCopy` method documented below.

Here are some rules of thumb about the use of borrowed arrays:

- The creator of a borrowed array should guarantee that the data for the borrowed array will exist through the duration of any method calls using the borrowed array.
- Methods should not return a borrowed array as a return value or `out` parameter unless the method can guarantee that the array element data will be available until the process shuts down.
- There is a negligible performance cost when using `smartCopy` when the array is not borrowed, and there is a huge correctness benefit when the array is borrowed.

The Language Bindings

The C++ binding for array provides access to the C API in case you need to take the gloves off and revel in the data directly. But the C++ binding also provides a templated wrapper class to provide a more natural look and feel for C++ programmers.

The Python binding for arrays involves copying SIDL arrays to/from Numeric Python arrays. Arrays in Python don't have the SIDL methods available. They just have the Numeric Python API available.

The FORTRAN 77 API mimics the C API; all the C functions have been FORTRANified and have _f appended to their names. The FORTRAN 90 API uses function overloading to allow programmers to use the short array method names.

The Array API

In the following presentation, we use the SIDL *double* type; however, everything in this section applies to all types except where noted. The basic types are in the SIDL namespace. Table 5.2 shows the prefix for SIDL base types and the actual value type held by the array...

For arrays of interfaces or classes, the name of the array function prefix is derived from the fully qualified type name. For example, for the type *sidl.BaseClass*, the array functions all begin with `sidl_BaseClass`. For *sidl.BaseInterface*, they all begin with `sidl_BaseInterface`.

When you add an object or interface to an array, the reference count of the element being overwritten is decremented, and the reference count of the element being added is incremented. When you get an object or interface from an array, the caller owns the returned reference.

For arrays of strings when you add a string to any array, the array will store a copy of the string. When you retrieve a string from an array, you will receive a copy of the string. You should `sidl_String_free` the returned string when you are done with it.

When you create an array of interfaces, classes, or strings, all elements of the array are initialized to NULL. Other arrays are not initialized. When an array of interfaces, classes, or strings is destroyed, it releases any held references in the case of objects or interfaces. In the case of strings, it frees any non-NULL pointers.

The name of the data structure that holds the array if double is `struct sidl_double__array`. For some types, the data structure is an opaque type, and for others, it is defined in a public C header file.

The functions are listed succinctly in Table 5.3 as well as in detail over the next few pages.

Function: createCol

```
/* C */
struct sidl_double__array*
sidl_double__array_createCol(int32_t       dimen,
                             const int32_t lower[],
                             const int32_t upper[]);
//
// C++
static sidl::array<double>
sidl::array<double>::createCol(int32_t       dimen,
                               const int32_t lower[],
                               const int32_t upper[]);
C
C FORTRAN 77
       subroutine sidl_double__array_createCol_f(dimen, lower, upper, result)
       integer*4 dimen
       integer*4 lower(dimen), upper(dimen)
       integer*8 result
!
! FORTRAN 90
subroutine createCol(lower, upper, result)
  integer (selected_int_kind(9)), dimension(:), intent(in) :: lower, upper
  type (sidl_double_3d), intent(out) :: result ! type depends on dimension
! dimension of result is inferred from the size of lower

// Java
// (isRow should be false to get a column order array)
  public Array(int dim, int[] lower, int[] upper, boolean isRow);
```

This method creates a column-major, multi-dimensional array in a contiguous block of memory. `dimen` should be strictly greater than zero, and `lower` and `upper` should have `dimen` elements. `lower[i]` must be less than or equal to `upper[i]-1` for $i \geq 0$ and $i < dimen$. If this function fails for some reason, it returns `NULL`. `lower[i]` specifies the smallest valid index for dimension `i`, and `upper[i]` specifies the largest. Note this definition is somewhat un-C like where the upper bound is often one past the end. In SIDL, the size of dimension `i` is `1 + upper[i] - lower[i]`.

The function makes copies of the information provided by `dimen`, `lower`, and `upper`, so the caller is not obliged to maintain those values after the function call.

For FORTRAN, the new array is returned in the last parameter, `result`. A zero value in `result` indicates that the operation failed. For Fortran 90, you can use the function `not_null` to verify that `result` is a valid array.

Function: createRow

```
/* C */
struct sidl_double__array*
sidl_double__array_createRow(int32_t       dimen,
                             const int32_t lower[],
                             const int32_t upper[]);
//
// C++
static sidl::array<double>
```

Table 5.3: SIDL Array Functions

SHORT NAME	DESCRIPTION
createCol	Creates a column-major order SIDL array
createRow	Creates a row-major order SIDL array
create1d	Creates a dense one-dimensional SIDL array
create2dCol	Creates a dense, column-major, two-dimensional SIDL array
create2dRow	Creates a dense, column-major, two-dimensional SIDL array
slice	Creates a sub-array of another array. Takes parameters to define array properties.
borrow	Makes a SIDL array from thrid party data without copying it
smartCopy	Copies a borrowed array or addRefs a non-borrowed array
addRef	Increments the reference count.
deleteRef	Decrements the reference count.
get1	Returns the indexed element from a one-dimensional array
get2	Returns the indexed element from a two-dimensional array
get3	Returns the indexed element from a three-dimensional array
get4	Returns the indexed element from a four-dimensional array
get5	Returns the indexed element from a five-dimensional array
get6	Returns the indexed element from a six-dimensional array
get7	Returns the indexed element from a seven-dimensional array
get	Returns the indexed element from an array of any dimension
set1	Sets the indexed element in a one-dimensional array
set2	Sets the indexed element in a two-dimensional array
set3	Sets the indexed element in a three-dimensional array
set4	Sets the indexed element in a four-dimensional array
set5	Sets the indexed element in a five-dimensional array
set6	Sets the indexed element in a six-dimensional array
set7	Sets the indexed element in a seven-dimensional array
set	Sets the indexed element in an array of any dimension
dimen	Returns the dimension of the array
lower	Returns the lower bound of the specified dimension
upper	Returns the upper bound of the specified dimension
stride	Returns the stride of the specified dimension
length	Returns the length of the Array in the specified dimension
isColumnOrder	Returns true if the array is a dense column-major order array, false otherwise
isRowOrder	Returns true if the array is a dense row-major order array, false otherwise
copy	Copies the contents of source array to dest array
ensure	Returns an array with guaranteed ordering and dimension from any array.
first	Provides direct access to the element data of the array.

```
sidl::array<double>::createRow(int32_t       dimen,
                               const int32_t lower[],
                               const int32_t upper[]);
C
C FORTRAN 77
       subroutine sidl_double__array_createRow_f(dimen, lower, upper, result)
       integer*4 dimen
       integer*4 lower(dimen), upper(dimen)
       integer*8 result
!
! FORTRAN 90
subroutine createRow(lower, upper, result)
  integer (selected_int_kind(9)), dimension(:), intent(in) :: lower, upper
  type(sidl_double_3d), intent(out) :: result ! type depends on dimension
! dimension of result is inferred from the size of lower

// Java
// (isRow should be true to get a row order array)
  public Array(int dim, int[] lower, int[] upper, boolean isRow);
```

This method creates a row-major, multi-dimensional array in a contiguous block of memory. Other than the difference in the ordering of the array elements, this method is identical to `createCol`.

Function: create1d

```
/* C */
struct sidl_double__array*
sidl_double__array_create1d(int32_t len);

// C++
static sidl::array<double>
sidl::array<double>::create1d(int32_t len);

C FORTRAN 77
       subroutine sidl_double__array_create1d_f(len, result)
       integer*4 len
       integer*8 result

! FORTRAN 90
subroutine create1d(len, result)
  integer (selected_int_kind(9)), intent(in) :: len
  type(sidl_double_1d), intent(out) :: result

// Java
  public Array1(int s0, boolean isRow);
```

This method creates a dense, one-dimensional vector of ints with a lower index of 0 and an upper index of $len-1$. This is defined primarily as a convenience for C and C++ programmers. If $len \leq 0$, this routine returns NULL.

Function: create2dCol

```
/* C */
struct sidl_double__array*
sidl_double__array_create2dCol(int32_t m, int32_t n);

// C++
static sidl::array<double>
sidl::array<double>::create2dCol(int32_t m, int32_t n);
```

```
C FORTRAN 77
       subroutine sidl_double__array_create2dCol_f(m, n, result)
       integer*4 m, n
       integer*8 result

! FORTRAN 90
subroutine create2dCol(m, n, result)
  integer (selected_int_kind(9)), intent(in) :: m, n
  type(sidl_double_2d), intent(out) :: result

// Java
// isRow should be false to get a column order array
  public Array2(int s0, int s1, boolean isRow);
```

This method creates a dense, column-major, two-dimensional array of ints with a lower index of $(0, 0)$ and an upper index of $(m-1, n-1)$. If $m \leq 0$ or $n \leq 0$, this method returns NULL. This is defined primarily as a convenience for C and C++ programmers.

Function: create2dRow

```
/* C */
struct sidl_double__array*
sidl_double__array_create2dRow(int32_t m, int32_t n);

// C++
static sidl::array<double>
sidl::array<double>::create2dRow(int32_t m, int32_t n);

C FORTRAN 77
       subroutine sidl_double__array_create2dRow_f(m, n, result)
       integer*4 m, n
       integer*8 result

! FORTRAN 90
subroutine create2dRow(m, n, result)
  integer (selected_int_kind(9)), intent(in) :: m, n
  type(sidl_double_2d), intent(out) :: result

// Java
// isRow should be false to get a column order array
  public Array2(int s0, int s1, boolean isRow);
```

This method creates a dense, row-major, two-dimensional array of ints with a lower index of $(0, 0)$ and an upper index of $(m-1, n-1)$. If $m \leq 0$ or $n \leq 0$, this method returns NULL. This is defined primarily as a convenience for C and C++ programmers.

Function: slice

```
/* C */
struct sidl_double__array *
sidl_double__array_slice(struct sidl_double__array *src,
                         int32_t                    dimen,
                         const int32_t              numElem[],
                         const int32_t             *srcStart,
                         const int32_t             *srcStride,
                         const int32_t             *newStart);
//
// C++
array<double>
```

```
sidl::array<double>::slice(int dimen,
                           const int32_t numElem[],
                           const int32_t *srcStart = 0,
                           const int32_t *srcStride = 0,
                           const int32_t *newStart = 0);
C
C FORTRAN 77
       subroutine sidl_double__array_slice_f(src, dimen, numElem, srcStart,
      $                        srcStride, newStart, result)
       integer*8 src, result
       integer*4 dimen
       integer*4 numElem(srcDimen), srcStart(srcDimen)
       integer*4 srcStride(srcDimen),  newStart(dimen)
!
! FORTRAN 90
subroutine slice(src, dimen, numElem, srcStart, srcStride, newStart, result)
  type(sidl_double_3d), intent(in) :: src     ! type depends on dimension
  type(sidl_double_2d), intent(out) :: result ! type depends on dimension
  integer (selected_int_kind(9)), intent(in) :: dimen
  integer (selected_int_kind(9)), intent(in), dimension(:) :: &
     numElem, srcStart, srcStride, newStart

// Java
  public native Array _slice(int dimen, int[] numElem, int[] srcStart,
     int[] srcStride, int[] newStart);
```

This method will create a sub-array of another array. The resulting array shares data with the original array. The new array can be of the same dimension or potentially less than the original array. If you are removing a dimension, indicate the dimensions to remove by setting `numElem[i]` to zero for any dimension `i` that should go away in the new array. The meaning of each argument is covered below.

src the array to be created will be a subset of this array. If this argument is NULL, NULL will be returned. The returned array borrows data from `src`, so modifying one array modifies both. In C++, the `this` pointer takes the place of `src`.

dimen this argument must be greater than zero and less than or equal to the dimension of `src`. An illegal value will cause a NULL return value.

numElem this specifies how many elements from src should be in the new array in each dimension. A zero entry indicates that the dimension should not appear in the new array. This argument should be an array with an entry for each dimension of `src`. `NULL` will be returned for `src` if either

$$\begin{aligned} \texttt{srcStart[i] + numElem[i] * srcStride[i]} &> \texttt{upper[i]}, \text{ or} \\ \texttt{srcStart[i] + numElem[i] * srcStride[i]} &< \texttt{lower[i]} \end{aligned}$$

srcStart this parameter specifies which element of `src` will be the first element of the new array. If this argument is NULL, the first element of `src` will be the first element of the new array. If non-NULL, this argument provides the coordinates of an element of `src`, so it must have an entry for each dimension of `src`. `NULL` will be returned for `src` if either

`srcStart[i] < lower[i]`, or `srcStart[i] > upper[i]`.

srcStride this argument lets you specify the stride between elements of `src` for each dimension. For example with a stride of 2, you could create a sub-array with only the odd or even elements of `src`. If this argument is NULL, the stride is taken to be one in each dimension. If non-NULL, this argument should be an array with an entry for each dimension of `src`.

newLower this argument is like the `lower` argument in a create method. It sets the coordinates for the first element in the new array. If this argument is NULL, the values indicated by `srcStart` will be used. If non-NULL, this should be an array with `dimen` elements.

Assuming the method is successful and the return value is named newArray, `src[srcStart]` refers to the same underlying element as `newArray[newStart]`.

If `src` is not a borrowed array (i.e., it manages its own data), the returned array can manage its by keeping a reference to `src`. It is not considered a borrowed array for purposes of `smartCopy`.

Function: borrow

```
/* C */
struct sidl_double__array*
sidl_double__array_borrow(double*       firstElement,
                          int32_t       dimen,
                          const int32_t lower[],
                          const int32_t upper[],
                          const int32_t stride[]);
//
// C++
void
sidl::array<double>::borrow(double*       firstElement,
                            int32_t       dimen,
                            const int32_t lower[],
                            const int32_t upper[],
                            const int32_t stride[]);
C
C FORTRAN 77
       subroutine sidl_double__array_borrow_f(firstElement, dimen, lower,
     $      upper, stride, result)
       real*8 firstElement()
       integer*4 dimen, lower(dimen), upper(dimen), stride(dimen)
       integer*8 result
!
! FORTRAN 90
subroutine borrow(firstElement, dimen, lower, upper, stride, &
                                    result)
  real (selected_real_kind(17,308)), intent(in) :: firstElement
  integer (selected_int_kind(9)), intent(in) :: dimen
  integer (selected_int_kind(9)), dimension(:), intent(in) :: lower, upper,&
                                                 stride
  type(sidl_double_1d), intent(out) :: result ! type depends on array dimension
```

This method creates a proxy SIDL multi-dimensional array using data provided by a third party. In some cases, this routine can be used to avoid making a copy of the array data. `dimen`, `lower`, and `upper` have the same meaning and constraints as in `SIDL_double__array_createCol`. The `firstElement` argument should be a pointer to the first element of the array; in this context, the first element is the one whose index is `lower`.

`stride[i]` specifies the signed offset from one element in dimension `i` to the next element in dimension `i`. For a one dimensional array, the first element has the address `firstElement`, the second element has the address `firstElement + stride[0]`, the third element has the address `firstElement + 2 * stride[0]`, etc. The algorithm for determining the address of the element in a multi-dimensional array whose index is in array `ind[]` is as follows:

```
int32_t* addr = firstElement;
for(int i = 0; i < dimen; ++i) {
  addr += (ind[i] - lower[i])*stride[i];
}
/* now addr is the address of element ind */
```

Note elements of stride need not be positive.

The function makes copies of the information provided by `dimen`, `lower`, `upper`, and `stride`. The type of `firstElement` is changed depending on the array value type (see Table 5.2).

Function: smartCopy

```
/* C */
struct sidl_double__array*
sidl_double__array_smartCopy(struct sidl_double__array *array);

// C++
void
sidl::array<double>::smartCopy();

C FORTRAN 77
       subroutine sidl_double__array_smartCopy_f(array, result)
       integer*8 array, result

! FORTRAN 90
subroutine smartCopy(array, result)
  type(sidl_double_1d), intent(in) :: array   ! type depends on dimension
  type(sidl_double_1d), intent(out) :: result ! type depends on dimension

// Java
  public native Array _smartCopy();
```

This method will copy a borrowed array or increment the reference count of an array that is able to manage its own data. This method is useful when you want to keep a copy of an incoming array. The C++ method operates on `this`.

Function: addRef

```
/* C */
void
sidl_double__array_addRef(struct sidl_double__array* array);

// C++
void
sidl::array<double>::addRef() throw ( NullIORException );

C FORTRAN 77
       subroutine sidl_double__array_addRef_f(array)
       integer*8 array

! FORTRAN 90
subroutine addRef(array)
  type(sidl_double_1d), intent(in) :: array ! type depends on array dimension
```

This increments the reference count by one. In C++, this method should be avoided because the C++ wrapper class manages the reference count for you.

Function: deleteRef

```
/* C */
void
sidl_double__array_deleteRef(struct sidl_double__array* array);

// C++
void
sidl::array<double>::deleteRef() throw ( NullIORException );
```

```
C FORTRAN 77
       subroutine sidl_double__array_deleteRef_f(array)
       integer*8 array

! FORTRAN 90
subroutine deleteRef(array)
  type(sidl_double_1d), intent(out) :: array ! type depends on dimension
```

This decreases the reference count by one. If this reduces the reference count to zero, the resources associated with the array are reclaimed. In C++, this method should be avoided because the C++ wrapper class manages the reference count for you.

Function: get1

```
/* C */
double
sidl_double__array_get1(const struct sidl_double__array* array,
                        int32_t                          i1);

// C++
double
sidl::array<double>::get(int32_t i1);

C FORTRAN 77
       subroutine sidl_double__array_get1_f(array, i1, result)
       integer*8 array
       integer*4 i1
       real*8 result

! FORTRAN 90
subroutine get(array, i1, result)
  type(sidl_int_1d), intent(in) :: array
  integer (selected_int_kind(9)), intent(in) :: i1
  real (selected_real_kind(17,308)), intent(out) :: result

// Java
  public double get(int i);
```

This method returns the element with index `i1` for a one dimensional array. The return type of this method is the value type for the SIDL type being held (see Table 5.2). This method must only be called for one dimensional arrays. For objects and interfaces, the client owns the returned reference (i.e., the client is obliged to call `deleteRef()` when they are done with the reference unless it is `NULL`). For arrays of strings, the client owns the returned string (i.e., the client is obliged to call free on the returned pointer unless it is `NULL`). There is no reliable way to determine from the return value cases when i1 is out of bounds.

Function: get2

```
/* C */
double
sidl_double__array_get2(const struct sidl_double__array* array,
                        int32_t                       i1,
                        int32_t                       i2);

// C++
double
sidl::array<double>::get(int32_t i1, int32_t i2);
```

```
C FORTRAN 77
       subroutine sidl_int__array_get2_f(array, i1, i2, result)
       integer*8 array
       integer*4 i1, i2
       real*8 result

! FORTRAN 90
subroutine get(array, i1, i2, result)
  type(sidl_int_2d), intent(in) :: array
  integer (selected_int_kind(9)), intent(in) :: i1, i2
  real (selected_real_kind(17,308)), intent(out) :: result

// Java
  public double get(int i, int j);
```

This method returns the element with indices (`i1`, `i2`) for a two dimensional array. The return type of this method is the value type for the SIDL type being held (see Table 5.2. This method must only be called for two dimensional arrays. For objects and interfaces, the client owns the returned reference (i.e., the client is obliged to call `deleteRef` when they are done with the reference unless it is `NULL`). For arrays of strings, the client owns the returned string (i.e., the client is obliged to call free on the returned pointer unless it is `NULL`). There is no reliable way to determine from the return value cases when `i1`, `i2` are out of bounds.

Function: get3

```
/* C */
double
sidl_double__array_get3(const struct sidl_double__array* array,
                        int32_t                          i1,
                        int32_t                          i2,
                        int32_t                          i3);
// C++
double
sidl::array<double>::get(int32_t i1, int32_t i2, int32_t i3);

C FORTRAN 77
       subroutine sidl_double__array_get3_f(array, i1, i2, i3, result)
       integer*8 array
       integer*4 i1, i2, i3
       real*8 result

! FORTRAN 90
subroutine get(array, i1, i2, i3, result)
  type(sidl_double_3d), intent(in) :: array
  integer (selected_int_kind(9)), intent(in) :: i1, i2, i3
  real (selected_real_kind(17,308)), intent(out) :: result

// Java
  public double get(int i, int j, int k);
```

This method returns the element with indices (`i1`, `i2`, `i3`) for a three dimensional array. The return type of this method is the value type for the SIDL type being held (see Table 5.2). This method must only be called for three dimensional arrays. For objects and interfaces, the client owns the returned reference (i.e., the client is obliged to call `deleteRef()` when they are done with the reference unless it is `NULL`). For arrays of strings, the client owns the returned string (i.e., the client is obliged to call `free()` on the returned pointer unless it is `NULL`). There is no reliable way to determine from the return value cases when i1, i2, i3 are out of bounds.

Function: get4

```
/* C */
double
sidl_double__array_get4(const struct sidl_double__array* array,
                     int32_t                       i1,
                     int32_t                       i2,
                     int32_t                       i3,
                     int32_t                       i4);
// C++
double
sidl::array<double>::get(int32_t i1, int32_t i2, int32_t i3, int32_t i4);

C FORTRAN 77
       subroutine sidl_double__array_get4_f(array, i1, i2, i3, i4, result)
       integer*8 array
       integer*4 i1, i2, i3, i4
       real*8 result

! FORTRAN 90
subroutine get(array, i1, i2, i3, i4, result)
  type(sidl_double_4d), intent(in) :: array
  integer (selected_int_kind(9)), intent(in) :: i1, i2, i3, i4
  real (selected_real_kind(17,308)), intent(out) :: result

// Java
  public double get(int i, int j, int k, int l);
```

This method returns the element with indices(`i1`, `i2`, `i3`, `i4`) for a four dimensional array. The return type of this method is the value type for the SIDL type being held (see Table 5.2). This method must only be called for four dimensional arrays. For objects and interfaces, the client owns the returned reference (i.e., the client is obliged to call `deleteRef()` when they are done with the reference unless it is `NULL`). For arrays of strings, the client owns the returned string (i.e., the client is obliged to call `free()` on the returned pointer unless it is `NULL`). There is no reliable way to determine from the return value cases when `i1`, `i2`, `i3`, or `i4` are out of bounds.

Function: get5-7

Methods `get5`–`get7` are defined in an analogous way.

Function: get

```
/* C */
double
sidl_double__array_get(const struct sidl_double__array* array,
                    const int32_t                 indices[]);

// C++
double
sidl::array<double>::get(const int32_t indices[]);

C FORTRAN 77
       subroutine sidl_double__array_get_f(array, indices, result)
       integer*8 array
       integer*4 indices()
       real*8 result

! FORTRAN 90
subroutine get(array, indices, result)
  type(sidl_real_1d), intent(in) :: array ! type depends on dimension
  integer (selected_int_kind(9)), dimension(:), intent(in) ::indices
```

```
    real (selected_real_kind(17,308)), intent(out) :: result

  // Java
    public native double _get(int i, int j, int k, int l, int m, int n, int o);
```

This method returns the element whose index is indices for an array of any dimension. The return type of this method is the value type for the SIDL type being held (see Table 5.2). This method can be called for any positively dimensioned array. For objects and interfaces, the client owns the returned reference (i.e., the client is obliged to call `deleteRef()` when they are done with the reference unless it is `NULL`). For arrays of strings, the client owns the returned string (i.e., the client is obliged to call `free()` on the returned pointer unless it is `NULL`). There is no reliable way to determine from the return value cases when indices has an element out of bounds.

Function: set1

```
/* C */
void
sidl_double__array_set1(struct sidl_double__array* array,
                        int32_t                    i1,
                        double                     value));

// C++
void
sidl::array<int32_t>::set(int32_t i1, double value);

C FORTRAN 77
       subroutine sidl_double__array_set1_f(array, i1, value)
       integer*8 array
       integer*4 i1
       real*8 value

! FORTRAN 90
subroutine set(array, i1, value)
  type(sidl_double_1d), intent(in) :: array
  integer (selected_int_kind(9)), intent(in) :: i1,
  real (selected_real_kind(17,308)), intent(in) :: value

// Java
  public void set(int i, double value) {
```

This method sets the value in index `i1` of a one dimensional array to value. The type of the argument value is the value type for the SIDL type being held (see Table 5.2). This method must only be called for one dimensional arrays. For arrays of objects and interfaces, the array will make its own reference by calling `addRef()` on value, so the client retains its reference to value. For arrays of strings, the array will make a copy of the string, so the client retains ownership of the value pointer.

Function: set2

```
/* C */
void
sidl_double__array_set2(struct sidl_double__array* array,
                        int32_t                    i1,
                        int32_t                    i2,
                        double                     value));

// C++
void
sidl::array<double>::set(int32_t i1, int32_t i2, double value);
```

```
C FORTRAN 77
       subroutine sidl_double__array_set2_f(array, i1, i2, value)
       integer*8 array
       integer*4 i1, i2
       real*8 value

! FORTRAN 90
subroutine set(array, i1, i2, value)
  type(sidl_int_2d), intent(in) :: array
  integer (selected_int_kind(9)), intent(in) :: i1, i2
  real (selected_real_kind(17,308)), intent(in) :: value

// Java
  public void set(int i, int j, double value) {
```

This method sets the value in index (`i1`, `i2`) of a two dimensional array to value. The type of the argument value is the value type for the SIDL type being held (see table 5.2). This method must only be called for two dimensional arrays. For arrays of objects and interfaces, the array will make its own reference by calling `addRef()` on value, so the client retains its reference to value. For arrays of strings, the array will make a copy of the string, so the client retains ownership of the value pointer.

Function: set3

```
/* C */
void
sidl_double__array_set3(struct sidl_double__array* array,
                        int32_t                    i1,
                        int32_t                    i2,
                        int32_t                    i3,
                        double                     value));

// C++
void
sidl::array<double>::set(int32_t i1, int32_t i2, int32_t i3, double value);

C FORTRAN 77
       subroutine sidl_double__array_set3_f(array, i1, i2, i3, value)
       integer*8 array
       integer*4 i1, i2, i3
       real*8 value

! FORTRAN 90
subroutine set(array, i1, i2, i3, value)
  type(sidl_double_3d), intent(in) :: array
  integer (selected_int_kind(9)), intent(in) :: i1, i2, i3
  real (selected_real_kind(17,308)), intent(in) :: value

// Java
  public void set(int i, int j, int k, double value) {
```

This method sets the value in index (`i1`, `i2`, `i3`) of a three dimensional array to value. The type of the argument value is the value type for the SIDL type being held (see table 5.2). This method must only be called for three dimensional arrays. For arrays of objects and interfaces, the array will make its own reference by calling `addRef()` on value, so the client retains its reference to value. For arrays of strings, the array will make a copy of the string, so the client retains ownership of the value pointer.

Function: set4

```
/* C */
void
sidl_double__array_set4(struct sidl_double__array* array,
                        int32_t                    i1,
                        int32_t                    i2,
                        int32_t                    i3,
                        int32_t                    i4,
                        double                     value));
//
// C++
void
sidl::array<double>::set(int32_t i1, int32_t i2,
                         int32_t i3, int32_t i4, double value);
C
C FORTRAN 77
       subroutine sidl_double__array_set4_f(array, i1, i2, i3, i4, value)
       integer*8 array
       integer*4 i1, i2, i3, i4
       real*8 value
!
! FORTRAN 90
subroutine set(array, i1, i2, i3, i4, value)
  type(sidl_double_4d), intent(in) :: array
  integer (selected_int_kind(9)), intent(in) :: i1, i2, i3, i4
  real (selected_real_kind(17,308)), intent(in) :: value

// Java
  public void set(int i, int j, int k, int l, double value) {
```

This method sets the value in index (`i1`, `i2`, `i3`, `i4`) of a four dimensional array to value. The type of the argument value is the value type for the SIDL type being held (see table 5.2). This method must only be called for four dimensional arrays. For arrays of objects and interfaces, the array will make its own reference by calling `addRef()` on value, so the client retains its reference to value. For arrays of strings, the array will make a copy of the string, so the client retains ownership of the value pointer.

Function: set5-7

Methods `set5`–`set7` are defined in an analogous way.

Function: set

```
/* C */
void
sidl_double__array_set(struct sidl_double__array* array,
                       const int32_t              indices[],
                       double                     value);

// C++
void
sidl::array<double>::set(const int32_t indices[], double value);

C FORTRAN 77
       subroutine sidl_double__array_set_f(array, indices, value)
       integer*8 array
       integer*4 indices()
       real*8 value

! FORTRAN 90
subroutine set(array, indices, value)
```

```
  type(sidl_double_1d), intent(in) :: array ! type depends on dimension
  integer (selected_int_kind(9)), intent(in), dimension(:) :: indices
  real (selected_real_kind(17,308)), intent(in) :: value

// Java
  public native void _set(int i, int j, int k, int l, int m, int n,
                          int o, double value);
```

This method sets the value in index indices for an array of any dimension to value. The type of the argument value is the value type for the SIDL type being held (see table 5.2). For arrays of objects and interfaces, the array will make its own reference by calling `addRef()` on value, so the client retains its reference to value. For arrays of strings, the array will make a copy of the string, so the client retains ownership of the value pointer.

Function: dimen

```
/* C */
int32_t
sidl_double__array_dimen(const struct sidl_double__array *array);

// C++
int32_t
sidl::array<double>::dimen() const;

C FORTRAN 77
       subroutine sidl_double__array_dimen_f(array, result)
       integer*8 array
       integer*4 result

! FORTRAN 90
integer (selected_int_kind(9)) dimen(array)
  type(sidl_double_1d) :: array ! type depends on dimension

// Java
public native int _dim();
```

This method returns the dimension of the array.

Function: lower

```
/* C */
int32_t
sidl_double__array_lower(const struct sidl_double__array *array, int32_t ind);

// C++
int32_t
sidl::array<double>::lower(int32_t ind) const;

C FORTRAN 77
       subroutine sidl_double__array_lower_f(array, ind, result)
       integer*8 array
       integer*4 ind, result

! FORTRAN 90
integer (selected_int_kind(9)) function lower(array, ind)
  type(sidl_double_1d), intent(in) :: array ! type depends on dimension
  integer (selected_int_kind(9)) :: ind

// Java
  public native int _lower(int dim);
```

This method returns the lower bound on the index for dimension `ind` of array.

Function: upper

```
/* C */
int32_t
sidl_double__array_upper(const struct sidl_double__array *array, int32_t ind);

// C++
int32_t
sidl::array<double>::upper(int32_t ind) const;

C FORTRAN 77
       subroutine sidl_double__array_upper_f(array, ind, result)
       integer*8 array
       integer*4 ind, result

! FORTRAN 90
integer (selected_int_kind(9)) function upper(array, ind)
  type(sidl_double_1d), intent(in) :: array ! type depends on dimension
  integer (selected_int_kind(9)), intent(in) :: ind

// Java
  public native int _upper(int dim);
```

This method returns the upper bound on the index for dimension ind of array. If the upper bound is greater than or equal to the lower bound, the upper bound is a valid index (i.e., it is not one past the end).

Function: stride

```
/* C */
int32_t
sidl_double__array_stride(const struct sidl_double__array *array, int32_t ind);

// C++
int32_t
sidl::array<double>::stride(int32_t ind) const;

C FORTRAN 77
       subroutine sidl_double__array_stride_f(array, ind, result)
       integer*8 array
       integer*4 ind, result

! FORTRAN 90
integer (selected_int_kind(9)) function stride(array, ind)
  type(sidl_double_1d), intent(in) :: array ! type depends on dimension
  integer (selected_int_kind(9)) :: ind

// Java
  public native int _stride(int dim);
```

This method returns the stride for a particular dimension. This stride indicates how much to add to a pointer to get for the current element this the particular dimension to the next.

Function: length

```
/* C */
int32_t
sidl_double__array_length(const struct sidl_double__array *array, int32_t ind);

// C++ Default dimension is 1.
```

```
int32_t
sidl::array<int32_t>::length(int32_t ind = 0) const;

C FORTRAN 77
       subroutine sidl_double__array_length_f(array, ind, result)
       integer*8 array
       integer*4 ind, result

! FORTRAN 90
integer (selected_int_kind(9)) function length(array, ind)
  type(sidl_double_1d), intent(in) :: array ! type depends on dimension
  integer (selected_int_kind(9)) :: ind

// Java
  public native int _length(int dim);

// For one dimensional Java arrays. Array1:
  public int lenth();
```

This method returns the length for a particular dimension. It is equivalent to the statement `upper(dim) - lower(dim) + 1`.

There is also a shortcut for one-dimensional arrays available in C++ and Java. In C++, if `length` is called with no arguments, it defaults to the first dimension. In Java `Array1` one-dimensional Java arrays have a length function that takes no arguments.

Function: isColumnOrder

```
/* C */
sidl_bool
sidl_double__array_isColumnOrder(const struct sidl_double__array *array);

// C++
bool
sidl::array<double>::isColumnOrder() const;

C FORTRAN 77
       subroutine sidl_double__array_isColumnOrder_f(array, result)
       integer*8 array
       logical   result

! FORTRAN 90
logical function isColumnOrder(array)
  type(sidl_double_2d), intent(in) :: array ! type depends on dimension

// Java
  public native boolean _isColumnOrder();
```

This method returns a true value if and only if `array` is dense, column-major ordered array. It does not modify the array at all.

Function: isRowOrder

```
/* C */
sidl_bool
sidl_double__array_isRowOrder(const struct sidl_double__array *array);

// C++
bool
sidl::array<double>::isRowOrder() const;
```

```
C FORTRAN 77
       subroutine sidl_double__array_isRowOrder_f(array, result)
       integer*8 array
       logical   result

! FORTRAN 90
logical function isRowOrder(array)
  type(sidl_double_1d), intent(int) :: array ! type depends on dimension

// Java
  public native boolean _isRowOrder();
```

This method returns a true value if and only if `array` is dense, row-major ordered array. It does not modify the array at all.

Function: copy

```
/* C */
void
sidl_double__array_copy(const struct sidl_double__array *src
                              struct sidl_double__array *dest);

// C++
void
sidl::array<double>::copy(const sidl::array<double> &src);

C FORTRAN 77
       subroutine sidl_double__array_copy_f(array, dest)
       integer*8 array, dest

! FORTRAN 90
subroutine copy(array, dest)
  type(sidl_double_1d), intent(in) :: array ! type depends on array dimension
  type(sidl_double_1d), intent(in) :: dest  ! type depends on array dimension

// Java
  public void _copy(sidl.Double.Array dest);
```

This method copies the contents of `src` to `dest`. For the copy to take place, both arrays must exist and be of the same dimension. This method will not modify `dest`'s size, index bounds, or stride; only the array element values of `dest` may be changed by this function. No part of `src` is changed by this method.

If `dest` has different index bounds than `src`, this method only copies the elements where the two arrays overlap. If `dest` and `src` have no indices in common, nothing is copied. For example, if `src` is a 1-d array with elements 0-5 and dest is a 1-d array with element 2-3, this function will copy element 2 and 3 from `src` to `dest`. If `dest` had elements 4-10, this method could copy elements 4 and 5.

Function: ensure

```
/* C */
struct sidl_double__array *
sidl_double__array_ensure(const struct sidl_double__array *src,
                          int32_t                          dimen,
                          int                              ordering);

// C++
void
sidl::array<double>::ensure(int32_t dimen, int ordering);
```

```
C FORTRAN 77
       subroutine sidl_double__array_ensure_f(src, dimen, ordering, result)
       integer*8 src, result
       integer*4 dimen, ordering

! FORTRAN 90
subroutine ensure(src, dimen, ordering, result)
   type(sidl_double_1d), intent(in) :: src    ! type depends on array dimension
   type(sidl_double_1d), intent(out) :: result! type depends on array dimension
   integer (selected_int_kind(9)) :: dimen, ordering
```

This method is used to obtain a matrix with a guaranteed ordering and dimension from an array with uncertain properties. If the incoming array has the required ordering and dimension, its reference count is incremented, and it is returned. If it doesn't, a copy with the correct ordering is created and returned. In either case, the caller knows that the returned matrix (if not NULL) has the desired properties.

This method is used internally to enforce the array ordering constraints in SIDL. Clients can use it in similar ways. However, because the method was intended as an internal Babel feature, is not available in Java or Python.

The ordering parameter should be one of the constants defined in `enum sidl_array_ordering` (e.g. `sidl_general_order`, `sidl_column_major_order`, or `sidl_row_major_order`). If you pass in `sidl_general_order`, this routine will only check the dimension of the matrix.

Function: first

```
/* C */
double *
sidl_double__array_first(const struct sidl_double__array *src);

// C++
double* first() throw();

C FORTRAN 77
      subroutine sidl_double__array_access_f(array, ref, lower, upper,
     $   stride, index)
      integer*8 array
      integer*4 lower(), upper(), stride(), index
      integer*4 ref()
```

This method provides direct access to the element data. Using this pointer and the stride information, you can perform your own array accesses without function calls. This method isn't available for arrays of strings, interface and objects because of memory/reference management issues. There is no equivalent of this of this function in Java or Python. To see how to get direct array access in FORTRAN 90, see Chapter 9.

The FORTRAN versions of the method return the lower, upper and stride information in three arrays, each with enough elements to hold an entry for each dimension of `array`. Because FORTRAN 77 does not have pointers, you must pass in a reference array, `array`. Upon exit, `ref(index)` is the first element of the array. The type of `ref` depends on the type of the array.

WARNING: While calling the FORTRAN direct access routines, there is a possibility of an alignment error between your reference pointer, `ref`, and the pointer to the first element of the array data. The problem is more likely with arrays of `double` or `dcomplex`; although, it could occur with any type on some future platform. If `index` is zero on return, an alignment error occurred. If an alignment error occurs, you may be able to solve it by recompiling your FORTRAN files with flags to force doubles to be aligned on 8 byte boundaries. For example, the `-malign-double` flag for g77 forces doubles to be aligned on 64-bit boundaries. An alignment error occurs when `(char *)ref` minus `(char *)sidl_double__array_first(array)` is not integer divisible by `sizeof(datatype)` where `ref` refers to the address of the reference array.

Here is an example FORTRAN 77 subroutine to output each element of a 1-dimensional array of doubles using the direct access routine. FORTRAN 90 has a pointer in the array derived type when direct access is possible.

```
C This subroutine will print each element of an array of doubles
      subroutine print_array(dblarray)
      implicit none
      integer*8 dblarray
      real*8 refarray(1)
      integer*4 lower(1), upper(1), stride(1), index, dimen, i
      if (dblarray .ne. 0) then
         call sidl_double__array_dimen_f(dblarray, dimen)
         if (dimen .eq. 1) then
            call sidl_double__array_access_f(dblarray, refarray,
     $           lower, upper, stride, index)
            if (index .ne. 0) then
               do i = lower(1), upper(1)
                  write(*,*) refarray(index + (i-lower(1))*stride(1))
               enddo
            else
               write(*,*) 'Alignment error occured'
            endif
         endif
      endif
      end
```

For a 2-dimensional array, the loop and array access is

```
      do i = lower(1), upper(1)
         do j = lower(2), upper(2)
            write(*,*) refarray(index+(i-lower(1))*stride(1)+
     $           (j - lower(2))*stride(2))
         enddo
      enddo
```

Suppose you are wrapping a legacy FORTRAN application and you need to pass a SIDL array to a FORTRAN subroutine. Further suppose there is a FORTRAN 77 and FORTRAN 90 version of the subroutine. For example, the FORTRAN 77 subroutine has a signature such as:

```
      subroutine TriedAndTrue(x, n)
      integer n
      real*8 x(n)
C insert wonderful, efficient, debugged code here
      end
```

The FORTRAN 90 subroutine has basically the same signature as follows:

```
subroutine TriedAndTrue(x, n)
  integer (selected_int_kind(9))     :: n
  real (selected_real_kind(17, 308)) :: x(n)

  ! insert wonderful, efficient, debugged code here
end subroutine TriedAndTrue
```

Here is one way to wrap this method using SIDL. First of all, the SIDL method definition specifies that the array must be a 1-dimensional, column-major ordered array. This forces the incoming array to be a dense column.

```
    static void TriedAndTrue(inout array<double,1,column-major> arg);
```

Given that method definition in a class named Class and a package named Pkg, the implementation of the wrapper should look something like the following for FORTRAN 77:

```
        subroutine Pkg_Class_TriedAndTrue_fi(arg)
        implicit none
        integer*8 arg
C       DO-NOT-DELETE splicer.begin(Pkg.Class.TriedAndTrue)
        real*8 refarray(1)
        integer*4 lower(1), upper(1), stride(1), index
        integer n
        call sidl_double__array_access_f(arg, refarray,
     $       lower, upper, stride, index)
        if (index .ne. 0) then
c we can assume stride(1) = 1 because of column-major specification
           n = 1 + upper(1) - lower(1)
           call TriedAndTrue(refarray(index), n)
        else
           write(*,*) 'ERROR: array alignment'
        endif
C       DO-NOT-DELETE splicer.end(Pkg.Class.TriedAndTrue)
        end
```

Similarly, it should look something like the following for FORTRAN 90, where the include statements are required at the top of the Impl file to ensure proper handling of subroutine names that have automatically been mangled by the Babel compiler:

```
#include "Pkg_Class_fAbbrev.h"
#include "sidl_BaseClass_fAbbrev.h"
#include "sidl_BaseInterface_fAbbrev.h"
! DO-NOT-DELETE splicer.begin(_miscellaneous_code_start)
#include "sidl_double_fAbbrev.h"
! DO-NOT-DELETE splicer.end(_miscellaneous_code_start)
.
.
.
subroutine Pkg_Class_TriedAndTrue_mi(arg)
  ! DO-NOT-DELETE splicer.begin(Pkg.Class.TriedAndTrue.use)
  use SIDL_double_array
  ! DO-NOT-DELETE splicer.end(Pkg.Class.TriedAndTrue.use)
  implicit none
  type(sidl_double_a) :: arg

! DO-NOT-DELETE splicer.begin(Pkg.Class.TriedAndTrue)
  real (selected_real_kind(17,308)), dimension(1) :: refarray
  integer (selected_int_kind(8)), dimension(1)    :: low, up, str
  integer (selected_int_kind(8))    :: index, n
  call access(arg, refarray, low, up, str, index)
  if (index .ne. 0) then
    ! We can assume stride(1) = 1 because of column-major specification
    n = 1 + upper(1) - lower(1)
    call TriedAndTrue(refarray(index), n)
  else
    write(*,*) 'ERROR: array alignment'
  endif
! DO-NOT-DELETE splicer.end(Pkg.Class.TriedAndTrue)
end subroutine Pkg_Class_TriedAndTrue_mi
```

The C Macro API

For all the SIDL basic types except string, there is a C macro API for those who fear the function overhead of the C function API. When efficiency is not a concern, I recommend using the function API, but the C macro API is preferable to the direct access to the data structure. The macro API is not available for arrays of strings, interfaces or objects because the issues associated with memory and object reference management.

The macro API is very similar to the function API; however, a single set of macros applies to all the supported array types. The macro names are independent of the type of array you're accessing.

```
sidlArrayDim(array)
```

Return the dimension of array.

```
sidlLower(array,ind)
```

Return the lower bound on dimension ind.

```
sidlUpper(array,ind)
```

Return the upper bound on dimension ind.

```
sidlStride(array,ind)
```

Return the stride for dimension ind. The stride is the offset between elements in a particular dimension. It can be positive or negative. It is in terms of number of value types (i.e., it's 1 means contiguous regardless of what data type).

```
sidlArrayElem1(array, ind1)
sidlArrayElem2(array, ind1, ind2)
sidlArrayElem3(array, ind1, ind2, ind3)
sidlArrayElem4(array, ind1, ind2, ind3, ind4)
sidlArrayElem5(array, ind1, ind2, ind3, ind4, ind5)
sidlArrayElem6(array, ind1, ind2, ind3, ind4, ind5, ind6)
sidlArrayElem7(array, ind1, ind2, ind3, ind4, ind5, ind6, ind7)
```

Provide access to array elements to arrays of dimension 1–7. This macro can appear on the left hand side of an assignment or on the right hand side in an expression. These macros blindly assume that the dimension and indices are correct.

```
sidlArrayAddr1(array, ind1)
sidlArrayAddr2(array, ind1, ind2)
sidlArrayAddr3(array, ind1, ind2, ind3)
sidlArrayAddr4(array, ind1, ind2, ind3, ind4)
sidlArrayAddr5(array, ind1, ind2, ind3, ind4, ind5)
sidlArrayAddr6(array, ind1, ind2, ind3, ind4, ind5, ind6)
sidlArrayAddr7(array, ind1, ind2, ind3, ind4, ind5, ind6, ind7)
```

Return the address of elements in arrays of dimension 1–7. This macro can appear on the left hand side of an assignment or on the right hand side in an expression. These macros blindly assume that the dimension and indices are correct.

The C Data Structure

If even the macro interface is not fast enough for you, you can access the internal data structure for all the basic types except string. You cannot access the internal data structure for arrays of strings, interfaces and objects.

The basic form of the C data structure for type XXXX is:

```
struct sidl__array_vtable {

  /* Release resources associted with the array (refcount at zero) */
  void (*d_destroy)(struct sidl__array *);

  /* Clone or addRef depending on whether data is borrowed */
  struct sidl__array *(*d_smartcopy)(struct sidl__array *);

  /* Return the type of the array. */
  int32_t (*d_arraytype)(void);
};

struct sidl__array {
  int32_t                         *d_lower;
  int32_t                         *d_upper;
  int32_t                         *d_stride;
  const struct sidl__array_vtable *d_vtable;
  int32_t                          d_dimen;
  int32_t                          d_refcount;
};


struct sidl_XXXX__array {
  struct sidl__array       d_metadata;
  <value type for XXXX>   *d_firstElement;
};
```

The string "<value type for XXXX>" should be replaced by something like `sidl_bool`for an array of *`bool`*, `int32_t` for any array of *`int`*, `double` for an array of *`double`*, `int64_t` for an array of *`long`*, etc. (See Table 5.2)

`d_dimen` tells the dimension of the multi-dimensional array. `d_lower`, `d_upper`, and `d_stride` each point to arrays of `d_dimen int32_t`'s. `d_lower[i]` provides the lower bound for the index in dimension `i`, and `d_upper[i]` provides the upper bound for the index in dimension `i`. Both the lower and upper bounds are valid index values; the upper bound is not one past the end.

`d_borrowed` is true if the array does not managed the data that `d_firstElement` points too, and it is false otherwise. This mainly influences the behavior of the destructor.

Clients should not modify `d_lower`, `d_upper`, `d_stride`, `d_dimen`, `d_borrowed` or (in the case of pointers) the values to which they point.

`d_stride[i]` determines how elements are packed in dimension `i`. A value of 1 means that to get from element `j` to `j+1` in dimension `i`, you add one to the data pointer. Negative values for `d_stride` can be used to express a transposed matrix. The definition also allows either column or row major ordering for the data, and it also allows treating a subsection of an array as an array.

The data structure was inspired by the data structure used by Numeric Python; although, in Numeric Python, the stride is in terms of bytes. In SIDL, the stride is in terms of number of objects. One can convert to the Numeric Python view of things by multiplying the stride by the sizeof the value type.

5.5 SIDL Runtime

Inheritance

There is a small collection of interfaces and classes that are defined by the SIDL runtime library. Some of these objects are implicitly inherited by objects and classes.

All classes that do not explicitly extend another class implicitly extend *`sidl.BaseClass`*. All interfaces that do not explicitly extend another interface implicitly extend *`sidl.BaseInterface`*. Furthermore, *`sidl.BaseClass`* implements *sidl.BaseInterface*. This means that all classes can be cast to a *sidl.BaseClass* and all objects can be cast to *sidl.BaseInterface*.

All exceptions must explicitly implement the interfaces in *sidl.BaseException*. The easiest way to do this is to is to extend the provided class *sidl.SIDLException*. This is a class the implements the basic Exception functionality for you, including `getNote` and `setNote`. You may also override one or more of these functions if you wish.

If a method in SIDL claims to throw an object that does not inherit from *sidl.BaseException*, this is an error and will be reported by Babel.

Interfaces

The SIDL runtime library provides three sets of interfaces:

Base The base class, interface, and exception upon which all Babel-enabled software builds.

Library Handler The DLL and Loader classes facilitate dynamic loading of objects at runtime.

Introspection The ClassInfo interface and ClassInfoI class enable checking meta-data associated with a class.

The interfaces for the runtime library, as described in SIDL, are:

```
//
// File:        sidl.sidl
// Release:     $Name:  $
// Revision:    @(#) $Revision: 1.4 $
// Date:        $Date: 2004/01/28 19:32:28 $
// Description: SIDL interface description for the basic SIDL run-time library
//
// Copyright (c) 2001, The Regents of the University of Calfornia.
// Produced at the Lawrence Livermore National Laboratory.
// Written by the Components Team <components@llnl.gov>
// UCRL-CODE-2002-054
// All rights reserved.
//
// This file is part of Babel. For more information, see
// http://www.llnl.gov/CASC/components/. Please read the COPYRIGHT file
// for Our Notice and the LICENSE file for the GNU Lesser General Public
// License.
//
// This program is free software; you can redistribute it and/or modify it
// under the terms of the GNU Lesser General Public License (as published by
// the Free Software Foundation) version 2.1 dated February 1999.
//
// This program is distributed in the hope that it will be useful, but
// WITHOUT ANY WARRANTY; without even the IMPLIED WARRANTY OF
// MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the terms and
// conditions of the GNU Lesser General Public License for more details.
//
// You should have recieved a copy of the GNU Lesser General Public License
// along with this program; if not, write to the Free Software Foundation,
// Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA

/**
 * The <code>SIDL</code> package contains the fundamental type and interface
 * definitions for the <code>SIDL</code> interface definition language.  It
 * defines common run-time libraries and common base classes and interfaces.
 * Every interface implicitly inherits from <code>sidl.BaseInterface</code>
 * and every class implicitly inherits from <code>sidl.BaseClass</code>.
 *
```

```
 */
package sidl version 0.9.0 {

  /**
   * Every interface in <code>SIDL</code> implicitly inherits
   * from <code>BaseInterface</code>, and it is implemented
   * by <code>BaseClass</code> below.
   */
  interface BaseInterface {

    /**
     * <p>
     * Add one to the intrinsic reference count in the underlying object.
     * Object in <code>SIDL</code> have an intrinsic reference count.
     * Objects continue to exist as long as the reference count is
     * positive. Clients should call this method whenever they
     * create another ongoing reference to an object or interface.
     * </p>
     * <p>
     * This does not have a return value because there is no language
     * independent type that can refer to an interface or a
     * class.
     * </p>
     */
    void addRef();

    /**
     * Decrease by one the intrinsic reference count in the underlying
     * object, and delete the object if the reference is non-positive.
     * Objects in <code>SIDL</code> have an intrinsic reference count.
     * Clients should call this method whenever they remove a
     * reference to an object or interface.
     */
    void deleteRef();

    /**
     * Return true if and only if <code>obj</code> refers to the same
     * object as this object.
     */
    bool isSame(in BaseInterface iobj);

    /**
     * Check whether the object can support the specified interface or
     * class.  If the <code>SIDL</code> type name in <code>name</code>
     * is supported, then a reference to that object is returned with the
     * reference count incremented.  The callee will be responsible for
     * calling <code>deleteRef</code> on the returned object.  If
     * the specified type is not supported, then a null reference is
     * returned.
     */
    BaseInterface queryInt(in string name);

    /**
     * Return whether this object is an instance of the specified type.
     * The string name must be the <code>SIDL</code> type name.  This
     * routine will return <code>true</code> if and only if a cast to
     * the string type name would succeed.
     */
    bool isType(in string name);
```

```
  /**
   * Return the meta-data about the class implementing this interface.
   */
  ClassInfo getClassInfo();
}

/**
 * Every class implicitly inherits from <code>BaseClass</code>.  This
 * class implements the methods in <code>BaseInterface</code>.
 */
class BaseClass implements BaseInterface {
  /**
   * <p>
   * Add one to the intrinsic reference count in the underlying object.
   * Object in <code>SIDL</code> have an intrinsic reference count.
   * Objects continue to exist as long as the reference count is
   * positive. Clients should call this method whenever they
   * create another ongoing reference to an object or interface.
   * </p>
   * <p>
   * This does not have a return value because there is no language
   * independent type that can refer to an interface or a
   * class.
   * </p>
   */
  final void addRef();

  /**
   * Decrease by one the intrinsic reference count in the underlying
   * object, and delete the object if the reference is non-positive.
   * Objects in <code>SIDL</code> have an intrinsic reference count.
   * Clients should call this method whenever they remove a
   * reference to an object or interface.
   */
  final void deleteRef();

  /**
   * Return true if and only if <code>obj</code> refers to the same
   * object as this object.
   */
  final bool isSame(in BaseInterface iobj);

  /**
   * Check whether the object can support the specified interface or
   * class.  If the <code>SIDL</code> type name in <code>name</code>
   * is supported, then a reference to that object is returned with the
   * reference count incremented.  The callee will be responsible for
   * calling <code>deleteRef</code> on the returned object.  If
   * the specified type is not supported, then a null reference is
   * returned.
   */
  BaseInterface queryInt(in string name);

  /**
   * Return whether this object is an instance of the specified type.
   * The string name must be the <code>SIDL</code> type name.  This
   * routine will return <code>true</code> if and only if a cast to
   * the string type name would succeed.
```

```
     */
    bool isType(in string name);

    /**
     * Return the meta-data about the class implementing this interface.
     */
    final ClassInfo getClassInfo();
  }

  /**
   * Every exception implements <code>BaseException</code>. This interface
   * declares the basic functionality to get and set error messages and stack
   * traces.
   */
  interface BaseException {

    /**
     * Return the message associated with the exception.
     */
    string getNote();

    /**
     * Set the message associated with the exception.
     */
    void setNote(in string message);

    /**
     * Returns formatted string containing the concatenation of all
     * tracelines.
     */
    string getTrace();

    /**
     * Adds a stringified entry/line to the stack trace.
     */
    void add[Line](in string traceline);

    /**
     * Formats and adds an entry to the stack trace based on the
     * file name, line number, and method name.
     */
    void add(in string filename, in int lineno, in string methodname);
  }

  /**
   * <code>SIDLException</code> provides the basic functionality of the
   * <code>BaseException</code> interface for getting and setting error
   * messages and stack traces.
   */
  class SIDLException implements-all BaseException {
  }

  /**
   * When loading a dynamically linked library, there are three
   * settings: LOCAL, GLOBAL and SCLSCOPE.
   */
  enum Scope {
     /** Attempt to load the symbols into a local namespace. */
     LOCAL,
```

```
    /** Attempt to load the symbols into the global namespace. */
    GLOBAL,
    /** Use the scope setting from the SCL file. */
    SCLSCOPE
 }

 /**
  * When loading a dynmaically linked library, there are three
  * settings: LAZY, NOW, SCLRESOLVE
  */
  enum Resolve {
    /** Resolve symbols on an as needed basis. */
    LAZY,
    /** Resolve all symbols at load time. */
    NOW,
    /** Use the resolve setting from the SCL file. */
    SCLRESOLVE
  }

 /**
  * The <code>DLL</code> class encapsulates access to a single
  * dynamically linked library.  DLLs are loaded at run-time using
  * the <code>loadLibrary</code> method and later unloaded using
  * <code>unloadLibrary</code>.  Symbols in a loaded library are
  * resolved to an opaque pointer by method <code>lookupSymbol</code>.
  * Class instances are created by <code>createClass</code>.
  */
 class DLL {

   /**
    * Load a dynamic link library using the specified URI.  The
    * URI may be of the form "main:", "lib:", "file:", "ftp:", or
    * "http:".  A URI that starts with any other protocol string
    * is assumed to be a file name.  The "main:" URI creates a
    * library that allows access to global symbols in the running
    * program's main address space.  The "lib:X" URI converts the
    * library "X" into a platform-specific name (e.g., libX.so) and
    * loads that library.  The "file:" URI opens the DLL from the
    * specified file path.  The "ftp:" and "http:" URIs copy the
    * specified library from the remote site into a local temporary
    * file and open that file.  This method returns true if the
    * DLL was loaded successfully and false otherwise.  Note that
    * the "ftp:" and "http:" protocols are valid only if the W3C
    * WWW library is available.
    *
    * @param uri          the URI to load. This can be a .la file
    *                     (a metadata file produced by libtool) or
    *                     a shared library binary (i.e., .so,
    *                     .dll or whatever is appropriate for your
    *                     OS)
    * @param loadGlobally <code>true</code> means that the shared
    *                     library symbols will be loaded into the
    *                     global namespace; <code>false</code>
    *                     means they will be loaded into a
    *                     private namespace. Some operating systems
    *                     may not be able to honor the value presented
    *                     here.
    * @param loadLazy     <code>true</code> instructs the loader to
    *                     that symbols can be resolved as needed (lazy)
```

```
     *                     instead of requiring everything to be resolved
     *                     now (at load time).
     */
    bool loadLibrary(in string uri,
                     in bool loadGlobally,
                     in bool loadLazy);

    /**
     * Get the library name.  This is the name used to load the
     * library in <code>loadLibrary</code> except that all file names
     * contain the "file:" protocol.
     */
    string getName();

    /**
     * Unload the dynamic link library.  The library may no longer
     * be used to access symbol names.  When the library is actually
     * unloaded from the memory image depends on details of the operating
     * system.
     */
    void unloadLibrary();

    /**
     * Lookup a symbol from the DLL and return the associated pointer.
     * A null value is returned if the name does not exist.
     */
    opaque lookupSymbol(in string linker_name);

    /**
     * Create an instance of the SIDL class.  If the class constructor
     * is not defined in this DLL, then return null.
     */
    BaseClass createClass(in string sidl_name);
  }

  /**
   * Class <code>Loader</code> manages dyanamic loading and symbol name
   * resolution for the SIDL runtime system.  The <code>Loader</code> class
   * manages a library search path and keeps a record of all libraries
   * loaded through this interface, including the initial "global" symbols
   * in the main program.  Unless explicitly set, the search path is taken
   * from the environment variable SIDL_DLL_PATH, which is a semi-colon
   * separated sequence of URIs as described in class <code>DLL</code>.
   */
  class Loader {

    /**
     * Set the search path, which is a semi-colon separated sequence of
     * URIs as described in class <code>DLL</code>.  This method will
     * invalidate any existing search path.
     */
    static void setSearchPath(in string path_name);

    /**
     * Return the current search path.  If the search path has not been
     * set, then the search path will be taken from environment variable
     * SIDL_DLL_PATH.
     */
    static string getSearchPath();
```

```
/**
 * Append the specified path fragment to the beginning of the
 * current search path.  If the search path has not yet been set
 * by a call to <code>setSearchPath</code>, then this fragment will
 * be appended to the path in environment variable SIDL_DLL_PATH.
 */
static void addSearchPath(in string path_fragment);

/**
 * Load the specified library if it has not already been loaded.
 * The URI format is defined in class <code>DLL</code>.  The search
 * path is not searched to resolve the library name.
 *
 * @param uri          the URI to load. This can be a .la file
 *                     (a metadata file produced by libtool) or
 *                     a shared library binary (i.e., .so,
 *                     .dll or whatever is appropriate for your
 *                     OS)
 * @param loadGlobally <code>true</code> means that the shared
 *                     library symbols will be loaded into the
 *                     global namespace; <code>false</code>
 *                     means they will be loaded into a
 *                     private namespace. Some operating systems
 *                     may not be able to honor the value presented
 *                     here.
 * @param loadLazy     <code>true</code> instructs the loader to
 *                     that symbols can be resolved as needed (lazy)
 *                     instead of requiring everything to be resolved
 *                     now.
 * @return if the load was successful, a non-NULL DLL object is returned.
 */
static DLL loadLibrary(in string uri,
                       in bool loadGlobally,
                       in bool loadLazy);

/**
 * Append the specified DLL to the beginning of the list of already
 * loaded DLLs.
 */
static void addDLL(in DLL dll);

/**
 * Unload all dynamic link libraries.  The library may no longer
 * be used to access symbol names.  When the library is actually
 * unloaded from the memory image depends on details of the operating
 * system.
 */
static void unloadLibraries();

/**
 * Find a DLL containing the specified information for a SIDL
 * class. This method searches SCL files in the search path looking
 * for a shared library that contains the client-side or IOR
 * for a particular SIDL class.
 *
 * @param sidl_name  the fully qualified (long) name of the
 *                   class/interface to be found. Package names
 *                   are separated by period characters from each
```

```
     *                   other and the class/interface name.
     * @param target     to find a client-side binding, this is
     *                   normally the name of the language.
     *                   To find the implementation of a class
     *                   in order to make one, you should pass
     *                   the string "ior/impl" here.
     * @param lScope     this specifies whether the symbols should
     *                   be loaded into the global scope, a local
     *                   scope, or use the setting in the SCL file.
     * @param lResolve   this specifies whether symbols should be
     *                   resolved as needed (LAZY), completely
     *                   resolved at load time (NOW), or use the
     *                   setting from the SCL file.
     * @return a non-NULL object means the search was successful.
     *         The DLL has already been added.
     */
    static DLL findLibrary(in string  sidl_name,
                           in string  target,
                           in Scope   lScope,
                           in Resolve lResolve);
  }

  /**
   * This provides an interface to the meta-data available on the
   * class.
   */
  interface ClassInfo {
     /**
      * Return the name of the class.
      */
     string getName();

    /**
     * Get the version of the intermediate object representation.
     * This will be in the form of major_version.minor_version.
     */
     string getIORVersion();
  }

  /**
   * An implementation of the <code>ClassInfo</code> interface. This provides
   * methods to set all the attributes that are read-only in the
   * <code>ClassInfo</code> interface.
   */
  class ClassInfoI implements-all ClassInfo {
     /**
      * Set the name of the class.
      */
     final void setName(in string name);

     /**
      * Set the IOR major and minor version numbers.
      */
     final void setIORVersion(in int major, in int minor);
  }
}
```

5.6 Objects

One of the strategies that SIDL uses to enforce language interoperability is to define an object model that it supports across all language bindings. This enables real object-oriented programming in non OO languages such as C and FORTRAN 77. This also means that the inheritance mechanisms inside real OO languages may be circumvented.

Contrary to newer scripting languages such as Python and Ruby, not everything in SIDL is an object. Only classes (abstract or not) and interfaces are objects. Everything else (e.g. arrays, enums, strings, ints) is something other than an object and therefore outside the scope of this Section.

Babel's Object Model

SIDL defines three types of objects: interfaces, classes, and abstract classes. A SIDL *`interface`* is akin to a Java interface or a C++ pure abstract base class. It is an object that defines methods (aka member functions), but carries no implementation of those methods. A *`class`* by comparison is always concrete; meaning that there is an implementation for each of its methods and it can be instantiated. An *`abstract class`* falls somewhere between an *`interface`* and a *`class`*. It has at least one method unimplemented, so it cannot be instantiated, but it also may have several methods that are implemented and these implementations can be inherited.

SIDL supports multiple inheritance of interfaces and single inheritance of implementation. This is a strategy found in other OO languages such as Java and ObjectiveC. The words to distinguish these two forms of inheritance are *`extends`* and *`implements`*. Interfaces can extend multiple interfaces, but they cannot implement anything. Classes can extend at most one other class (abstract or not), but can implement multiple interfaces.

Furthermore, any inherited abstract methods (inherited from either and abstract parent class or and implemented interface) will default to abstract unless they are re-declared in the current class. If a concrete class implements many large interfaces, this can result in a fairly large list of redeclared functions in the class definition. As a shortcut, we included the `implements-all` directive, a short hand that states explicitly that we intend to implement every method in the named interface concretely. That's why, in the following example, class B must be declared abstract, but class D is concrete. Class B does not redeclare the `printMe` function, but class D `implements-all`. There is no similar directive for inheritance from abstract classes.

We display a small SIDL file below and finish this Subsection with a discussion of its details.

```
package object version 1.0 {

  interface A {
    void display();
    void printMe();
  }

  abstract class B implements A {
    void display();
  }

  class C extends B {
    void printMe();
  }

  class D implements-all A {
  }
}
```

`object.A` is an interface that has two methods *`display()`* and *`print()`*. Both of these methods take no arguments and return no value. (We will discuss arguments and return values in the next section.) Since *`object.A`* is an interface, there is no implementation associated with it, and Babel will not generate any implementation code associated with it.

`object.B` is an abstract class that inherits from *`object.A`*. Since it redeclares the *`display()`* method, Babel will generate the appropriate code for an implementation of this method only. It will not generate code for the other inherited method *`print()`* (since it wasn't declared in the SIDL file) and it will not generate constructors/destructors since the class is abstract.

object.C is a concrete class that extends the abstract class *object.B* it then lists only the unimplemented method *print()*, implying that it will use the implementation of *display()* it inherited from its parent.

object.D is also a concrete class that uses the *implements-all*

directive. This is identical to using *implements* and then listing all the methods declared in the interface. The *implements-all* directive was added to SIDL as a convenience construct and to save excessive typing in the SIDL file. By virtue of the *implements-all* directive, *object.D* will provide its own implementation of all of *object.A*'s methods, namely *display()* and *print()*.

Methods on Objects

Methods in SIDL are virtual by default. This means that the actual binding of a method invocation to an actual implementation is determined at runtime, based on the concrete type of the object.

SIDL currently defines three modifiers to methods that change their default behavior.

- *final* : Final methods are the opposite of virtual. While they may still be inherited by child classes, they cannot be overridden.

- *static* : Static methods are sometimes called "class methods" because they are part of a class, but do not depend on an object instance. In non-OO languages, this means that the typical first argument of an instance is removed. In OO languages, these are mapped directly to an Java or C++ static method.

- *oneway* : reserved for future use.

Parameter Passing

Each parameter in a method call obeys the following syntax

```
[ (modifier) ] (mode) (type) (name)
```

Where `(mode)` is one of *in*, *out*, or *inout*; `(type)` is any SIDL recognized type; and `(name)` is any non-reserved word[2]. The `(modifier)` is optional, and currently unimplemented. SIDL currently reserves the word *copy* for future use as an parameter modifier, and may add others in the future[3].

For new users, the parameter's mode (e.g. *in*, *out*, or *inout*) is perhaps the most troublesome. On the surface, it's easy to explain that *in* parameters are passed into the code, *out* parameters come out, and *inout* parameters do both. More specifically the rules are:

1. *in* does not mean `const`.

2. *in* arguments are passed by value, therefore what happens inside the function has no effect on the value passed in (from the perspective of the caller).

3. *inout* arguments are passed by reference. The callee is allowed to do whatever it wants with the data passed in, and changes made by the callee are sent back to the caller. For interfaces, classes, and normal arrays, the callee can even destroy the reference, create a new object or array, and return a reference to it.

4. Objects, interfaces and arrays should be allocated using the create methods provided. Types created on the stack should never be passed as an *inout* argument, since the implementation may want to destroy it.

5. *out* arguments are also passed by reference, but the incoming value is ignore and typically overwritten. *Do Not* attempt to pass in a value to a function through an out argument. There is no guarantee that the data will make it to the Implementation, and if the data is lost, there is no guarantee the reference will be correctly destroyed.

[2] Refer to Section A.2 for the list of reserved words

[3] Babel is still pre-1.0 after all!

Method Overloading

Method overloading is the object-oriented practice of defining more than one method with the same name in a class. Doing so allows the convenient reuse of a method name when, for example, the underlying implementations differ based on the types of the arguments. Actually, support for overloaded methods typically relies on the signature of each method to ensure uniqueness. In this case, the signature consists of the method name along with the number, types, and ordering of its arguments.

Since Babel supports languages that do not support method overloading, a mechanism for generating unique names was needed. These are typically generated by compilers based on hashing the argument types into the method name. However, developers often manually address this with far fewer characters than would be used by a compiler. Consequently, it was determined it would be more efficient to leave the task of identifying the unique name to the developer. Therefore, Babel allows the specification of the base, or short, method name along with an optional method name extension as illustrated in the SIDL file below for the `getValue` method.

```
package Overload version 1.0 {

  class Sample {
    int      getValue ( );
    int      getValue[Int]( in int v );
    double   getValue[Double]( in double v );
  }
}
```

Thus, the full method name is the concatenation of the short name followed by the name extension. When generating code for supported languages, Babel makes use of either the short or full method name as appropriate for the language(s) involved. For those that support method overloading, such as C++ and Java, Babel relies only on the short method name, thus ignoring the extension. For the rest, like C, Fortran, and Python, Babel must make use of the full name to ensure methods are uniquely identified.

In the example above, the first method specification takes no arguments so has no name extension. This is acceptable because there are no potentially conflicting methods at this point for any programming language supported by Babel. The second method, with the user-defined name extension of `Int`, takes a single int argument, resulting in the unique method name `getValueInt`. The last method, with a user-defined name extension of `Double`, takes a single double argument, resulting in the unique method name of `getValueDouble`. Examples of calling overloaded methods from Babel-supported languages can be found in the respective language binding chapters.

5.7 XML Repositories

Even though SIDL is currently the primary input format for Babel, it is not the only format Babel understands. For type repositories (similar in function to include directories for C/C++ headers) the preferred language to articulate types is XML.

Babel has the capabilities to convert SIDL files into XML files adhering to the `SIDL.dtd`. This capability is explained further in Chapter 13. The XML files in these repositories can be included in subsequent runs quickly since all the external references were resolved by Babel during their creation. A SIDL file may refer to unresolved types.

Part II

Supported Language Bindings

Chapter 6

C Bindings

Contents

6.1 Introduction

This chapter provides an introduction to the C bindings for SIDL. Babel supports both callers and callees written in C so this chapter illustrates the use of Babel for both. That is, it shows how to use Babel to wrap the implementation of software written in C as well as how to call software, possibly implemented in any other supported language, from C.

Since Babel's Intermediate Object Representation (IOR) is written in C, the C bindings are very similar to the IOR. In addition, all of the objects in the sidl namespace (e.g. sidl::BaseClass, etc.) are implemented in C, so clients can develop solely with a C compiler if necessary. Of course this seems a little silly since the intent of Babel is to provide multilingual interoperability.

6.2 Basic Types

The basic types in SIDL are mapped into C according to Table 6.1.

6.3 Header files

If you would like to use type `X.Y.Z` from C (package X, subpackage Y, class Z), you should `#include "X_Y_Z.h"` . If you would like to include the header files for a whole package `X.Y`, you can `#include "X_Y.h"`. For example, you can include all the types in the `sidl` namespace with `#include "sidl.h"`.

Each client side header file will ensure that `sidl_header.h` is included. `sidl_header.h` defines:

1. `struct sidl_dcomplex` for the SIDL dcomplex type with parts named `real` and `imaginary`;
2. `struct sidl_fcomplex` for the SIDL fcomplex type with parts named `real` and `imaginary`;

Table 6.1: SIDL to C Type Mappings

SIDL TYPE	C TYPE
int	int32_t
long	int64_t
float	float
double	double
bool	typedef sidl_bool
char	char
string	char *
fcomplex	struct sidl_fcomplex
dcomplex	struct sidl_dcomplex
enum	enum
opaque	void *
interface	typedef
class	typedef
array	struct *

3. `int32_t` and `int64_t` for the SIDL int and long types;
4. a typedef for `sidl_bool` for the SIDL bool type;
5. preprocessor symbols `TRUE` and `FALSE`; and
6. function prototypes for the multi-dimensional array APIs for the basic SIDL types.

In general, clients don't need to worry about including `sidl_header.h` because the Babel generated header files will include it for you.

6.4 Mapping for classes, interfaces, arrays and r-arrays

Because C doesn't have built in mechanisms for protecting the global namespace, the C mapping attempts to avoid namespace collisions by using struct and method names that incorporate all the naming information from the package, class and method names. For a type `Z` in package `X.Y`, the name of the type that C clients use for an object reference is `X_Y_Z`. `X_Y_Z` is defined as follows in the `X_Y_Z.h` header file:

```
struct X_Y_Z__object;
struct X_Y_Z__array;
typedef struct X_Y_Z__object* X_Y_Z;
```

This code fragment also shows that `struct X_Y_Z__array` is used for a multi-dimensional array of `X.Y.Z` objects. Here are some additional concrete examples of the object and interface reference types derived by the C mapping:

```
/**
 * Symbol "sidl.BaseClass" (version 0.5.1)
 *
 * Every class implicitly inherits from <code>BaseClass</code>.  This
 * class implements the methods in <code>BaseInterface</code>.
 */
struct sidl_BaseClass__object;
struct sidl_BaseClass__array;
typedef struct sidl_BaseClass__object* sidl_BaseClass;

/**
```

```
 * Symbol "sidl.BaseInterface" (version 0.5.1)
 *
 * Every interface in <code>SIDL</code> implicitly inherits
 * from <code>BaseInterface</code>, and it is implemented
 * by <code>BaseClass</code> below.
 */
struct sidl_BaseInterface__object;
struct sidl_BaseInterface__array;
typedef struct sidl_BaseInterface__object* sidl_BaseInterface;
```

Here is an example of the C client-side binding for an r-array. This example is for the `solve` example from Section 5.4. Here, I assume that the package name is `num`, and the class name is `Linsol`. The data for each array is passed as a `double` pointer, and the index parameters are normal `in` ints.

```
/** C client-side API for solve method */
void num_Linsol_solve(/*in*/ num_Linsol self,
                      /*in*/ double* A,
                      /*inout*/ double* x,
                      /*in*/ double* b,
                      /*in*/ int32_t m,
                      /*in*/ int32_t n);
```

The one catch for C programmers is that `A` is in column-major order — not the typical row-major ordering used in C. To access the element in row `i` and column `j`, you can use the `RarrayElem2(A,i,j,m)`. `RarrayElem2` is a convenience macro for C and C++ programmers to access r-arrays in column-major order. To access memory by stride one make `i` your inner loop.

Passing `NULL` for `A`, `x`, or `b` is not allowed. You must always pass a valid pointer.

6.5 Calling SIDL methods from C

The names of the C functions used to call SIDL methods are a concatenation of the package name, the class or interface name and the method name(s) with the period characters changed to underscores . If the method is specified as being overloaded (i.e., has a name extension) , the full method name is the concatenation of the package name, the class or interface name, the method name, *and* the type extension. For non-static methods, the object or interface pointer is passed as the first parameter before any of the formal parameters. This parameter operates like an `in` parameter.

Examples of calls to SIDL overloaded methods are based on the `overload_sample.sidl` file shown in Section 5.6. Recall that the file describes three versions of the `getValue` method. The first takes no arguments, the second takes an integer argument, and the third takes a boolean. Each is called in the code snippet below:

```
  int b1, i1, iresult, nresult;

  Overload_Sample t  = Overload_Sample__create ();

  nresult = Overload_Sample_getValue(t);
  iresult = Overload_Sample_getValueInt(t, i1);
  bresult = Overload_Sample_getValueBool(t, b1);
```

Here are the C bindings for the critical `addRef` and `deleteRef` methods from `sidl.BaseInterface`. These methods are mentioned in particular because C clients must manage object reference counts themselves.

```
void
sidl_BaseInterface_addRef(
  sidl_BaseInterface self);

void
sidl_BaseInterface_deleteRef(
  sidl_BaseInterface self);
```

These same methods can be called from the `sidl.BaseClass` bindings. In fact, every C binding for an interface or class will have entries for `addRef` and `deleteRef`.

```
void
sidl_BaseClass_addRef(
  sidl_BaseClass self);

void
sidl_BaseClass_deleteRef(
  sidl_BaseClass self);
```

6.6 Catching and Throwing Exceptions in C

For methods that can throw exceptions, there is an extra `out` argument in the generated code that holds the exception. For maximum backward compatibility and consistency, the extra argument is of type `sidl.BaseInterface`. When the exception parameter is not `NULL`, it indicates that an exception has been thrown. When an exception is thrown, the caller should ignore the value of the other `out` parameters as well as the function's return value. Every time you call a method that potentially can throw an exception, you must check the result. Otherwise, those exceptions will be utterly ignored and leak memory. There are are four macros provided in sidl_Exception.h to help with exception checking. Their use is fairly obvious from their names. They are:

```
SIDL_THROW(EX_VAR,EX_CLS,MSG)
SIDL_CHECK(EX_VAR)
SIDL_CLEAR(EX_VAR)
SIDL_CATCH(EX_VAR,sidl_NAME)
```

In these macros, EX_VAR is the exception object itself, EX_CLS is the name of the SIDL type we wish the exception to be in a string, MSG is the message we wish to include with the exception and a string, and sidl_NAME is the type of the exception we expect to catch, as a string.

The following SIDL method taken from the Babel regression tests demonstrates how exceptions are handled.

```
int getFib(in int n, in int max_depth, in int max_value, in int depth)
  throws NegativeValueException, FibException;
```

Here is the C binding for this method:

```
int32_t
ExceptionTest_Fib_getFib(
  ExceptionTest_Fib self,
  int32_t n,
  int32_t max_depth,
  int32_t max_value,
  int32_t depth,
  sidl_BaseInterface *_ex);
```

Here is an example of how to perform exception handling in C using a package of macros defined in `sidl_Exception.h`. Note that the macros assume the exception class that is being thrown and caught inherits from or implements *`sidl.BaseException`* — something guaranteed by Babel.

```
#include "sidl_Exception.h"
/* ...numerous lines deleted... */
  int x;
  sidl_BaseInterface _ex = NULL;

  x = ExceptionTest_Fib_getFib(f, 10, 1, 100, 0, &_ex);
  if (SIDL_CATCH(_ex, "ExceptionTest.TooDeepException")) {
    traceback(_ex);
    SIDL_CLEAR(_ex);
```

```
    }
    else if (SIDL_CATCH(_ex, "ExceptionTest.TooBigException")) {
      traceback(_ex);
      SIDL_CLEAR(_ex);
    }
    else if (_ex == NULL) {
      return FALSE;
    }
    SIDL_CHECK(_ex);
    return TRUE;

    EXIT:;
      traceback(_ex);
      SIDL_CLEAR(_ex);
      return FALSE;
```

You do not have to use the macros provided in `sidl_Exception.h` if you do no want to. You can check `_ex` by checking if it is not `NULL` and then trying to cast it to the various potential exception types.

The following code snippet shows how to throw an exception in C using the macros from `sidl_Exception.h`. The first argument to `SIDL_THROW` is the exception output parameter, and the second argument is the type of exception being thrown. The third argument provides a textual description of the exception.

```
#include "sidl_Exception.h"
/* ...numerous lines deleted... */
int32_t
impl_ExceptionTest_Fib_getFib(
  ExceptionTest_Fib self, int32_t n, int32_t max_depth, int32_t max_value,
    int32_t depth, sidl_BaseInterface* _ex)
{
  /* DO-NOT-DELETE splicer.begin(ExceptionTest.Fib.getFib) */
  if (n < 0) {
    SIDL_THROW(*_ex,
               ExceptionTest_NegativeValueException,
               "called with negative n");
  }
  /* ...lines deleted... */
  EXIT:;
    /* SIDL_THROW macro will jump here. */
    /* Clean up code should be here. */
    return theValue;
  /* DO-NOT-DELETE splicer.end(ExceptionTest.Fib.getFib) */
}
```

The code section labeled `EXIT` is where you should put clean up code. The caller will ignore all the values leaving your C function (i.e., `out` or `inout` parameters) because you have thrown an exception, so your code should delete any references you were planning to return to the caller. It's good practice to set all `inout` and `out` array, interface or class pointers to `NULL`. This makes things work out better for clients who forget to check if an exception occurred or willfully choose to ignore it.

6.7 Implicitly defined methods

The C binding for interfaces and classes includes two methods for perform type casts. The methods are named `_cast` and `_cast2`. The leading underscore prevents these built in methods from conflicting with a user method because user methods cannot begin with an underscore. Neither of these methods increases the reference count of the underlying object — this is contrary to standard methods that always return new reference counts. Every object has these two methods, we will use `sidl.BaseClass` as an example. Here are the signatures for `_cast` and `_cast2` from `sidl.BaseClass`.

```
sidl_BaseClass
sidl_BaseClass__cast(
  void* obj);

void*
sidl_BaseClass__cast2(
  void* obj,
  const char* type);
```

The _cast method attempts to cast a SIDL interface or object pointer to a pointer to sidl.BaseClass. The _cast2 method attempts to cast a SIDL interface or object pointer to a pointer to an interface or object pointer of the type named type. In the case of _cast2, the client is responsible for casting the return value into the proper pointer type. Both methods are NULL safe. A NULL return value indicates that the cast failed or that obj was NULL.

Non-abstract classes have an additional implicit method called _create to create new instances of the class. Interfaces and abstract classes do not have this method because you cannot instantiate them. The _create method returns a new reference that the client must manage. Here is an example of its signature.

```
/**
 * Constructor function for the class.
 */
sidl_BaseClass
sidl_BaseClass__create(void);
```

6.8 Invoking Babel to generate C bindings

To create C stubs (i.e. code to support C clients to a set of SIDL classes or interfaces), you should invoke Babel as follows [1]:

```
% babel --client=C file.sidl
```

or more cryptically

```
% babel -cC file.sidl
```

This will create more files than you can shake a stick at. The files ending in _IOR.h and _IOR.c are the Intermediate Object Representation. The files ending with _Stub.c are the C stubs — the interface between a C client and the IOR. The remaining header files have external C API that C clients may use.

To use the C stubs, you must compile the stub files whose file names end with _Stub.c and link them against the SIDL runtime library and a backend implementation.

6.9 Invoking Babel to generate C implementations

To implement a set of SIDL classes in C, you should invoke Babel as follows:

```
% babel --server=C file.sidl
```

or use the short form

```
% babel -sC file.sidl
```

[1] For information on additional command line options, refer to Section 3.2.

This will create a Makefile frament called `babel.make`, several C headers and source files. To create a working C implementation, the only files that need to be hand-edited are the C "Impl" files (header and source files that end in `_Impl.h` or `_Impl.c`. Changes to these files should be made between code splicer pairs. Code splicing is a technique Babel uses to preserve hand-edited code between multiple invocations of Babel. This allows a developer to refine their SIDL file without ruining all their previous implementations. Code between splicer pairs will be retained by subsequent invocations of Babel; code outside splicer pairs is not.

Here is an example of a code splicer pair in C.

```
/* DO-NOT-DELETE splicer.begin(num.Linsol._includes) */
/* Put additional includes or other arbitrary code here... */
/* DO-NOT-DELETE splicer.end(num.Linsol._includes) */
```

The following example shows the Babel generate implementation file for the `solve` example from Section 5.4. The r-array data is presented as `double` pointers, and the index variables are normal integers.

```
void
impl_num_Linsol_solve(/*in*/ num_Linsol self,
                      /*in*/ double* A, /*inout*/ double* x,
                      /*in*/ double* b,
                      /*in*/ int32_t m, /*in*/ int32_t n)
{
  /* DO-NOT-DELETE splicer.begin(num.Linsol.solve) */
  /* Insert the implementation of the solve method here... */
  /* DO-NOT-DELETE splicer.end(num.Linsol.solve) */
}
```

The data for the 2-D array `A` is in column-major order. Use of the `RarrayElem2` macro to access `A` is covered above in Section 6.4.

Chapter 7

C++ Bindings

Contents

7.1 Introduction

This chapter provides an introduction to Babel's C++ bindings. It illustrates the support provided for both C++ callers and C++ implementations, or callees.

Unlike C or FORTRAN 77, there is no runtime library created for a particular C++ compiler at installation. Instead, when you generate C++ from SIDL, you will find Stubs (aka proxy classes) generated for SIDL base classes and will have to compile and link them into your application.

That said, if you switch to a different compiler after installation, there may be some values set in `babel_config.h` that become invalid. This can be overcome by copying the header file, making the necessary changes, and placing the modified header file earlier in the include path than the original one.

7.2 Basic Types

The basic types in SIDL are mapped into C++ according to Table 7.1.

7.3 SIDL C++ Header Suffix

The first thing that C++ users will notice is that C++ headers have a ".hh" suffix to distinguish them from C's ".h" suffix. This convention was born out of necessity to distinguish both differing header files and their include guards.

Table 7.1: SIDL to C++ Type Mappings

SIDL TYPE	C++ TYPE
int	int32_t
long	int64_t
float	float
double	double
bool	bool
char	char
string	std::string
fcomplex	sidl::fcomplex
dcomplex	sidl::dcomplex
enum	enum
opaque	sidl::opaque
interface	class
class	class
array	sidl::array (template specialization)

7.4 SIDL's Main C++ Header File

All C++ code generated by Babel `#include`'s a file called "`sidl_cxx.hh`". This file includes `babel_config.h`, the C header file that defines configuration information. Finally, `sidl_cxx.hh` defines some C++ classes in the SIDL namespace such as

- `sidl::StubBase` [implementation detail] Common base class for all C++ stubs (proxy classes)
- template<T,U,V> SIDL::array_mixin [implementation detail] Common base class for all C++ array classes.
- typedefs for `sidl::fcomplex`, `sidl::dcomplex`, and `sidl::opaque` (usually `std::complex`, `std::complex` and `void*`, respectively)
- `template<T> sidl::array` Template array type for SIDL arrays.
- template specializations [implementation detail] specialization of arrays of all SIDL types are defined in this file.

7.5 Calling Methods from C++

Since C++ is an object-oriented language, there is a lot less programmer overhead in using SIDL from the C++ perspective than from non-OO languages such as C or FORTRAN 77.

These proxy classes (we call "stubs") serve as the firewall between the application in C++ and Babel's internal workings. As one would expect, the proxy classes maintain minimal state so that, unlike C or FORTRAN 77, there is no special context argument added to non-static member functions.

Below are examples using standard classes. The first is an example of creating an object of the base class and its association to the base interface.

```
sidl::BaseClass object = sidl::BaseClass::_create();
sidl::BaseInterface interface = object;
```

Here is an example call to the addSearchPath in the SIDL.Loader class:

```
std::string s("/try/looking/here");
sidl::Loader::addSearchPath( s );
```

Table 7.2: SIDL Features Mapped onto C++

SIDL Feature	C++ Implementation
packages	C++ namespaces (no name transformations)
version numbers	ignored
interface	C++ class (called "stub", serves as a proxy to the implementation)
class	C++ class (called "stub", serves as a proxy to the implementation)
methods	C++ member functions; uses base method name when overloading; no name mangling; NOTE: Member functions beginning with a leading underscore "_" may be Babel internals, or specific to C++ binding.
static methods	Static C++ member functions; uses base method name when overloading; no name mangling; even works for dynamically loaded object's exceptions thrown and caught using C++ exception handling.
reference counting	SIDL C++ stubs can be treated as smart-pointers. Constructors, destructors, and operators are overloaded so that explicit calls to `addRef()` or `deleteRef()` are rarely needed.
casting	Assignment operators are overloaded to handle safe casting up and down the inheritance hierarchy. User should never call `dynamic_cast<>()` on a SIDL object since the stubs inheritance hierarchy does not follow the SIDL inheritance hierarchy. Attempted downcasts using assignment should be checked by a call to (`_is_nil()`, or `_not_nil()`).
instance creation	Use static member function "`_create`". The default constructor for a C++ stub creates the equivalent of a NULL pointer. Works only with non-abstract classes.

Examples of calls to SIDL overloaded methods are based on the `overload_sample.sidl` file shown in Section 5.6. Recall that the file describes three versions of the `getValue` method. The first takes no arguments, the second takes an integer argument, and the third takes a boolean. Each is called in the code snippet below:

```
bool b1, bresult;
int  i1, iresult, nresult;

Overload::Sample t  = Overload::Sample::_create();

nresult = t.getValue();
bresult = t.getValue(b1);
iresult = t.getValue(i1);
```

7.6 Catching and Throwing Exceptions in C++

Adapted from the Babel regression tests, the following is an example of a package called ExceptionTest that has a class named Fib with a method declared in SIDL as follows:

```
int getFib(in int n, in int max_depth, in int max_value, in int depth)
  throws NegativeValueException, FibException;
```

The corresponding C++ code fragment to use this method is:

```
ExceptionTest::Fib fib = ExceptionTest::Fib::_create();
try {
  int result = fib.getFib( 4, 100, 32000, 0 );
  cout << "Result of fib.getFib() = " << result << endl;
} catch ( ExceptionTest::NegativeValueException  e ) {
  // ...
```

```
} catch ( ExceptionTest::FibException e ) {
  // ...
}
```

This example shows the standard way to throw an exception in C++. You are not strictly required to call the `setNote` and `add` methods; however, these methods provide information that may be helpful in debugging or error reporting.

```
int32_t
ExceptionTest::Fib_impl::getFib (
  /*in*/ int32_t n,         /*in*/ int32_t max_depth,
  /*in*/ int32_t max_value, /*in*/ int32_t depth )
throw (
  ::ExceptionTest::NegativeValueException,
  ::ExceptionTest::FibException
){
  // DO-NOT-DELETE splicer.begin(ExceptionTest.Fib.getFib)
  if (n < 0) {
    NegativeValueException ex = NegativeValueException::_create();
    ex.setNote("n negative");
    ex.add(__FILE__, __LINE__, "ExceptionTest::Fib_impl::getFib");
    throw ex;
  }
  // several lines delete
  // DO-NOT-DELETE splicer.end(ExceptionTest.Fib.getFib)
}
```

7.7 Invoking Babel to generate C++ stubs

To create the C++ stubs from a SIDL file, invoke Babel as follows [1]:

```
% babel --client=C++ file.sidl
```

or simply

```
% babel -cC++ file.sidl
```

This will create a babel.make file, some C headers and sources, and many C++ headers and sources. Files ending in ".c" or ".h" are in C, files ending in ".cc" or ".hh" are C++.

You will need to compile and link the files together to use the C++ stubs.

7.8 Implementing SIDL Classes in C++

Much of the information from the previous section is pertinent to implementing a SIDL class in C++. The types of the arguments are as indicated in Table 7.1. Your implementation can call other SIDL methods, in which case follow the rules for client calls.

To create the implementation, you must first have a valid SIDL file, then invoke Babel as follows:

```
% babel --server=C++ file.sidl
```

or simply

[1] For information on additional command line options, refer to Section 3.2.

```
% babel -sC++ file.sidl
```

This will create a makefile fragment called babel.make, several C headers and source files, and numerous C++ header and source files. To create a working implementation, the only files that need to be hand-edited are the C++ ”Impl” files (header and source files that end in `_Impl.hh` or `_Impl.cc`). All your additions to this file should be made between code splicer pairs. Code splicing is a technique Babel uses to preserve hand-edited code between multiple invocations of Babel. This allows a developer to refine their SIDL file without ruining all their previous implementations. Code between splicer pairs will be retained by subsequent invocations of Babel; code outside splicer pairs is not.

Here is an example of a code splicer pair in C++. In this example, you would replace the line ”// Insert code here... ” with your implementation.

```
void MyPackage::MyClass::myMethod() {
    // DO-NOT-DELETE splicer.begin(MyPackage.MyClass.myMethod)
    // Insert code here...
    // DO-NOT-DELETE splicer.end(MyPackage.MyClass.myMethod)
}
```

It is important to understand where and why splicer blocks occur. Splicer blocks appear at the beginning and end of each Impl header and source file; for developers to add `#include`’s and other miscellaneous items respectively. In the headers, there is a splicer block that allows a user to make the impl class inherit from some other class. From SIDL’s point of view this is private inheritance — meaning that it is useful for inheriting implementation details, but they can’t be automatically exposed to the SIDL method dispatch mechanism. There is a splicer block inside the class definition for developers to add any data members the wish to the class. In the source files, splicer blocks appear in each method implementation. There are two implicit methods (i.e., methods that did not appear in the SIDL file) that must also be implemented. The _ctor method is a constructor function that is run whenever an object is created. The _dtor method is a destructor function that is run whenever an object is destroyed. If the object has no state, these functions are typically empty.

7.9 Accessing SIDL Arrays From C++

Although it is feasible to expose the underlying C array API to create, destroy and access array elements and meta-data, the C++ bindings provide a `sidl::array<T>` template mechanism that is more in keeping with C++ idioms.

For SIDL built-in types, template specializations of `sidl::array<T>` are defined in `sidl_cxx.hh`. For SIDL interface and classes, the array template is again specialized in the corresponding stub header. The reason for the extensive use of template specialization is an effort to hide the detail that the array implementation is really templated on three terms: the type of the C struct that represents the array internally, the internal representation of each item in the array, and the C++ representation of each item in the array. (See `array_mixin` in `sidl_cxx.hh` for grungy implementation details.)

An example is given below.

```
int32_t len = 10; // array length=10
int32_t dim = 1;  // one dimensional
int32_t lower[1] = {0}; // zero offset
int32_t upper[1] = {len-1};
int32_t prime = nextPrime(0);

// create a SIDL array of primes.
sidl::array<int32_t> a = sidl::array<int32_t>::createRow(dim, lower, upper);
for( int i=0; i<len; ++i ) {
   prime = nextPrime( prime );
   a.set(i, v);
}
```

Of course, the example above is only one way to create an array. The list of member functions for all C++ array classes is:

```
// constructors
array ( array_ior_t * src ); // internal
array () ;                   // empty

// destructor
~array() ;

// creation
static array<x>
createRow( int32_t dimen, const int32_t lower[],
           const int32_t upper[]);
static array<x>
createCol( int32_t dimen, const int32_t lower[],
           const int32_t upper[]);
static array<x>
create1d( int32_t len);
static array<x>
create2dCol( int32_t m, int32_t n);
static array<x>
create2dRow( int32_t m, int32_t n);
array<x>
slice( int32_t dimen, const int32_t numElem[],
       const int32_t *srcStart = 0,
       const int32_t *srcStride = 0,
       const int32_t *newStart = 0);

void borrow( item_ior_t * first_element, int32_t dimen,
        const int32_t lower[], const int32_t upper[],
        const int32_t stride[]);

void addRef();
void deleteRef();

// get/set
item_cxx_wrapper_t get(int32_t i);
item_cxx_wrapper_t get(int32_t i1, int32_t i2);
item_cxx_wrapper_t get(int32_t i1, int32_t i2, int32_t i3);
item_cxx_wrapper_t get(int32_t i1, int32_t i2, int32_t i3, int32_t i4);
item_cxx_wrapper_t get(const int32_t *indices);

void set(int32_t i, item_cxx_wrapper_t elem);
void set(int32_t i1, int32_t i2, item_cxx_wrapper_t elem);
void set(int32_t i1, int32_t i2, int32_t i3,
 item_cxx_wrapper_t elem);
void set(int32_t i1, int32_t i2, int32_t i3, int32_t i4,
 item_cxx_wrapper_t elem);
void set(const int32_t *indices, item_cxx_wrapper_t elem);

// other accessors
int32_t dimen() const;

int32_t lower( int32_t dim ) const;

int32_t upper( int32_t dim ) const;

int32_t stride( int32_t dim ) const;

bool _is_nil() const;
```

```
bool _not_nil() const;

// get a const pointer to the actual array ior
const array_ior_t* _get_ior() const { return d_array; }

// get a non-const pointer to the actual array ior
array_ior_t* _get_ior() { return d_array;}
```

where

- `array_ior_t` is the type of the C struct that represents the array internally,
- `item_ior_t` is the internal representation of each item in the array,
- `item_cxx_wrapper_t` is the C++ representation of each item in the array

Please note that all SIDL array constructors are static methods returning a newly allocated array. Normally, you assign the return value to a variable.

The C++ mapping for r-arrays is essentially identical to the mapping for C (see Section 6.4). The only difference is that the C++ client header provides an overloaded version of each method containing an r-array taking normal SIDL arrays instead of raw data. For example, the `solve` method from Section 5.4 produces the following code in the client-side header file.

```
void solve (/*in*/    double* A,
            /*inout*/ double* x,
            /*in*/    double* b,
            /*in*/    int32_t m,
            /*in*/    int32_t n) throw ();

void solve (/*in*/    ::sidl::array<double> A,
            /*inout*/ ::sidl::array<double>& x,
            /*in*/    ::sidl::array<double> b) throw();
```

Please note that multi-dimensional arrays, such as `A` in this case, are stored in column-major ordering. Babel provides macros to access r-array data correctly. In this example, you can use `RarrayElem2(A, i, j, m)` to access the element in row `i` and column `j`. There are similar macros for arrays of dimension 1 through 7 in `sidlArray.h`.

If you were implementing `solve` in C++, the Babel generated implementation file for it would look like this:

```
void num::Linsol_impl::solve (/*in*/    double* A,
                              /*inout*/ double* x,
                              /*in*/    double* b,
                              /*in*/    int32_t m,
                              /*in*/    int32_t n ) throw ()
{
  // DO-NOT-DELETE splicer.begin(num.Linsol.solve)
  // insert implementation here
  // DO-NOT-DELETE splicer.end(num.Linsol.solve)
}
```

To access memory by stride one make the row index your inner loop and the column index your outer loop.

7.10 C++ Specific Babel Command Line Options

The Babel C++ binding has one command line option particular to it. Using the option –cxx-ior-exception (or it's short form '-x') will generate C++ Babel stubs that check for a null IOR whenever a method is called on them. If a method is called on a Stub holding a null IOR, it will throw a NullIORException. If this option is not passed to Babel, the program will simply crash, as C++ would do normally with a null pointer.

Chapter 8

FORTRAN 77 Bindings

Contents

8.1 Introduction

This chapter provides an introduction to Babel's FORTRAN77 bindings. Babel supports both callers and callees written in FORTRAN 77 so this chapter illustrates the use of Babel for both. That is, it shows how to use Babel to wrap the implementation of software written in FORTRAN 77 as well as how to call software, possibly implemented in any other supported language, from FORTRAN 77.

8.2 Basic Types

For pointer types, such as opaque, interface, class, and array, a 64-bit integer is used, so FORTRAN 77 code will be portable between systems with a 32 bit address space and systems with a 64 bit address space. On a 32 bit system, the upper 32 bits of these quantities are ignored. Systems with more than 64-bit pointers aren't currently supported.

Generally, clients should treat opaque, interface, class and array values as black boxes. However, there is one value that is special. A value of zero for any of these quantities indicates that the value does not refer to an object. Zero is the FORTRAN 77 equivalent of NULL . Any nonzero value is or should be a valid object reference. Developers writing in FORTRAN 77 should initialize values to be passed as in or inout parameters to zero or a valid object reference.

When mapping the SIDL string type into FORTRAN 77, some capability was sacrificed to make it possible to use normal looking FORTRAN 77 string handling. One difference is that all FORTRAN 77 strings have a limited fixed size. When implementing a subroutine with an out parameter, the size of the string is limited to 512 characters.

Enumerated types are just integer values. The constants are defined in an includable file assuming your FORTRAN 77 compiler supports some form of including.

Table 8.1: SIDL to FORTRAN 77 type mapping

SIDL TYPE	FORTRAN 77 TYPE
int	INTEGER*4
long	INTEGER*8
float	REAL
double	DOUBLE PRECISION
bool	LOGICAL
char	CHARACTER*1
string	CHARACTER*(*)
fcomplex	COMPLEX
dcomplex	DOUBLE COMPLEX
enum	INTEGER
opaque	INTEGER*8

8.3 Calling Methods From FORTRAN 77

All SIDL methods are implemented as FORTRAN 77 subroutines regardless of whether they have a return value or not. For object methods, the object or interface pointer is passed as the first argument to the subroutine before all the formally declared arguments. The exception is static methods, where the object or interface pointer does not appear in the argument list at all.

When a method has a return value, a variable to hold the return value should be passed as an argument following the formally declared arguments. This extra argument behaves like an `out` parameter.

The name of the subroutine that FORTRAN 77 clients should call is derived from the fully qualified name of the class and the name(s) of the method. If the method is specified as overloaded (i.e., has a name extension), the method's full name will be used. That is, the concatenation of the short name and the name extension will be used for a unique method name. Hence, to determine the subroutine name for FORTRAN 77, take the fully qualified name, replace all the periods with underscores, append an underscore, append the short method name, append the method name extension (if any) and then append "_f".

For example, to call the `deleteRef()` method on a *sidl.BaseInterface* interface, you would write:

```
        integer*8 interface1, interface2
        logical areSame
C       code to initialize interface1 & interface 2 here
        call sidl_BaseInterface_deleteRef_f(interface1)
```

To call the isSame method on a sidl.BaseInterface , you would write:

```
        call sidl_BaseInterface_queryInt_f(interface1, 'My.Interface.Name', interface2)
```

To call the queryInt method on a sidl.BaseInterface, you would write:

```
       call sidl_BaseInterface_queryInt_f(interface1, 'My.Interface.Name', interface2)
```

Examples of calls to SIDL overloaded methods are based on the `overload_sample.sidl` file shown in Section 5.6. Recall that the file describes three versions of the `getValue` method. The first takes no arguments, the second takes an integer argument, and the third takes a boolean. Each is called in the code snippet below:

```
integer*8 t
logical b1, bretval
integer*4 i1, iretval

call Overload_Sample__create_f (t)

call Overload_Sample_getValue_f (t, iretval)
call Overload_Sample_getValueInt_f (t, i1, iretval)
call Overload_Sample_getValueBool_f (t, b1, bretval)
```

For interfaces and classes, there are two implicit methods called `_cast()` and `_cast2()`. Both of these methods are used to convert from one type to another, and each can be used for upcasting up downcasting. Neither method will increment the reference count of the object.

`_cast()` is a static method. It tries to convert its opaque argument to the type of the class indicated by the method name. For example, `x_y_z__cast(obj, xyz)` will try to convert `obj` to type *x.y.z*. If `xyz` is nonzero, the cast was successful.

`_cast2()` is an object method. Its return type is opaque, and it has one formal argument, a string in addition to the implicit object/interface reference. The `_cast()` method attempts to cast the object/interface to the named type. It is similar to the queryInt method in *sidl.BaseInterface* except it does not increment the reference count of the return object or interface, and it may return an object or an interface pointer. The `queryInt()` method always returns an interface pointer.

For non-abstract classes, there is an implicit method called `_create()`. It creates and returns an instance of the class.

Here are examples of the use of these two methods:

```
      integer*8 object, interface
      call sidl_BaseClass__create_f(object)
      call sidl_BaseInterface__cast_f(object, interface)
c     the following call to _cast2 is equivalent to the previous _cast call
      call sidl_BaseClass__cast2_f(object, 'SIDL.BaseInterface',
     $     interface)
```

Please note the presence of two underscores between BaseClass and create and between BaseClass and cast; the extra underscore is there because the first character of the method name is an underscore.

Here is an example call to the `addSearchPath()` in the *sidl.Loader* class:

```
      call sidl_Loader_addSearchPath_f('/try/looking/here')
```

Your FORTRAN 77 must manage any object references created by the calls you make.

8.4 Catching and Throwing Exceptions in FORTRAN 77

When a method can throw an exception (i.e., its SIDL definition has a throws clause), an extra variable of type `INTEGER*8` should be passed to hold a pointer if an exception is thrown. For maximum backward compatibility, the base exception type argument is *sidl.BaseInterface* though the base exception class is *sidl.SIDLException*. The exception argument appears after the return value when both occur in a method. After the call, the client must test this argument. If a function does not test the exception argument, thrown exceptions will be utterly ignored — not propagated to higher level functions. If the exception parameter is non-zero, an exception was thrown by the method, and the method should respond appropriately. When an exception is thrown, the value of all other arguments is undefined.

When the exception parameter is non-zero, your code should try casting it to each of the possible exceptions in turn. A successful cast indicates the type of exception that has occured. If one of the possible exception types is a subclass of another one, you should try casting to the subclass before casting to the super class — assuming that the distinction between the two exception types results in different exception recovery behavior.

`sidl.BaseException` defines two methods that can be helpful when reporting exceptions to end users: `getNote` and `getTrace`. `getNote` often provides some indication of what went wrong. Its contents are provided by the implementor of the function you called, so it can be empty. Similarly, `getTrace` make provide a summary of the call stack. Again, it depends on implementors to provide information.

One approach to exception handling is to pass the exception on to your caller. In this case, you should call `sidl.BaseException.add` to add another line in the stack trace for the exception.

Here is another example adapted from the Babel regression tests. Package ExceptionTest has a class named Fib with a method declared in SIDL as follows:

```
int getFib(in int n, in int max_depth, in int max_value, in int depth)
    throws NegativeValueException, FibException;
```

Here is the outline of a FORTRAN 77 code fragment to use this method. When an exception is thrown, the value of the `out` and `inout` parameters is unknown, the best practice is to ignore their values.

```
       integer*8 fib, except, except2
        integer*4 index, maxdepth, maxval, depth, result
        call ExceptionTest_Fib__create_f(fib)
        index = 4
        maxdepth = 100
        maxvalue = 32000
        depth = 0
        call ExceptionTest_getFib_f(fib, index, maxdepth,
      $       maxvalue, depth, result, except)
        if (except .ne. 0) then
          call ExceptionTest_FibException__cast_f(except, except2)
          if (except2 .ne. 0) then
c            do something here with the FibException
          else
            call ExceptionTest_NegativeValueException__cast_f
      $          (exception, except2)
c           do something here with the NegativeValueException
          endif
          call sidl_BaseException_deleteRef_f(except)
        else
          write (*,*) 'getFib for ', index, ' returned ', result
        endif
        call ExceptionTest_Fib_deleteRef_f(fib)
```

Here is an example of FORTRAN 77 code that throws an exception.

```
         subroutine ExceptionTest_Fib_getFib_fi(self, n, max_depth,
      &     max_value, depth, retval, exception)
         implicit none
         integer*8 self, exception
         integer*4 n, max_depth, max_value, depth, retval
C        DO-NOT-DELETE splicer.begin(ExceptionTest.Fib.getFib)
         character*(*) myfilename
         parameter(myfilename='ExceptionTest_Fib_Impl.f')
C ...lines of code deleted...
         if (n .lt. 0) then
           call ExceptionTest_NegativeValueException__create_f(exception)
           if (exception .ne. 0) then
             call ExceptionTest_NegativeValueException_setNote_f(
      $             exception,
      $             'called with negative n')
             call ExceptionTest_NegativeValueException_add_f(
      $             exception,
      $             myfilename,
      $             57,
      $             'ExceptionTest_Fib_getFib_impl')
             return
           endif
C ...lines of code deleted...
C        DO-NOT-DELETE splicer.end(ExceptionTest.Fib.getFib)
         end
```

Please note that when your code throws an exception it should `deleteRef` any references it was planning to return to its caller. Any caller of a method that returns an exception should ignore the values of `out` and `inout` parameters, so anything you do not free will become a reference and memory leak. In general, it is good practice to set all `out` and `inout` array, class and interface arguments before returning when throwing an exception. This makes things work out better for clients who forget to check if an exception occurred or willfully choose to ignore it.

8.5 Invoking Babel to generate FORTRAN 77 Stubs

Here is how you should invoke Babel to create the FORTRAN 77 stubs for an IDL file [1].

```
% babel --client=f77 file.sidl
```

or simply

```
% babel -c=f77 file.sidl
```

This will create a babel.make file, numerous C headers, numerous C source files, and some FORTRAN 77 files. The files ending in `_fStub.c` are the FORTRAN 77 stubs that allow FORTRAN 77 to call a SIDL method.

You will need to compile and link the files ending in `_fStub.c` into your application (i.e. `STUBSRCS` in `babel.make`). Normally, the IOR files (`_IOR.c`) are linked together with the implementation file, so you probably don't need to compile them.

If you have some *enum*'s defined in your SIDL file, Babel will generate FORTRAN 77 include files in the style of DEC FORTRAN (Compaq FORTRAN? (now HP Fortran???)) `%INCLUDE`. These files are named by taking the fully qualified name of the *enum*, changing the periods to underscores, and appending `.inc` . Here is an example of a generated include file.

```
C       File:          enums_car.inc
C       Symbol:        enums.car-v1.0
C       Symbol Type:   enumeration
C       Babel Version: 0.5.0
C       Description:   Automatically generated; changes will be lost
C
C       babel-version = 0.5.0
C       source-line   = 25
C
        integer porsche
        parameter (porsche = 911)
        integer ford
        parameter (ford = 150)
        integer mercedes
parameter (mercedes = 550)
```

8.6 Implementing Classes in FORTRAN 77

Much of the information from the previous section is pertinent to implementing a SIDL class in FORTRAN 77. The types of the arguments are as indicated in Table 8.1. Your implementation can call other SIDL methods in which case follow the rules for client calls.

You should invoke Babel:

```
% babel --server=f77 file.sidl
```

or simply

[1] For information on additional command line options, refer to Section 3.2.

```
% babel -s=f77 file.sidl
```

This will create a babel.make, numerous C headers, numerous C source files and some FORTRAN 77 source files. Your job is to fill in the FORTRAN 77 source files with the implementation of the methods. The files you need to edit all end with `_Impl.f`. All your changes to the file should be made between code splicer pairs. Code between splicer pairs will be retained by subsequent invocations of Babel; code outside splicer pairs is not. Here is an example of a code splicer pair. In this example, you would replace the line "C Insert extra code here... " with your lines of code.

```
C       DO-NOT-DELETE splicer.begin(_miscellaneous_code_start)
C       Insert extra code here...
C       DO-NOT-DELETE splicer.end(_miscellaneous_code_start)
```

Each _Impl.f file contains numerous empty subroutines. Each subroutine that you must implement is partially implemented. The `SUBROUTINE` statement is written, and the types of the arguments have been declared. You must provide the body of each subroutine that implements the expected behavior of the method.

There are two implicit methods (i.e. methods that did not appear in the SIDL file) that must also be implemented. The `_ctor()` method is a constructor function that is run whenever an object is created. It's purpose is to initialize the object to make it ready for any of the other method calls. The `_dtor()` method is a destructor function that is run whenever an object is destroyed. The destructor's purpose is to free any resources allocated by the object. If the object has no state, these functions are typically empty.

The SIDL IOR keeps a pointer (i.e. C void *) for each object that is intended to hold a pointer to the object's internal data. The FORTRAN 77 skeleton provides two functions that the FORTRAN 77 developer will need to use to access the private pointer. The name of the function is derived from the fully qualified type name as follows. Replace periods with underscores and append `__get_data_f` or `__set_data_f`. The first argument is the object pointer (i.e. self), and the second argument is an opaque . These arguments are 64 bit integers in FORTRAN 77, but the number of bits actually stored by the IOR is determined by the sizeof(void *).

Babel/SIDL does not provide a low level mechanism for FORTRAN 77 to allocate memory to use for the private data pointer; however, there is an example of a FORTRAN 77 object with state in Section 8.8.

8.7 Accessing SIDL Arrays From FORTRAN 77

For FORTRAN 77, the difference in how you access normal SIDL arrays and r-arrays is profound. Normal SIDL arrays are passed in as an `integer*8`, and you either access them using an function API or by converting the array data to a index into a known array. R-arrays appear like normal FORTRAN 77 arrays, so there is a big incentive to use r-arrays unless you cannot.

The client-side interface for the `solve` example introduced in Section 5.4 behaves as if it is a FORTRAN 77 function with the following declarations:

```
        subroutine num_Linsol_solve_f(self, A, x, b, m, n)
        implicit none
C       in num.Linsol self
        integer*8 self
        integer*4 m, n
C       in rarray<double,2> A(m,n)
        double precision A(0:m-1, 0:n-1)
C       inout rarray<double> x(n)
        double precision x(0:n-1)
C       in rarray<double> b(m)
        double precision b(0:m-1)
        end
```

FORTRAN 77 programmers should note that the array indices go from 0 to $m-1$ instead of the normal 1 to m. This is a concession to the C and C++ programmers who have to deal with the fact that A is stored in column-major order.

On the server-side, the interface for `solve` appears as follows:

```
      subroutine num_Linsol_solve_fi(self, A, x, b, m, n)
      implicit none
C     in num.Linsol self
      integer*8 self
C     in int m
      integer*4 m
C     in int n
      integer*4 n
C     in rarray<double,2> A(m,n)
      double precision A(0:m-1, 0:n-1)
C     inout rarray<double> x(n)
      double precision x(0:n-1)
C     in rarray<double> b(m)
      double precision b(0:m-1)

C     DO-NOT-DELETE splicer.begin(num.Linsol.solve)
C     Insert the implementation here...
C     DO-NOT-DELETE splicer.end(num.Linsol.solve)
      end
```

Note again that the array indices go from 0 to m−1. The implementation should avoid changing the data in `in` parameters.

The remainder of this section is dedicated to how you access normal SIDL arrays. The normal SIDL C function API is available from FORTRAN 77 to create, destroy and access array elements and meta-data. The function name for FORTRAN has `_f` appended.

For SIDL types dcomplex, double, fcomplex , float, int and long, SIDL provides a method to get direct access to the array elements. For the other types, you must use the functional API to access array elements.

For type X, there is a FORTRAN 77 function called `sidl_X__array_access_f` to provide a method to get direct access. An example is given below. Of course, this will not work if your FORTRAN 77 compiler does array bounds checking.

```
      integer*4 lower(1), upper(1), stride(1), i, index(1)
      integer*4 value, refindex, refarray(1), modval
      integer*8 nextprime, tmp
      lower(1) = 0
      value = 0
      upper(1) = len - 1
      call sidl_int__array_create_f(1, lower, upper, retval)
      call sidl_int__array_access_f(retval, refarray, lower,
     $     upper, stride, refindex)
      do i = 0, len - 1
         tmp = value
         value = nextprime(tmp)
         modval = mod(i, 3)
         if (modval .eq. 0) then
            call sidl_int__array_set1_f(retval, i, value)
         else
            if (modval .eq. 1) then
               index(1) = i
               call sidl_int__array_set_f(retval, index, value)
            else
C this is equivalent to the sidl_int__array_set_f(retval, index, value)
               refarray(refindex + stride(1)*(i - lower(1))) =
     $              value
            endif
         endif
      enddo
```

To access a two dimensional array, the expression referring to element i, j is

```
refarray(refindex + stride(1) * (i - lower(1)) + stride(2) * (j - lower(2))
```

To access a three dimensional array, the expression referring to element i, j k is

```
refarray(refindex + stride(1) * (i - lower(1)) + stride(2) * (j - lower(2))
```

You can call things like LINPACK or BLAS if you want, but you should check the stride to make sure the array is packed as you need it to be. `stride(i)` indicates the distance between elements in dimension `i`. A value of 1 means elements are packed densely in dimension `i`. Negative stride values are possible, and when an array is a slice of another array, there may be no dimension with a stride of 1.

For a *dcomplex* array, the reference array should a FORTRAN array of `REAL*8` instead of a FORTRAN array of double complex to avoid potential alignment problems. For a *fcomplex* array, the reference array is a `COMPLEX*8` because we don't anticipate an alignment problem in this case.

8.8 FORTRAN 77 objects with state

If you need to implement a FORTRAN 77 class with state, you can use SIDL arrays to store the state information. This is certainly not the only way to implement a FORTRAN 77 class with state, but it's one that will work wherever Babel works. For example, if you have a class whose state requires three boolean variables and two double precision variables, your constructor might look something like the following:

```
      subroutine example_withState__ctor_fi(self)
      implicit none
      integer*8 self
C     DO-NOT-DELETE splicer.begin(example.withState._ctor)
      integer*8 statearray, logarray, dblarray
      call sidl_opaque__array_create1d_f(2, statearray)
      call sidl_bool__array_create1d_f(3, logarray)
      call sidl_double__array_create1d_f(2, dblarray)
      if ((statearray .ne. 0) .and. (logarray .ne. 0) .and.
     $      (dblarray .ne. 0)) then
         call sidl_opaque__array_set1_f(statearray, 0, logarray)
         call sidl_opaque__array_set1_f(statearray, 1, dblarray)
      else
C        a real implementation would not leak memory like this one
         statearray = 0
      endif
      call example_withState__set_data_f(self, statearray)
C     DO-NOT-DELETE splicer.end(example.withState._ctor)
      end
```

Of course, it is up to your application make the association between elements of the arrays and particular state variables. For example, you could say that element 0 of the double array is the kinematic viscosity and element 1 could be the airspeed velocity of an unladen swallow. Element 0 of the boolean array could specify African (true) or European (false). The destructor for this class could look something like this:

```
      subroutine example_withState__dtor_fi(self)
      implicit none
      integer*8 self
C     DO-NOT-DELETE splicer.begin(example.withState._dtor)
      integer*8 statearray, logarray, dblarray
      call example_withState__get_data_f(self, statearray)
      if (statearray .ne. 0) then
         call sidl_opaque__array_get1_f(statearray, 0, logarray)
         call sidl_opaque__array_get1_f(statearray, 1, dblarray)
         call sidl_bool__array_deleteRef_f(logarray)
         call sidl_double__array_deleteRef_f(dblarray)
         call sidl_opaque__array_deleteRef_f(statearray)
```

```
C       the following two lines are not strictly necessary
           statearray = 0
           call example_withState__set_data_f(self, statearray)
        endif
C       DO-NOT-DELETE splicer.end(example.withState._dtor)
        end
```

In this example, an accessor function for the airspeed velocity of an unladen swallow could be implemented as follows:

```
        subroutine example_withState_getAirspeedVelocity_fi(
     $     self, velocity)
        implicit none
        integer*8 self
        real*8 velocity
C       DO-NOT-DELETE splicer.begin(example.withState.getAirspeedVelocity)
        integer*8 statearray, dblarray
        call example_withState__get_data_f(self, statearray)
        if (statearray .ne. 0) then
           call sidl_opaque__array_get1_f(statearray, 1, dblarray)
           call sidl_double__array_get1_f(dblarray, 1, velocity)
        endif
C       DO-NOT-DELETE splicer.end(example.withState.getAirspeedVelocity)
        end
```

Chapter 9

FORTRAN 90 Bindings

Contents

9.1 Introduction

This chapter provides an introduction to the FORTRAN 90 bindings supported by Babel. Software written in FORTRAN 90 that illustrates both the caller, or client, side as well as the callee, or server side, is provided.

For ease of comparison, this chapter is patterned after the chapter on FORTRAN 77 bindings. Further, the initial support described below is very similar to that provided for FORTRAN 77.

9.2 Basic Types

The mapping for simple SIDL types to FORTRAN 90 is given in Table 9.1. For opaque pointers, the equivalent of a SIDL double is used. That is, the intermediate object reference assumes a 64-bit integer is used to enable portability between systems with a 32 bit address space and those with a 64 bit address space. On a 32 bit system, the upper 32 bits of these quantities are ignored. Systems with more than 64-bit pointers aren't currently supported.

For interfaces, classes and arrays, there is a derived type that holds an opaque pointer. The derived type for arrays of numeric types also has a F90 pointer to an array to provide native array access without function calls. For each interface and class, there are two modules created. In the first module, the derived type for the object and array are defined. In the second, the methods for the object/interface and arrays of the object/interface are defined. Clients of a class or interface, typically `use` the module containing the methods, and it in turn uses the module containing the types.

Generally, clients should treat opaque, interface, class and array values as black boxes. However, there is one value that is special. A value of zero for any of these quantities indicates that the value does not refer to an object. Zero is the equivalent of NULL. Any nonzero value is or should be a valid object reference. The method module provides functions to test whether an interface, class or array value `is_null` or is `not_null`. There is also a subroutine to initialize the value to indexFORTRAN 90!set_null`set_null`. Clients should generally initialize new interface or class values to NULL.

Table 9.1: SIDL to FORTRAN 90 type mapping

SIDL TYPE	FORTRAN 90 TYPE
int	INTEGER (SELECTED_INT_KIND(9))
long	INTEGER (SELECTED_INT_KIND(18))
float	REAL (SELECTED_REAL_KIND(6,37))
double	REAL (SELECTED_REAL_KIND(15, 307))
bool	LOGICAL
char	CHARACTER (LEN=1)
string	CHARACTER (LEN=*)
fcomplex	COMPLEX (SELECTED_REAL_KIND(6, 37))
dcomplex	COMPLEX (SELECTED_REAL_KIND(15, 307))
enum	INTEGER (SELECTED_INT_KIND(9))
opaque	INTEGER (SELECTED_INT_KIND(18))

The SIDL string type mapping is currently identical to that of the FORTRAN 77 mapping. That is, all FORTRAN 90 strings have a limited fixed size. When implementing a subroutine with an out parameter, the size of the string is limited to 512 characters. This can be changed when configuring babel by changing the value of `SIDL_F90_STR_MINSIZE` in `runtime/sidl/babel_config.h` before compiling and installing babel.

Enumerated types are just integer values. The integer parameters are defined in a module.

9.3 Calling Methods From FORTRAN 90

All SIDL methods are implemented as FORTRAN 90 subroutines regardless of whether they have a return value or not. For object methods, the object or interface pointer is passed as the first argument to the subroutine before all the formally declared arguments. The exception is static methods, where the object or interface pointer does not appear in the argument list at all.

When a method has a return value, a variable to hold the return value should be passed as an argument following the formally declared arguments.

The name of the module that holds the method definitions is derived from the fully qualified name of the class or interface. You can generate the module name by replacing all the periods with underscores. For example, the methods for *sidl.SIDLException* are defined in a module named `sidl_SIDLException` in the file `sidl_SIDLException.F90`. The name of the module holding the derived type of the class or interface is the same as the one holding the methods except that it has `_type` appended to it. The types for *sidl.SIDLException* are called `sidl_SIDLException_t` and `sidl_SIDLException_a`, for the array, and they are defined in the file `sidl_SIDLException_type.F90`.

The name of the subroutine that FORTRAN 90 clients is the method's full name from the SIDL description. If the method is specified as overloaded (i.e., has a name extension), the method's full name will be used. That is, the concatenation of the short name and the name extension will be used for a unique method name.

For example, to call the `deleteRef()` method on a *SIDL.BaseInterface* interface, you would write:

```
use sidl_BaseInterface
type(sidl_BaseInterface_t) :: interface1, interface2
logical                    :: areSame
!
! code to initialize interface1 & interface 2 here
!
call deleteRef(interface1)
```

To call the isSame method on a *sidl.BaseInterface*, you would write:

```
use sidl_BaseInterface
! later in the code
call isSame(interface1, interface2, areSame)
! areSame holds the return value
```

To call the queryInt method on a *`sidl.BaseInterface`*, you would write:

```
use sidl_BaseInterface
! later
call queryInt(interface1, 'My.Interface.Name', interface2)
! interface2 holds the return value now
```

Examples of calls to SIDL overloaded methods are based on the `overload_sample.sidl` file shown in Section 5.6. Recall that the file describes three versions of the `getValue` method. The first takes no arguments, the second takes an integer argument, and the third takes a boolean. Each is called in the code snippet below:

```
use Overload_Sample
type(Overload_Sample_t)         :: t
logical                         :: b1, bretval
integer (selected_int_kind(9))  :: i1, iretval

call new(t)

call getValue (t, iretval)
call getValueInt (t, i1, iretval)
call getValueBool (t, b1, bretval)
```

Here is an example of what Babel will produce for an enumerated type with some of the whitespace and comments reduced for brevity.

```
! File:          enums_car.F90
! Symbol:        enums.car-v1.0
! Symbol Type:   enumeration
! Babel Version: 0.8.2
! Description:   Client-side module for enums.car

module enums_car
! Symbol "enums.car" (version 1.0)

  integer (selected_int_kind(9)), parameter :: porsche = 911
  integer (selected_int_kind(9)), parameter :: ford = 150
  integer (selected_int_kind(9)), parameter :: mercedes = 550
end module enums_car
```

For interfaces and classes, there is an implicit method called `cast()`. There are actually a set of overloaded methods that support every allowable cast between a type an all its parent types (objects and interfaces). The first argument is the object/interface to be cast, and the second argument is a variable of the desired type. If the value of the second argument after the call is `not_null`, the cast was successful; otherwise, the cast failed. `cast()` is similar to the queryInt method in *`sidl.BaseInterface`* except it does not increment the reference count of the return object or interface, and it may return an object or an interface pointer. The `queryInt()` method always returns an interface pointer.

For non-abstract classes, there is an implicit method called `new()`. It creates and returns an instance of the class.

Here are examples of the use of these two methods:

```
use sidl_BaseClass
use sidl_BaseInterface
type(sidl_BaseClass_t)     :: object
type(sidl_BaseInterface_t) :: interface
! perhaps other code here
call new(object)
call cast(object, interface)
```

Here is an example call to the `addSearchPath()`, a static method, in the *`sidl.Loader`* class:

```
use sidl_Loader
! later
call addSearchPath('/try/looking/here')
```

Your FORTRAN 90 must manage any object references created by the calls you make.

9.4 Catching and Throwing Exceptions in Fortran 90

When a method can throw an exception (i.e., its SIDL definition has a throws clause), a variable should be passed to hold an exception. For maximum backward compatibility, the exception argument type is a *`sidl.BaseInterface`* pointer that is assumed to implement the *`sidl.BaseException`* interface. The exception argument should appear after the return value when both occur in a method, and it behaves like an `out` parameter. After the call, the client should test this argument using `is_null` or `not_null`. If it is `not_null`, an exception was thrown by the method, and the method should respond appropriately. When an exception is thrown, the value of all other arguments is undefined, and the best course of action is to ignore their values. If your code does not check the exception argument after each call that can throw an exception, any exceptions that are thrown will be utterly ignored; they will not propagate automatically to higher level routines.

Here is another example adapted from the Babel regression tests. Package ExceptionTest has a class named Fib with a method declared in SIDL as follows:

```
int getFib(in int n, in int max_depth, in int max_value, in int depth)
    throws NegativeValueException, FibException;
```

Here is the outline of a FORTRAN 90 code fragment to use this method.

```
  use ExceptionTest_Fib
  use ExceptionTest_FibException
  use ExceptionTest_NegativeValueException
  use sidl_BaseInterface
  type(ExceptionTest_Fib_t)                    :: fib
  type(sidl_BaseInterface_t)                   :: except
  type(ExceptionTest_FibException_t)           :: fibexcept
  type(ExceptionTest_NegativeValueException_t) :: nvexcept
  integer (selected_int_kind(9))  :: index, maxdepth, maxval, depth, result
  call new(fib)

  index    = 4
  maxdepth = 100
  maxvalue = 32000
  depth    = 0
  call getFib(fib, index, maxdepth, maxvalue, depth, result, except)
  if (not_null(except)) then
    call cast(except, fibexcept)
    if (not_null(fibexcept)) then
!      do something here with the FibException
    else
      call cast(except, nvexcept)
!     do something here with the NegativeValueException
    endif
    call deleteRef(except)
  else
    write (*,*) 'getFib for ', index, ' returned ', result
  endif
  call deleteRef(fib)
```

Here is an example of an implementation subroutine that throws an exception. Note you must `cast` the returned exception object into the exception `out` parameter. The `setNote` method provides a useful error message, and the `add` method helps provide a multi-language traceback capability (provided each layer of the call stack calls `add`).

```
recursive subroutine ExceptionTest_Fib_getFib_mi(self, n, max_depth, &
  max_value, depth, retval, exception)
  use sidl_BaseInterface
  use ExceptionTest_Fib
  use ExceptionTest_NegativeValueException
```

```
    use ExceptionTest_FibException
    use ExceptionTest_Fib_impl
    ! DO-NOT-DELETE splicer.begin(ExceptionTest.Fib.getFib.use)
    use ExceptionTest_TooBigException
    use ExceptionTest_TooDeepException
    ! DO-NOT-DELETE splicer.end(ExceptionTest.Fib.getFib.use)
    implicit none
    type(ExceptionTest_Fib_t) :: self
    integer (selected_int_kind(9)) :: n, max_depth, max_value
    integer (selected_int_kind(9)) :: retval, depth
    type(sidl_BaseInterface_t) :: exception
  ! DO-NOT-DELETE splicer.begin(ExceptionTest.Fib.getFib)
    type(ExceptionTest_NegativeValueException_t) :: negexc
  ! ...lines deleted...
    character (len=*) myfilename
    parameter(myfilename='ExceptionTest_Fib_Impl.f')
    retval = 0
    if (n .lt. 0) then
       call new(negexc)
       if (not_null(negexc)) then
          call setNote(negexc, &
               'called with negative n')
          call add(negexc, myfilename, 57, &
                   'ExceptionTest_Fib_getFib_impl')
          call cast(negexc, exception)
          return
       endif
    else
  ! ...numerous lines deleted....
  ! DO-NOT-DELETE splicer.end(ExceptionTest.Fib.getFib)
  end subroutine ExceptionTest_Fib_getFib_mi
```

Please note that when your code throws an exception it should `deleteRef` any references it was planning to return to its caller. Any caller of a method that returns an exception should ignore the values of `out` and `inout` parameters, so anything you do not free will become a reference and memory leak. In general, it is good practice to call `set_null` on all `out` and `inout` array, class and interface arguments before returning when throwing an exception. This makes things work out better for clients who forget to check if an exception occurred or willfully choose to ignore it.

9.5 Invoking Babel to Generate F90 Stubs

Here is how you should invoke Babel to create the FORTRAN 90 stubs for an IDL file [1].

```
% babel --client=f90 file.sidl
```

or simply

```
% babel -c=f90 file.sidl
```

This will create a babel.make file, numerous C headers, numerous C source files, and some FORTRAN 90 files. The files ending in `_fStub.c` are called by the FORTRAN 90 module which in turn allow FORTRAN 90 to call a SIDL method. The files ending in `_type.F90` contain derived type definitions for classes and interfaces., and the other files ending in `.F90` are FORTRAN 90 modules containing methods.

[1] For information on additional command line options, refer to Section 3.2.

You will need to compile and link the files ending in _fStub.c (i.e., STUBSRCS in babel.make) and all the files ending in .F90 (i.e., STUBMODULESRCS and TYPEMODULESRCS in babel.make) into your application. Normally, the IOR files (_IOR.c) are linked together with the implementation file, so you probably don't need to compile them.

9.6 Implementing Classes in FORTRAN 90

Much of the information from the previous section is pertinent to implementing a SIDL class in FORTRAN 90. The types of the arguments are as indicated in Table 9.1. Your implementation can call other SIDL methods in which case follow the rules for client calls.

You should invoke Babel:

```
% babel --server=f90 file.sidl
```

or simply

```
% babel -s=f90 file.sidl
```

This will create a babel.make, numerous C headers, numerous C source files and some FORTRAN 90 source files. Your job is to fill in the FORTRAN 90 source files with the implementation of the methods. The files you need to edit all end with _Impl.F90 and _Mod.F90. All your changes to the file should be made between code splicer pairs. Code between splicer pairs is retained by subsequent invocations of Babel; code outside splicer pairs is not.

Here is an example of the standard code splicer pairs in generated FORTRAN 90 code. You would replace the comment ”Insert extra code here... ” associated with the ”miscellaneous code start” splicer pair with code needed for your implementation such as additional abbreviation file(s) and any local, or private, subroutines. For the subroutine's ”use” splicer pair, you would replace the ”Insert use statements here...” comment with any use statements that are needed by the subroutine. Finally, you would add the implementation between the subroutine body's splicer pairs in the place of the ”Insert the implementation here...” comment.

```
! DO-NOT-DELETE splicer.begin(_miscellaneous_code_start)
! Insert extra code here...
! DO-NOT-DELETE splicer.end(_miscellaneous_code_start)

.
.
.

subroutine Pkg_Class_name_mi(args)
  ! DO-NOT-DELETE splicer.begin(Pkg.Class.name.use)
  ! Insert use statements here...
  ! DO-NOT_DELETE splicer.end(Pkg.Class.name.use)
  implicit none
  integer (selected_int_kind(18)) :: arg

! DO-NOT-DELETE splicer.begin(Pkg.Class.name)
! Insert the implementation here...
! DO-NOT-DELETE splicer.end(Pkg.Class.name)
```

Each _Impl.F90 file contains numerous partially implemented subroutines. The SUBROUTINE and END SUBROUTINE statements have been generated and the types of the arguments declared. As mentioned above, you must provide any needed use statements and the body of each subroutine to implement the expected behavior of the method.

There are two implicit methods (i.e., methods that did not appear in the SIDL file) that must also be implemented if the object is to have state (i.e., data associated with the instance). The _ctor() method is a constructor function that is run whenever an object is created. The _dtor() method is a destructor function that is run whenever an object is destroyed. If there is not state then these functions are typically empty.

The SIDL IOR keeps a pointer for each object that is intended to hold a pointer to the object's internal data. The FORTRAN 90 skeleton provides two functions that the FORTRAN 90 developer will need to use to access the private pointer. The name of the function is derived from the fully qualified type name by replacing periods with underscores and appending `__get_data_m` or `__set_data_m`. The first argument is the object pointer (i.e., self), and the second is a derived type defined in the `_Mod.F90` file. Here is an excerpt from a `_Mod.F90` file for an object whose state requires a single integer value.

```
#include"sort_SimpleCounter_fAbbrev.h"
module sort_SimpleCounter_impl

type sort_SimpleCounter_private
   sequence
  ! DO-NOT-DELETE splicer.begin(sort.SimpleCounter.private_data)
   integer(selected_int_kind(9)) :: count
  ! DO-NOT-DELETE splicer.end(sort.SimpleCounter.private_data)
end type sort_SimpleCounter_private

type sort_SimpleCounter_wrap
   sequence
   type(sort_SimpleCounter_private), pointer :: d_private_data
end type sort_SimpleCounter_wrap

end module sort_SimpleCounter_impl
```

The derived type `sort_SimpleCounter_private` is the type where the developer adds data to store the object's state, and `sort_SimpleCounter_wrap` exists simply to facilitate transferring the pointer to a `sort_SimpleCounter_private` to and from the IOR.

Typically for a class with state, the developer needs to `allocate(pd%d_private_data)` in the constructor, `_ctor`, and `deallocate(pd%d_private_data)` in the destructor, `_dtor`. Here is a concrete example of a constructor.

```
recursive subroutine sort_SimpleCounter__ctor_mi(self)
  use sort_SimpleCounter
  use sort_SimpleCounter_private
  ! DO-NOT-DELETE splicer.begin(sort.SimpleCounter._ctor.use)
  ! DO-NOT-DELETE splicer.end(sort.SimpleCounter._ctor.use)
  implicit none
  type(sort_SimpleCounter_t) :: self

! DO-NOT-DELETE splicer.begin(sort.SimpleCounter._ctor)
  type(sort_SimpleCounter_wrap) :: dp
  allocate(dp%d_private_data)
  dp%d_private_data%count = 0
  call sort_SimpleCounter__set_data_m(self, dp)
! DO-NOT-DELETE splicer.end(sort.SimpleCounter._ctor)
end subroutine sort_SimpleCounter__ctor_mi
```

Here is the corresponding destructor.

```
recursive subroutine sort_SimpleCounter__dtor_mi(self)
  use sort_SimpleCounter
  use sort_SimpleCounter_private
  ! DO-NOT-DELETE splicer.begin(sort.SimpleCounter._dtor.use)
  ! DO-NOT-DELETE splicer.end(sort.SimpleCounter._dtor.use)
  implicit none
  type(sort_SimpleCounter_t) :: self

! DO-NOT-DELETE splicer.begin(sort.SimpleCounter._dtor)
  type(sort_SimpleCounter_wrap) :: dp
```

```
    call sort_SimpleCounter__get_data_m(self, dp)
    deallocate(dp%d_private_data)
  ! DO-NOT-DELETE splicer.end(sort.SimpleCounter._dtor)
  end subroutine sort_SimpleCounter__dtor_mi
```

9.7 Accessing SIDL Arrays From FORTRAN 90

SIDL r-arrays are passed to and from methods as normal FORTRAN 90 arrays. You do not need to include the index variables because the values are determined from the FORTRAN 90 array extents in each dimension.

The client-side interface for the `solve` example introduced in Section 5.4 behaves as if it is a FORTRAN 77 function with the following overloaded interface:

```
private :: solve_1s, solve_2s
interface solve
  module procedure solve_1s, solve_2s
end interface

recursive subroutine solve_1s(self, A, x, b)
  implicit none
  type(num_Linsol_t) , intent(in) :: self ! in num.Linsol self
  ! in array<double,2,column-major> A
  type(sidl_double_2d) , intent(in) :: A
  ! inout array<double,column-major> x
  type(sidl_double_1d) , intent(inout) :: x
  ! in array<double,column-major> b
  type(sidl_double_1d) , intent(in) :: b
  ! details deleted
end subroutine solve_1s

recursive subroutine solve_2s(self, A, x, b)
  implicit none
  type(num_Linsol_t) , intent(in) :: self ! in num.Linsol self
  ! in rarray<double,2> A(m,n)
  real (selected_real_kind(15, 307)) , intent(in), dimension(:, :) :: A
  ! inout rarray<double> x(n)
  real (selected_real_kind(15, 307)) , intent(inout), dimension(:) :: x
  ! in rarray<double> b(m)
  real (selected_real_kind(15, 307)) , intent(in), dimension(:) :: b
  ! details deleted
end subroutine solve_2s
```

You can use either normal FORTRAN 90 arrays or normal SIDL arrays when calling a FORTRAN 90 method, but you cannot use a mixture.

The server-side interface for `solve` is similar. Note, the lower index each dimension of every incoming array is always zero.

```
recursive subroutine num_Linsol_solve_mi(self, A, x, b, m, n)
  use num_Linsol
  use sidl_double_array
  use num_Linsol_impl
  ! DO-NOT-DELETE splicer.begin(num.Linsol.solve.use)
  ! DO-NOT-DELETE splicer.end(num.Linsol.solve.use)
  implicit none
  type(num_Linsol_t) :: self ! in
  integer (selected_int_kind(9)) :: m ! in
  integer (selected_int_kind(9)) :: n ! in
  real (selected_real_kind(15, 307)), dimension(0:m-1, 0:n-1) :: A ! in
  real (selected_real_kind(15, 307)), dimension(0:n-1) :: x ! inout
```

```
    real (selected_real_kind(15, 307)), dimension(0:m-1) :: b ! in

  ! DO-NOT-DELETE splicer.begin(num.Linsol.solve)
  ! Insert the implementation here...
  ! DO-NOT-DELETE splicer.end(num.Linsol.solve)
  end subroutine num_Linsol_solve_mi
```

For normal SIDL arrays, the normal SIDL C function API is available from FORTRAN 90 to create, destroy, and access array elements and meta-data. The array routines are in a module. For *sidl.SIDLException*, the array module is named `sidl_SIDLException_array`, and the array module is defined in the `sidl_SIDLException_array.F90`.

For SIDL types dcomplex, double, fcomplex , float, int, and long, SIDL provides an array pointer to get direct access to the array elements. For the other types, you must use the functional API to access array elements.

The SIDL derived type for a SIDL array is named after the class, interface or basic type that it holds and the dimension of the array. For *sidl.SIDLException*, the array derived types are named `sidl_SIDLException_1d`, `sidl_SIDLException_2d`, `sidl_SIDLException_3d`, . . . up to `sidl_SIDLException_7d`. For the basic types, they are treated as *sidl.dcomplex*, *sidl.double*, *sidl.fcomplex*, etc. Each of these derived types has a 64 bit integer to hold an opaque pointer.

The derived type for SIDL types dcomplex, double, fcomplex , float, int, and long also has a pointer to an array of the appropriate type and dimension. For example, here is the derived type for 2d and 3d arrays of doubles.

```
type sidl_double_2d
  sequence
  integer (selected_int_kind(18)) :: d_array
  real (selected_real_kind(15, 307)), pointer, &
    dimension(:,:) :: d_data
end type sidl_double_2d

type sidl_double_3d
  sequence
  integer (selected_int_kind(18)) :: d_array
  real (selected_real_kind(15, 307)), pointer, &
    dimension(:,:,:) :: d_data
end type sidl_double_3d
```

You can access the array with the F90 array pointer `d_data` just like any other F90 array. However, you *must not* use the F90 builtins `allocate` or `deallocate` on `d_data`. You must use SIDL functions `createCol`, `createRow`, `create1d`, `create2dRow`, `create2dCol` to create a new array. These SIDL routines initialize `d_data` to refer to the data allocated in `d_array`.

You can call things like LINPACK or BLAS if you want, but you should check the stride to make sure the array is packed as needed. You can check `stride(i)`, which indicates the distance between elements in dimension `i`. A value of 1 means elements are packed densely in dimension `i`. Negative stride values are possible. When an array is sliced, the resulting array might not even have one densely packed dimension.

Chapter 10

Java Bindings

Contents

10.1 Introduction

This chapter provides an introduction to the Java bindings for SIDL, including illustrations of both callers and callees written in Java. It shows how to use Babel to wrap the implementation of software written in Java as well as how to call software, possibly implemented in any other supported language, from Java.

10.2 Basic Types

Most SIDL types map directly into Java as shown in Table 10.1.

10.3 Client Side: Using SIDL Classes and Methods

SIDL's object model is very similar to Java's, and therefore maps easily into Java's object model. A SIDL object is treated almost exactly the same in Java as any other Java object, the only difference being that all data held by the object is private, and all methods are public.

Importing SIDL packages and classes is also exactly the same as in Java. For example, assume there is a package `test` that includes the class `HelloWorld`, and you wish to print this message in your program. The following code segment does this.

Table 10.1: SIDL to Java Type Mappings

SIDL TYPE	JAVA TYPE
int	int
long	long
float	float
double	double
bool	boolean
char	char
string	String
fcomplex	FloatComplex
dcomplex	DoubleComplex
enum	Enum
opaque	long
interface	interface
class	class
array	type.Array

```
import test.HelloWorld;

public static main(String args[]) {

  HelloWorld hi = new HelloWorld();
  hi.printMsg();
}
```

Writing the fully qualified class name would also have sufficed. `test.HelloWorld hi = new test.HelloWorld()`

Babel also generates Java code in line with Java's use of directories to organize packages and classes as files. For example, assume you are generating babel code in a directory named `babelcode`. Assume your package `test` contains 2 classes `HelloWorld` and `GoodbyeWorld`. After running `babel -cJava test.sidl` you will have a new directory in `babelcode` named `test` which will contain 2 files, `HelloWorld.java` and `GoodbyeWorld.java`. These classes will be accessible from your Java program as long as `babelcode` is in your `CLASSPATH`.

10.4 Server Side: Writing SIDL classes in Java

Babel also supports calls to SIDL classes implemented in Java. These classes obey the same rules as the client side Java classes, except that is this case the file, class, and method names all end in `_Impl`.

As is the case with other Babel server side files, only the code written between splicer blocks will be preserved between calls of Babel. Make sure any data and code is kept in the designated areas, otherwise it won't be there after you run Babel on those files.

Another interesting fact of the Server Side is that it inherits from the Client Side Java class. This allows us to call local methods directly. Take this recursive Fibonacci function implementation for example:

```
class Fib_Impl extends Fib {
  public int getFib_Impl(int x) {
    // DO-NOT-DELETE splicer.begin(ExceptionTest.Fib.getFib)
    if(x >= 2) {
      return getFib(x-1) + getFib(x-2);
    } else {
      return 1;
    }
    // DO-NOT-DELETE splicer.end(ExceptionTest.Fib.getFib)
  }
}
```

Here the client side class is name `Fib`, and therefore the Server Side class is `Fib_Impl`. The same relation is true for the `getFib` method. You can also see that we are able to call `getFib`, the client side method, directly. A call like this will go through Babel glue code, as it should. You should not try to make calls directly to `_Impl` methods. It won't work at all on different objects, and it breaks the object model if used on methods in the current object. (That is, it is possible to call `foo_Impl` in the current object, but because the call will not go through Babel, any inheritance information will be lost, and the wrong version of the method may be called. Simply call `foo` in the standard way.)

This also means there is no way to have Server Side object inherit from non SIDL Java classes, in fact, there are no splicer blocks available for inheritance, so implementing interfaces on the Server Side is also not supported. This is because we feel that having the Server side inherit from non-SIDL classes is probably not a good idea.

10.5 Casting Objects

In some cases it is necessary to cast the internal representation of an object as well as the Java object. (For example, getting an object from a SIDL array of superclass objects.) In these cases a Java cast is insufficient. Therefore we have provided two casting functions.

`_cast(object)` is a static function included with every SIDL class that returns object passed in to cast that class. For example, in order to cast an object of type `sidl.BaseClass` to `foo.Bar` simply write `foo.Bar newobj = (foo.Bar) foo.Bar._cast(oldobj)`. If this is an invalid cast, `_cast` will return `null`.

The alternative is `_cast2(''ClassName'')`. This is a cast function that is included with every SIDL object. It performs basically the same function as `_cast`, but the form is `object._cast2(''ClassName'')`. It takes a fully qualified class name. If the cast is invalid, or a class of that name cannot be found, this function returns `null`.

Both of these functions return a `sidl.BaseClass` which then must be Java casted to the correct Java class type. Also, in casting, they both create a new Java object that owns a new reference to the IOR object. In Java you never have to worry about reference counting, but this does mean that the pre-cast object still exists and is valid.

10.6 Out and Inout arguments

In C or C++ out and inout arguments are handled by passing pointers to the data so that if the data is changed, the pointer will be pointing to the new, correct, data. Because Java does not support pointers, each SIDL type and class has a static inner `Holder` class. This Holder class can hold a single variable or object of the correct type. There are functions `get()` and `set()` for getting or setting this object.

10.7 Using SIDL arrays with Java

Every object and type defined in SIDL can be put into a SIDL array of that type. Arrays are a fairly complex topic, and the specifics of the Babel Array API are discussed earlier in Section 5.4. Suffice to say that the entire API is available in Java, except for

`ensure`, `borrow`, and `first`, all of which have no real use in Java. `addRef` and `deleteRef` exist in Java, but shouldn't be used, because the Java decrements the reference count itself when it garbage collects a SIDL object or array. If it is necessary to `deleteRef` an array, you should use the `destroy()` array function instead.

More to the point are the specifics of the Java implementation. Each SIDL type and class includes a static inner class named `Array`. This is the main Array class, and in order to support up to 7 dimensional arrays, every method takes either 7 array indices, or an array of indices. For example, in order to get the element (2,3) from a 2 dimensional array, we would type `arry._get(2,3,0,0,0,0,0)`.

Since typing all those zeros can get a little tedious, we also implemented a set of subclasses of Array. One subclass for each dimension. So, if we had and `Array2` instead of an `Array` we could simply type `arry2._get(2,3)` to get the correct element.

These numbered Array subclasses improve on the Array API usability somewhat, but that do have a side effect. In order to avoid conflicts between the Array superclass and the numbered Array subclass functions, all other basic Array methods found in the Array superclass are preceded by an underscore '_'. For example, in order to get an array's dimension, you can type `arry._dim()`. The numbered arrays all inherit these methods, so `arry2._dim()` will also work, although in this case, the answer should be obvious.

Furthermore, there is another underscore rule for Arrays in Java. All numbered arrays have two get and two set functions. The _get and _set functions are the same in Array and all the Array# subclasses, they simply pass the arguments of the _get call down to the underlying implementation. However, the underscore-less get and set do bounds checking in Java before calling the underlying implementation, and, if there is a problem, throw an ArrayIndexOutOfBoundsException.

Because the numbered arrays are subclasses of Array, if necessary you can Java cast an Array# to an Array. However, some functions return an Array. In order to convert an Array to the correctly numbered array, we provided a function in Array called _dcast(). In order to cast an Array object to a numbered array, simply call _dcast() on it. For example, assume we have a 1 dimensional array of type foo.Bar called arry that is represented by the Java class Array. In order to get a correctly numbered array type:

```
foo.Bar.Array1 arry1 = arry._dcast();
```

After this cast we have 2 references to the same array, arry and arry1.

Finally, the Java array constructors are slightly different then they are in other languages. This is the constructor definition for Array.

```
public Array(int dim, int[] lower, int[] upper, boolean isRow)
```

This constructor creates and array of dimension dim. It takes two arrays of integers to define the lower and upper bounds of each dimension in the array. If lower or upper has fewer elements than there are dimensions in the array, or any element in lower is larger than the corresponding element in upper, this constructor will throw an exception. Finally, this constructor takes a boolean isRow. If isRow is true, this constructor will create a SIDL array in row-major order, if it is false, it will create an array in column-major order.

The constructors for numbered arrays are simpler. Here's the constructor for a 2 dimensional array:

```
public Array2( int l0, int l1, int u0, int u1, boolean isRow)
```

The dimension argument is no longer necessary, and it is no longer necessary to create arrays of bounds to pass into the constructor. l0 and l1 are the lower bounds. and u0 and u1 are the upper bounds. This constructor still includes the choice between column and row major orders.

If all your lower bounds are 0, you can use an even simpler constructor:

```
public Array2( int s0, int s1, boolean isRow)
```

Another alternate way to construct sidl arrays is present in numbered arrays. The following constructor takes a Java 2 dimensional array, and copies it into a SIDL 2 dimensional array:

```
public Array2(foo.Bar[][] array, boolean isRow)
```

If you already have a numbered SIDL array of the correct dimension, you can copy a java array into it with the method fromArray. The method takes the same arguments as the constructor above, and returns nothing.

If you wish to go the other way, to copy a sidl array into a Java array, you may use the numbered array function toArray. toArray takes no arguments, and returns a new Java array with the SIDL array elements copied into it.

10.8 Interfaces and Abstract Classes

Babel implements SIDL interfaces as Java interfaces in Java. This is a close mapping in general, but it does have the problem that Java interfaces can't hold data. Since we need the correct IOR pointer in order to place that interface in an array or throw it as an Exception, the lack of data becomes a problem. For this reason, we have created Wrapper classes for interfaces and abstract classes.

All interfaces and abstract classes have static inner class named Wrapper. This Wrapper class holds the interface IOR pointer, and also inherits from gov.llnl.babel.BaseClass and implements the outer interface. Therefore, you can call all the interface methods on the wrapper object, as well as gov.llnl.babel.BaseClass methods such as _cast2, and isType.

This wrapper class is what is returned when an interface is gotten out of an array, a method takes or returns an interface, or when an exception implemented as an interface is caught. (There's actually a difference here. While what is gotten out of the Array or returned from a method is a Wrapper object, the programmer doesn't usually need to worry about that, as is shown in the example below. In the case of exceptions, you actually do have to catch the Wrapper. Exceptions are covered in more detail in Subsection 10.9) Because wrapper classes inherit only from an interface, they can be java casted to their enclosing interface, or it's super-interfaces, but must be Babel casted to any classes. In this example, Subclass implements Super-Interface:

```
SuperInterface.Array1 arry = new SuperInterface.Array1(5, true);
SubClass obj = new SubClass();
arry.set(0, (SuperInterface)obj);
obj = null;
SuperInterface temp = arry.get(0);
obj = (SubClass) temp;   //INCORRECT Will throw ClassCastException

obj = (SubClass) SubClass._cast((SuperInterface.Wrapper)temp); //CORRECT
```

Sometimes you can get away with not Java casting the interface to the Wrapper class before Babel casting it, but not in general. (Usually you don't have to when the interface was gotten out of an array)

Here's an example of casting an interface on the server side:

```
public objarg.SubClass toClass_Impl (/*in*/ objarg.Iface ifcy ) {
  // DO-NOT-DELETE splicer.begin(objarg.SubClass.toClass)
  objarg.SubClass ret =  (objarg.SubClass)
    ((objarg.Iface.Wrapper)ifcy)._cast2("objarg.SubClass");
  return ret;
  // DO-NOT-DELETE splicer.end(objarg.SubClass.toClass)
}
```

10.9 Exceptions

Exceptions are caught and thrown in exactly the same way as Java exceptions. If an exception is defined in SIDL, Babel will generate the code for it, and the exception can be thrown in Java. The only difference is that SIDL exception constructor cannot take a String. Instead, the message must be set with SIDL's `setNote` method, the message is gotten with SIDL's `getNote` method. This is important because SIDL exceptions inherit from the Java Class `Exception`. The Java compiler *will not* give an error if getMessage is called, but the message returned will not have been from SIDL.

The other problem is that regular Java exceptions cannot be passed on by Babel. Of course, it's not possible to throw normal non-SIDL exceptions from a SIDL Java function, the Java compiler will throw an error. (Unless you have changed the Java method "throws" statement outside the splicer blocks, which you should never do.) However, Java runtime exceptions, such as `ArrayIndexOutOfBoundsException` can be thrown. In this case, an error message and stack trace are printed to stderr, the method returns 0, the values of any out or inout arguments are set to NULL, and the program proceeds.

Finally, SIDL Exceptions may be interfaces, where as Java exceptions are always classes. This means Babel allows you to throw an interface. However, in Java we actually need to throw the interface's Wrapper class.

In this example we have a class `FibException` which implements two exception interfaces, `NegativeValueException` and `TooDeepException`. These two Exceptions are thrown by a babelized method named `getFib`. `getFib` is a standard recursive Fibonacci number generating function, in which if something goes wrong, it throws one of these two exceptions. First, server side:

```
public int getFib_Impl ( /*in*/ int n)
throws NegativeValueException.Wrapper, TooDeepException.Wrapper {
  if (n < 0) {
    FibException fex = new FibException();
    NegativeValueException.Wrapper neg = (NegativeValueException.Wrapper)
      NegativeValueException.Wrapper._cast(fex);
```

```
      neg.setNote("n negative");
      throw neg;
    }

    //  .... Do Fibonacci stuff ....
  }
```

You can see here some of the hoops you have to jump though to throw an interface. First, since we cannot create an interface, or it's Wrapper, directly, we first create a new `FibException` and cast it to the interface we want. Secondly, we have to refer to the Wrapper's full name in this case, because it is impossible to throw interfaces in Java. Finally, as with all SIDL Exceptions, we use `setNote` to set the exception's message, as we cannot pass in a message with the constructor.

Next the client side:

```
  try {
    fib.getFib(-1);
  } catch (NegativeValueException.Wrapper ex) {
    System.err.println(ex.getNote());
  } catch (TooDeepException.Wrapper ex) {
    System.err.println(ex.getNote());
  } catch (java.lang.Exception ex) {
    if (((sidl.BaseInterface)ex).isType("sidl.SIDLException")) {
      check(PASS, true, "Unexpected SIDL Exception thrown");
    } else {
      check(PASS, false, "Unexpected and unkown exception thrown");
    }
  }
```

In order to differentiate between the two different interfaces in this case we must catch the Wrappers explicitly by their fully qualified names. In the `exceptions` regression test we discover the types of the Exceptions by calling the SIDL function `isType` on them. However, because SIDL can cast between the two interfaces, in this case `isType` would return true no matter what the exception originally was. The final catch `java.lang.Exception ex` should not ever be executed in out example code. `getFib` does not throw any other kinds of exceptions, and babel cannot throw non-SIDL Exceptions. This was included because it demonstrates the most basic way to differentiate a SIDL exception from a Java exception.

10.10 Enumerations

Enumerations are implemented as `final static ints` in their own Java class, and as such, are accessed just like variables in that class. For example, if we had a sidl package named `dealership` that contained the following code segment:

```
    enum car {
      porsche = 911,
      ford = 150,
      mercedes = 550
    };
```

we would be able to get the value assigned to a Porsche by typing `dealership.car.porsche`.

10.11 Invoking Babel to generate Java bindings

To create Java stubs (i.e. code to support Java clients to a set of SIDL classes or interfaces), you should invoke Babel as follows [1]:

[1] For information on additional command line options, refer to Section 3.2.

```
% babel --client=Java file.sidl
```

or more cryptically

```
% babel -cJava file.sidl
```

This will create a great plethora of files, including a directory named `file`. This directory contains the Java client classes, if you want to take a look at them. The files ending in `_IOR.h` and `_IOR.c` are the Intermediate Object Representation. The files ending with `_jniStub.c` are the JNI stubs — the interface between a Java client and the IOR. The "jni" in the filename represents the fact that we use the Java Native Interface to communicate between Java and the IOR representation. The remaining header files have external Java API that Java clients may use.

To use the Java stubs, you must compile the stub files whose file names end with `_jniStub.c` and link them against the SIDL runtime library and a backend implementation. The resulting library needs to be referenced in a `.scl` file listed in the SIDL_DLL_PATH environment variable so that the Babel runtime library loader can find it. Also, the current directory needs to be in the CLASSPATH environment variable so that Java can find the `file` and `sidl` directories that contain the Java component of the client side.

10.12 Invoking Babel to generate Java implementations

To implement a set of SIDL classes in Java, you should invoke Babel as follows:

```
% babel --server=Java file.sidl
```

or use the short form

```
% babel -sJava file.sidl
```

The directory structure that results from this command is that same as the client side, there are just a bunch more files. In the `file` directory there are new files that end in `_Impl.java`. These are the java files where you should write your implementation. All of your methods in this class now also end in `_Impl`. In the current directory there are also new files that end in `_jniSkel.c`. These files are the equivalent to the `_jniStub.c` for the client side.

You should also notice that all the Client side files have been generated in addition to the new Server side files. These files are present to allow for calling methods on the current object in the Implementation java file. You can safely ignore them.

10.13 Environment Variables

There are some environment variables associated with running Java with Babel. You can find examples for some of these in the regression tests included with babel.

CLASSPATH: The CLASSPATH is an environment variable that Java uses to find `.class` files. It's is not specific to Babel, but it is necessary. It consists of a colon delimited series of directories to search for Java classes. In addition to any of your own Class files for use in Java server side, you should include `build dir/lib/sidl-ver.jar` where ver is the current sidl version, and `build dir/runtime/java`.

BABEL_JVM_FLAGS: This environment variable is used *only* when passing java command line variables to Java server side. It consists of a semi-colon delimited list of command line variables you wish to pass to Java server side. (A useful one might be -Xcheck:jni))Here's an example:

```
BABEL_JVM_FLAGS="-verbose:gc;-Xmx500m"
```

It is also necessary to set your LD_LIBRARY_PATH (or LIBPATH on AIX) and SIDL_DLL_PATH correctly. Not including all the necessary files in the SIDL_DLL_PATH and LD_LIBRARY_PATH *can* crash the JVM in unhelpful ways. Babel tries to generate helpful error messages, but sometimes the JVM crashes due to missing files and doesn't generate very helpful output. If the JVM crashes, make sure you've included all the necessary files in your SIDL_DLL_PATH and LD_LIBRARY_PATH.

Chapter 11

Python Bindings

Contents

Babel requires a Python shared library. Because Python 2.3 has a configure/build system that builds shared libraries on many architectures, we recommend that you use Python 2.3 or beyond.

11.1 How to Create a SIDL Object in Python

(once you've built the Python extension module)

You need to import the extension module and then calling a method to create an instance. If you have a class whose fully qualified name is x.y.z, you would say:

```
>>> import x.y.z
>>> obj = x.y.z.z()
```

The last part of the class name is repeated. You can also use **`from x.y.z import *`** if you prefer; although, you must guarantee that there are no namespace collisions.

In some cases, the Python extension module may be name `zmodule.so` instead of simply `z.so`. This might tempt you to say **`import x.y.zmodule`** instead of just **`import x.y.z`**; resist this temptation!

11.2 How to Cast SIDL Objects in Python

Let's say you have an object `obj`, and you would like to see if it is an instance of a SIDL class or interface whose fully qualified name is `x.y.z`. Here is how you do it.

```
>>> import x.y.z
>>> zobj = x.y.z.z(obj)
```

Of course, you don't need the import if you know that `x.y.z` has already been imported. If `zobj` is not equal to `None`, the cast was successful.

11.3 How to Call Methods from Python

Once you have created an object, you call methods on it using normal Python method calls. The arguments to the method only include the `in` and `inout` arguments, and the return value of the Python method includes the SIDL return value and the `inout` and `out` parameters. Hopefully, this will seem natural to Python programmers. For the following example, the object `obj` has a method `passeverywhere` with the following SIDL declaration:

```
double passeverywhere( in double d1, out double d2, inout double d3 );
```

You can see the Python calling signature with **`print obj.passeverywhere.__doc__`**. Here is what that shows for this example:

```
$ python
>>> import Args.Cdouble
>>> obj = Args.Cdouble.Cdouble()
>>> print obj.passeverywhere.__doc__
passeverywhere(in double d1,
               inout double d3)
RETURNS
   (double _return,
    out double d2,
    inout double d3)
```

In the method documentation, the SIDL method's return value is called `_return`; and unless the method is `void`, the return value always appears first. The fact that `_return` starts with an underbar should alert you to the fact that it is not a parameter because parameter names cannot start with an underbar. The document comments from the SIDL file (i.e. comments enclosed in `/** */` comments) appear below the Babel generated signature documentation.

Static methods of a class are available in the Python `x.y.z` namespace assuming you use the **`import x.y.z`** command. Static methods have documentation just like class methods.

Examples of calls to SIDL overloaded methods are based on the `overload_sample.sidl` file shown in Section 5.6. Recall that the file describes three versions of the `getValue` method. The first takes no arguments, the second takes an integer argument, and the third takes a boolean. Each is called in the code snippet below:

```
b1 = 1
i1 = 1

t = Overload.Sample.Sample()

nresult = t.getValue()
iresult = t.getValueInt(i1)
bresult = t.getValueBool(b1)
```

11.4 Catching and Throwing Exceptions in Python

Python exceptions must be Python classes; they cannot be a C extension type — the mechanism used to wrap SIDL objects as Python objects. Because of this, Babel defines an exception class for each SIDL type that implements `sidl.BaseException`. For a type called `x.y.z`, the Python exception class is named `x.y.z._Exception`. In Babel 0.10.2 and previous releases, the Python exception class was named `x.y.z.Exception`, but this name can potentially collide with the class constructor or a static method named `Exception`. For backwards compatibility, Babel defines `x.y.z.Exception` if the name `Exception` is not used in the class.

SIDL exceptions are caught and thrown very much like normal Python exceptions are caught and thrown except you need to use the Python exception class for the SIDL type. The exception value holds the SIDL object as attribute `exception`. Here is an example of a code catching exceptions from a call to `getFib`. Note that `eobj.exception` is an instance of `ExceptionTest.NegativeValueException.NegativeValueException`, the Python type corresponding to the SIDL type `ExceptionTest.NegativeValueException`.

```
try:
  fib.getFib(-1, 10, 10, 0)
except ExceptionTest.NegativeValueException._Exception:
  (etype, eobj, etb) = sys.exc_info()
  # eobj is the SIDL exception object
  print eobj.exception.getNote()  # show the exception comment
  print eobj.exception.getTrace() # and traceback
```

Here is an example of a Python implementation function that throws an exception. The `setNote` method provides a useful error message, and the `add` method helps provide a multi-language traceback capability (provided each layer of the call stack calls `add`).

```
def getFib(self, n, max_depth, max_value, depth):
  # sidl EXPECTED INCOMING TYPES
  # ============================
  # int n, max_depth, max_value, depth
  #
  # sidl EXPECTED RETURN VALUE(s)
  # =============================
  # int _return
  # DO-NOT-DELETE splicer.begin(getFib)
  if (n < 0):
    ex = ExceptionTest.NegativeValueException.NegativeValueException()
    ex.setNote("n negative")
    ex.add(__name__, 0, "ExceptionTest.Fib.getFib")
    raise ExceptionTest.NegativeValueException._Exception, ex
  # numerous lines deleted
  # DO-NOT-DELETE splicer.end(getFib)
```

11.5 Building Python Extension Modules

SIDL creates a `setup.py` file that can be used to build the Python extension modules that you create. `setup.py` uses the Python distutils package to build the Python extension modules. There are two extra command line arguments.

- `--include-dirs=` — Use this to specify extra directories for the preprocessor include path. This is like `-I` for most C compilers.
- `--library-dirs=` — Use this to specific extra directories for static or shared libraries. This is like `-L` for most C compilers/loaders.

Normally, you need to specify the directory where the SIDL runtime headers and SIDL Python headers are stored with `--include-dirs=`. You also need to specify the directory where `libsidl.so` is stored. Here is a hypothetical example:

```
setup.py --include-dirs=/usr/local/include
--include-dirs=/usr/local/include/python
--library-dirs=/usr/local/lib build_ext --inplace
```

It is unlikely that any installation actually uses those settings.

11.6 Setting up to Run Python

Here I assume that you've installed Babel in directories rooted at `$PREFIX`. You need to have `$PREFIX/python` in your `PYTHONPATH` environment variable in addition to the directory where you are doing your work.

On many systems, you will need `$PREFIX/lib` in your `LD_LIBRARY_PATH` (or whatever system setting controls which directories are searched for shared libraries/dynamic link libraries).

You will likely need to use `SIDL_DLL_PATH` (a semicolon separated path) to provide the path to the directory that holds the shared library/dynamic link library containing the implementation of the SIDL objects.

11.7 Notes

The Python binding for SIDL long uses Python's unlimited precision integer data type, so it will not behave exactly like a 64 bit integer (i.e. there is no overflow). For Python versions before 2.2, your code needs to guarantee that a Python unlimited precision integer is used whenever a SIDL long is needed. For example, if you want to call a method whose SIDL signature is `bool isPrime(long num)`, calling `isPrime(1)` will fails; but calling `isPrime(1L)` will work fine.

The Python binding for an array of SIDL longs may use an array of 64 bit integers if Numeric Python supports a 64 bit integer. Otherwise, it uses an array of Python's indefinite precision integers (i.e., integers with unlimited bits).

What does this error message mean?

```
>>> import x.y.Zmodule
Traceback (innermost last):
File "<stdin>", line 1, in ?
ImportError:  dynamic module does not define init function (initZmodule)
```

Is the name of your SIDL interface/class `x.y.Z` or `x.y.Zmodule`, if it's the former, you should say **`import x.y.Z`**. If this isn't the problem, submit a bug report for Babel. It might be informative to look at the symbol of the shared library/dynamic link library using a tool like nm. I suppose it's also worth checking the PYTHONPATH environment variable to make sure it's pointing to the right place.

```
>>> import x.y.Z
Fatal Python error:  Cannot load implementation for SIDL class x.y.Z
Abort (core dumped)
```

This means that the Python stub code (the code that links Python to SIDL's independent object representation (IOR)) failed in its attempt to load the shared library or dynamic link library containing the implementation of SIDL class `x.y.Z`. Make sure the environment variable `SIDL_DLL_PATH` lists all the directories where the shared libraries/dynamic link libraries for your SIDL objects/interfaces are stored. `SIDL_DLL_PATH` is a semicolon separated list of directories where SIDL client stubs will search for shared libraries required for SIDL classes and interfaces. Make sure the directory in which the SIDL runtime resides is in the `LD_LIBRARY_PATH` (or whatever your machine's mechanism for locating shared library files is).

```
>>> import x.y.Z
Fatal Python error:  Cannot load implementation for SIDL interface x.y.Z
Abort (core dumped)
```

This is the same problem for an interface as described immediately above for a class.

11.8 How to Implement SIDL Objects in Python

To build server side Python, you must have Python compiled as a shared library or dynamically link library. The standard Python build only builds the necessary shared library on a few platforms — none of which are target platforms for Babel. Some Linux distributions include a Python shared library, and it is possible to make a Python shared library on Solaris. The Python shared library should contain the objects from libpythonx.y.a where x.y is your Python version. Making a shared library is different on each platform, so it is not covered here.

To implement an object in Python, first you must run Babel to create the Python server side bindings [1].

```
% babel --server=python file.sidl
```

[1] For information on additional command line options, refer to Section 3.2.

or simply

```
% babel -s=python file.sidl
```

This creates the IOR, Python skeleton (pSkel), and Python launch (pLaunch) files in your currect directory, and it will create tree of subdirectories based on the package hierarchy found in file.sidl. The IOR, pSkel and pLaunch files must be compiled and place in a shared library (in most cases).

The tree of subdirectories created by Babel includes Python implementation files whose name ends with `_Impl.py` and Python extension modules for the Python client side binding (`_Module.h` and `_Module.c`). The extension modules need to be compiled as above in section 11.5, and you need to fill in the implementations in the `_Impl.py` files.

Babel generates the outline of the implementation. It creates a class definition and empty methods for you to fill in the each `_Impl.py` file. If you put your code between the comments as indicated, your code will be preserved if you rerun Babel. Any changes out side the comment blocks will be lost if you rerun Babel. Here is an example of a method implementation:

```
def passeverywhere(self, d1, d3):
  #
  # SIDL EXPECTED INCOMING TYPES
  # ============================
  # double d1
  # double d3
  #

  #
  # SIDL EXPECTED RETURN VALUE(s)
  # =============================
  # (_return, d2, d3)
  # double _return
  # double d2
  # double d3
  #

  # DO-NOT-DELETE splicer.begin(passeverywhere)
  if (d1 == 3.14):
    retval = 3.14
  else:
    retval = 0
  return (retval, 3.14, -d3)
  # DO-NOT-DELETE splicer.end(passeverywhere)
```

Babel generated everything except the code that appears between the `splicer.begin` and `splicer.end` comments.

Chapter 12

SIDL Backend

Contents

12.1 Introduction

This chapter introduces the SIDL backend associated with symbols that may originate from a SIDL file or the corresponding Extensible Markup Language (XML) representation. Unlike most of the other supported language bindings, the output from this backend is textual in nature. That is, it is the textual, human-readable form of the interfaces description. An alternative text form, XML that is, which is also supported is described in Chapter 13.

12.2 Purpose

The primary reason for having a SIDL backend is to provide a mechanism for generating human-readable text for interfaces that are written in conformant XML. It is important to emphasize that Babel requires the XML to conform to the SIDL DTD in order to benefit from this feature.

Generating SIDL provides a feature to collaborators who are interested in experimenting with the XML form of the interfaces. Such groups should find the more human-readable descriptions of the interfaces to be helpful for distribution and discussion.

12.3 Generated versus Original SIDL files

Generated SIDL files may differ from their original SIDL files in several respects in terms of content as well as layout. These differences are summarized below.

Packages. The code generation is limited to one high-level package per generated file. In fact, the name of the generated file is currently defined to be the concatenation of the name of the highest-level package and .sidl.

Versioning. The generation of requires statements is inferred from the symbols that actually appear in the associated interface descriptions. The intent is to provide a requires statement for only the highest level package needed of a given version. Consequently, requires and imports statements that were not necessary for resolving symbols will not appear. Also, fully qualified names will be shortened in the generated files due to the automatic generation of the associated requires statement(s). Finally, since an import and require statement can be used in a SIDL file and no distinction is made in the XML, only a require statement will appear in the generated file.

Implements. Since there is no distinction between *implements-all* and *implements* in the XML version of the interfaces, the generated code outputs *implements* along with the inherited methods.

Comments. Babel preserves only document, or doc, comments so any comments that do not conform will not appear in the generated file [1].

Whitespace. Obviously there may be whitespace differences in the generated file. These include indentation, blank spaces and lines, and brace placement.

As an example, suppose we have a package in the file `foo.sidl`. The original file's contents are:

```
package foo version 1.0 {

  class A {}

  package bar version 2.0 {
    class B {}
  }

}
```

The resulting contents of the generated SIDL file are:

```
package foo version 1.0 {

  class A {
  }

  package bar version 2.0 {

    class B {
    }

  }


}
```

Notice the differences in white space. To illustrate more features, further suppose we have a package in the file `fooTest.sidl`. The original file's contents are:

```
// An ignored comment
require foo version 1.0;
require foo.bar version 2.0;

/**
 * Test of doc comment with XML special characters < & >.
 */
package fooTest version 0.1 {

  /**
   * Another doc comment for an empty class.
   */
  class A extends foo.bar.B {}

  class B extends foo.A {}
}
```

[1] For more information on comments and doc-comments, refer to **Comments and Doc-Comments** in Section 5.2.

The resulting contents of the generated SIDL file are:

```
require foo version 1.0;
require foo.bar version 2.0;

/**
 * Test of doc comment with XML special characters < & >.
 */
package fooTest version 0.1 {

  /**
   * Another doc comment for an empty class.
   */
  class A extends foo.bar.B {
  }

  class B extends foo.A {
  }

}
```

Here we see the exclusion of non-document comments and the retention of document comments. Refer to Section 5.2 and Appendix C for more information about document comments.

12.4 XML File Comparison

Testing has revealed that XML generated from the original SIDL file compared to XML generated from generated SIDL files have only minor differences. In fact, the differences are limited to specific metadata fields. Specifically, the date, source-url, and source-line entries can differ. The dates, however, will be the same if the ¨–suppress-timestampöption was used when both XML files were generated. Similarly, the source-line entries will be the same if the package started on the same line in both the original and generated SIDL files. The latter can happen if, for instance, there are no non-doc comments in the original file.

12.5 Babel Command Line Options

To generate SIDL from a file using the default repository to resolve symbols, you should invoke Babel as follows [2]:

```
% babel --text=SIDL file.sidl
```

or use the short form

```
% babel -tSIDL file.sidl
```

Alternatively, you can generate SIDL from XML symbols, again assuming the default repository is used to resolve symbols, by typing the following at the command line:

```
% babel --text=SIDL packagename
```

or use the short form

```
% babel -tSIDL packagename
```

[2] For information on additional command line options, refer to Section 3.2.

Chapter 13

XML Backend

Contents

13.1 Introduction

This chapter introduces the XML representation supported by Babel. Here we describe the motivation for having an XML backend and the basic structure of a conformant XML file. To illustrate, a few of the SIDL symbol XML files will be presented.

Details regarding the layout of XML files can be obtained by referring to the Document Type Definition (DTD) provided in Appendix C. For more on the type repositories, refer to[XML Repositories in Section 5.2.

13.2 Purpose

The XML backend is a key feature of Babel. It provides the basis upon which the symbol, or type, repository depends. SIDL files should be translated into their XML representations and stored in the type repository. This is the case for the SIDL interfaces and classes that are provided as part of the Babel toolkit.

13.3 Basic Structure

Each generated XML file specifies the interfaces for a given SIDL Symbol in an expanded textual representation. Although the structure of a given file depends upon the type of symbol it contains, the basic layout consists of a set of common elements followed by symbol-specific elements.

Common Elements

The common elements are *prolog*, *document type*, *name*, *metadata*, and *comment*. These elements, which are described below, are followed by symbol-specific information.

Prolog. The prolog simply identifies the XML version and encoding scheme associated with the file.

Document Type. The document type declaration states the document contains a *Symbol* and it identifies the associated DTD (i.e., *SIDL.dtd*).

Name. The symbol name is the first element within the symbol tag pair and it identifies the name and version of the SIDL symbol that is described in the file.

Metadata. The metadata element identifies the date the XML file was generated[1] along with a set of three key-value pair entries. The first, *source-url*, identifies the URL of the SIDL file that was used to generate the XML file. The second, *source-line*, identifies the line within the SIDL file at which the symbol was first detected. Finally, *babel-version* identifies the version of Babel that was used to generate the XML file.

Comment. The comment tag is used to save off any comment that is associated with the symbol.

Packages

In addition to the common elements, packages retain elements and attributes associated with SIDL packages. These include whether or not the package is *final* along with a list of the symbols contained within the package. The list of symbols consists of the tuple: name, type, and version.

For example, the XML representation of the toplevel SIDL package (i.e., `sidl`) is:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE Symbol PUBLIC "-//CCA//SIDL Symbol DTD v1.1//EN" "SIDL.dtd">
<Symbol>
   <SymbolName name="sidl" version="0.8.2"/>
   <Metadata date="20030320 13:29:02 PST">
      <MetadataEntry key="source-url" value="file:/home/dahlgren/RELEASE/linux_kcc/share/../../b
      <MetadataEntry key="source-line" value="40"/>
      <MetadataEntry key="babel-version" value="0.8.2"/>
   </Metadata>
   <Comment>
The &lt;code&gt;sidl&lt;/code&gt; package contains the fundamental type and interface
definitions for the &lt;code&gt;SIDL&lt;/code&gt; interface definition language.  It
defines common run-time libraries and common base classes and interfaces.
Every interface implicitly inherits from &lt;code&gt;sidl.BaseInterface&lt;/code&gt;
and every class implicitly inherits from &lt;code&gt;sidl.BaseClass&lt;/code&gt;.

   </Comment>
   <Package final="false">
      <PackageSymbol name="BaseInterface" type="interface" version="0.8.2"/>
      <PackageSymbol name="BaseClass" type="class" version="0.8.2"/>
      <PackageSymbol name="BaseException" type="class" version="0.8.2"/>
      <PackageSymbol name="DLL" type="class" version="0.8.2"/>
      <PackageSymbol name="Loader" type="class" version="0.8.2"/>
      <PackageSymbol name="ClassInfo" type="interface" version="0.8.2"/>
      <PackageSymbol name="ClassInfoI" type="class" version="0.8.2"/>
   </Package>
</Symbol>
```

Interfaces

Similarly, the XML for interface symbols contain the common elements. In addition, they retain elements and attributes associated with SIDL interfaces. These include any extensions, parent interfaces it implements, and its methods. Method information includes its name, communication mode, short name, name extension (for languages that don't support method overloading), comment, return type, argument list, and exception list.

For example, the XML representation of *`sidl.BaseInterface`* is:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE Symbol PUBLIC "-//CCA//SIDL Symbol DTD v1.1//EN" "SIDL.dtd">
<Symbol>
```

[1] Assuming the ¨–suppress-timestampöption was not used.

```
      <SymbolName name="sidl.BaseInterface" version="0.8.2"/>
      <Metadata date="20030320 13:29:02 PST">
         <MetadataEntry key="source-url" value="file:/home/dahlgren/RELEASE/linux_kcc/share/../../b
         <MetadataEntry key="source-line" value="47"/>
         <MetadataEntry key="babel-version" value="0.8.2"/>
      </Metadata>
      <Comment>
   Every interface in &lt;code&gt;SIDL&lt;/code&gt; implicitly inherits
   from &lt;code&gt;BaseInterface&lt;/code&gt;, and it is implemented
   by &lt;code&gt;BaseClass&lt;/code&gt; below.
      </Comment>
      <Interface>
         <ExtendsBlock/>
         <AllParentInterfaces/>
         <MethodsBlock>
            <Method communication="normal" copy="false" definition="abstract" extension="" shortnam
               <Comment>
   &lt;p&gt;
   Add one to the intrinsic reference count in the underlying object.
   Object in &lt;code&gt;SIDL&lt;/code&gt; have an intrinsic reference count.
   Objects continue to exist as long as the reference count is
   positive. Clients should call this method whenever they
   create another ongoing reference to an object or interface.
   &lt;/p&gt;
   &lt;p&gt;
   This does not have a return value because there is no language
   independent type that can refer to an interface or a
   class.
   &lt;/p&gt;
               </Comment>
               <Type type="void"/>
               <ArgumentList/>
               <ThrowsList/>
            </Method>
            <Method communication="normal" copy="false" definition="abstract" extension="" shortnam
               <Comment>
   Decrease by one the intrinsic reference count in the underlying
   object, and delete the object if the reference is non-positive.
   Objects in &lt;code&gt;SIDL&lt;/code&gt; have an intrinsic reference count.
   Clients should call this method whenever they remove a
   reference to an object or interface.
               </Comment>
               <Type type="void"/>
               <ArgumentList/>
               <ThrowsList/>
            </Method>
            <Method communication="normal" copy="false" definition="abstract" extension="" shortnam
               <Comment>
   Return true if and only if &lt;code&gt;obj&lt;/code&gt; refers to the same
   object as this object.
               </Comment>
               <Type type="boolean"/>
               <ArgumentList>
                  <Argument copy="false" mode="in" name="iobj">
                     <Type type="symbol">
                        <SymbolName name="sidl.BaseInterface" version="0.8.2"/>
                     </Type>
                  </Argument>
               </ArgumentList>
```

```
            <ThrowsList/>
         </Method>
         <Method communication="normal" copy="false" definition="abstract" extension="" shortnam
            <Comment>
Check whether the object can support the specified interface or
class.  If the &lt;code&gt;SIDL&lt;/code&gt; type name in &lt;code&gt;name&lt;/code&gt;
is supported, then a reference to that object is returned with the
reference count incremented.  The callee will be responsible for
calling &lt;code&gt;deleteRef&lt;/code&gt; on the returned object.  If
the specified type is not supported, then a null reference is
returned.
            </Comment>
            <Type type="symbol">
               <SymbolName name="sidl.BaseInterface" version="0.8.2"/>
            </Type>
            <ArgumentList>
               <Argument copy="false" mode="in" name="name">
                  <Type type="string"/>
               </Argument>
            </ArgumentList>
            <ThrowsList/>
         </Method>
         <Method communication="normal" copy="false" definition="abstract" extension="" shortnam
            <Comment>
Return whether this object is an instance of the specified type.
The string name must be the &lt;code&gt;SIDL&lt;/code&gt; type name.  This
routine will return &lt;code&gt;true&lt;/code&gt; if and only if a cast to
the string type name would succeed.
            </Comment>
            <Type type="boolean"/>
            <ArgumentList>
               <Argument copy="false" mode="in" name="name">
                  <Type type="string"/>
               </Argument>
            </ArgumentList>
            <ThrowsList/>
         </Method>
         <Method communication="normal" copy="false" definition="abstract" extension="" shortnam
            <Comment>
Return the meta-data about the class implementing this interface.
            </Comment>
            <Type type="symbol">
               <SymbolName name="sidl.ClassInfo" version="0.8.2"/>
            </Type>
            <ArgumentList/>
            <ThrowsList/>
         </Method>
      </MethodsBlock>
   </Interface>
</Symbol>
```

Classes

Class definitions are almost identical to that of interfaces except for additional attributes. The additional attribute, which include whether or not the class is *final*. Recall that Babel/SIDL supports only single inheritance of classes; therefore, only a single class will appear in the extends block. If one does not appear in the original SIDL file, by default the class will extend *`sidl.Baseclase`*.

For example, the XML representation of *`sidl.BaseClass`* is:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE Symbol PUBLIC "-//CCA//SIDL Symbol DTD v1.1//EN" "SIDL.dtd">
<Symbol>
   <SymbolName name="sidl.BaseClass" version="0.8.2"/>
   <Metadata date="20030320 13:29:02 PST">
      <MetadataEntry key="source-url" value="file:/home/dahlgren/RELEASE/linux_kcc/share/../../b
      <MetadataEntry key="source-line" value="109"/>
      <MetadataEntry key="babel-version" value="0.8.2"/>
   </Metadata>
   <Comment>
Every class implicitly inherits from &lt;code&gt;BaseClass&lt;/code&gt;.  This
class implements the methods in &lt;code&gt;BaseInterface&lt;/code&gt;.
   </Comment>
   <Class abstract="false">
      <Extends/>
      <ImplementsBlock>
         <SymbolName name="sidl.BaseInterface" version="0.8.2"/>
      </ImplementsBlock>
      <AllParentClasses/>
      <AllParentInterfaces>
         <SymbolName name="sidl.BaseInterface" version="0.8.2"/>
      </AllParentInterfaces>
      <MethodsBlock>
         <Method communication="normal" copy="false" definition="final" extension="" shortname="
            <Comment>
&lt;p&gt;
Add one to the intrinsic reference count in the underlying object.
Object in &lt;code&gt;SIDL&lt;/code&gt; have an intrinsic reference count.
Objects continue to exist as long as the reference count is
positive. Clients should call this method whenever they
create another ongoing reference to an object or interface.
&lt;/p&gt;
&lt;p&gt;
This does not have a return value because there is no language
independent type that can refer to an interface or a
class.
&lt;/p&gt;
            </Comment>
            <Type type="void"/>
            <ArgumentList/>
            <ThrowsList/>
         </Method>
         <Method communication="normal" copy="false" definition="final" extension="" shortname="
            <Comment>
Decrease by one the intrinsic reference count in the underlying
object, and delete the object if the reference is non-positive.
Objects in &lt;code&gt;SIDL&lt;/code&gt; have an intrinsic reference count.
Clients should call this method whenever they remove a
reference to an object or interface.
            </Comment>
            <Type type="void"/>
            <ArgumentList/>
            <ThrowsList/>
         </Method>
         <Method communication="normal" copy="false" definition="final" extension="" shortname="
            <Comment>
Return true if and only if &lt;code&gt;obj&lt;/code&gt; refers to the same
object as this object.
            </Comment>
```

```
            <Type type="boolean"/>
            <ArgumentList>
               <Argument copy="false" mode="in" name="iobj">
                  <Type type="symbol">
                     <SymbolName name="sidl.BaseInterface" version="0.8.2"/>
                  </Type>
               </Argument>
            </ArgumentList>
            <ThrowsList/>
         </Method>
         <Method communication="normal" copy="false" definition="normal" extension="" shortname=
            <Comment>
Check whether the object can support the specified interface or
class.  If the &lt;code&gt;SIDL&lt;/code&gt; type name in &lt;code&gt;name&lt;/code&gt;
is supported, then a reference to that object is returned with the
reference count incremented.  The callee will be responsible for
calling &lt;code&gt;deleteRef&lt;/code&gt; on the returned object.  If
the specified type is not supported, then a null reference is
returned.
            </Comment>
            <Type type="symbol">
               <SymbolName name="sidl.BaseInterface" version="0.8.2"/>
            </Type>
            <ArgumentList>
               <Argument copy="false" mode="in" name="name">
                  <Type type="string"/>
               </Argument>
            </ArgumentList>
            <ThrowsList/>
         </Method>
         <Method communication="normal" copy="false" definition="normal" extension="" shortname=
            <Comment>
Return whether this object is an instance of the specified type.
The string name must be the &lt;code&gt;SIDL&lt;/code&gt; type name.  This
routine will return &lt;code&gt;true&lt;/code&gt; if and only if a cast to
the string type name would succeed.
            </Comment>
            <Type type="boolean"/>
            <ArgumentList>
               <Argument copy="false" mode="in" name="name">
                  <Type type="string"/>
               </Argument>
            </ArgumentList>
            <ThrowsList/>
         </Method>
         <Method communication="normal" copy="false" definition="final" extension="" shortname="
            <Comment>
Return the meta-data about the class implementing this interface.
            </Comment>
            <Type type="symbol">
               <SymbolName name="sidl.ClassInfo" version="0.8.2"/>
            </Type>
            <ArgumentList/>
            <ThrowsList/>
         </Method>
      </MethodsBlock>
   </Class>
</Symbol>
```

13.4 Command Line Options

XML must be generated from a SIDL file. The Babel command line is as follows [2]:

```
% babel --text=XML file.sidl
```

or simply

```
% babel -tXML file.sidl
```

In both cases, the use of the default repository is assumed for resolving symbols. In addition, the output will appear in the default output directory.

[2] For information on additional command line options, refer to Section 3.2.

Chapter 14

HTML Interface Documentation

Contents

14.1 Introduction

Babel can automatically create interface documentation using the HTML backend. This capability is modeled after the javadoc documentation available with Java. It is invoked with the `--text=html` command line option.

Part III

Advanced Topics

Chapter 15

Building Portable Polyglot Software

Babel generates very portable source code for multilingual programing. There is also an art and science to transforming the source code to binary assets without breaking the language encapsulation Babel is trying to create. This chapter discusses the details: from the mundane issues of file layout, to the arcana of linker and loader flags.

Contents

15.1 Layout of Generated Files

Babel generates a lot of files. Many of these files you never have to look at in an editor, but they must all be compiled and properly linked into an application (see Section 15.2). In this section we discuss a host of flags that can affect where files get generated.

- **`--output-directory`**=*path*
  This sets the root directory of where your files will be generated. The path can be absolute, or relative to the current working directory.

- **`--generate-subdirs`**
  This option forces files to be laid out in a directory hierarchy following the package hierarchy in the SIDL file. This arrangement is required for the Java and Python languages, so those generators force this option on and allow no means to turn it off. For C/C++ and Fortran 77/90, the default is that all files be generated in the single output directory with no package-named subdirectories.

- **`--language-subdir`**
  This option was contributed by a user. This option appends a language-specific subdirectory (e.g. c, python, f77) to the end of the path.

- **`--hide-glue`**
  This option was contributed by a user. The intent here is to separate the Impl files (which must be modified) from all other files. If this flag is set, then wherever an Impl file gets generated, all the corresponding Skels, Stubs, IORs, etc get generated in a subdirectory named `glue`.

Arbitrary combinations of the above flags are allowed. Regardless of the order they appear in the commandline, they are applied to the resulting path in the order they are presented above. For example if a SIDL file `pkg.sidl` defines a *`Cls`* class in the *`pkg`* package, and the user runs Babel as follows:

```
% babel -lugo there -sc
```

Then the majority of the sources will be generated in the `there/pkg/c/glue/` directory (except the Impl files which will occur one directory up in `there/pkg/c/`). Note the use of equivalent short-form commands in this example. If readers wish to review long and short forms of command line arguments, see Tabel 3.1 on page 13.

Note that many of these options were contributed by users and are not employed in Babel's own build. Instead, we tend to put a SIDL file in a directory and then generate client-side or server-side bindings in in either `runXXX/` or `libXXX/` subdirectories, respectively (where `XXX` is a language name). We don't use the **`--generate-subdirs`** or **`--hide-glue`** flags because they place source files that belong in the same library in different directories. Automake, which Babel uses as part of its build system, works much more reliably when all the sources that go into a library appear in the same directory as the library to be. The **`--language-subdir`** has a similar effect to what we do manually, but doesn't capture if it was client-side or server-side. In our tests and demos, we tend to build these separately because we want to exercise different drivers with different implementations.

15.2 Grouping compiled assets into Libraries

Babel enables one to completely encapsulate language dependencies inside a static or dynamically loaded library. This means that one can take a SIDL file and a compiled library, generate the bindings they want in their language of choice from the SIDL file, link against the library, and use it. . . never knowing what the original implementation language is for the library.

Babel generates the source code to accomplish this level of language interoperability, but users must use their compilers and linkers correctly for the effect to be complete. This section deals with many of the details that

15.2.1 Basics of Compilation and Linkage

What we generally think of as a compiler is really an ensemble of related tools. Generally there is a preprocessing step where very simple transformations occur (e.g. #define, #include directives and others). Next, the compiler proper executes and typically transforms your sourcecode into assembler or some other intermediate form. Optimizers work on this intermediate form and do perform additional transformations. Most big vendors of C, C++, and Fortran compilers have a common optimizer for all languages. Next, assemblers transform the optimized codes into platform-specific binaries. But this is not the end. The binaries may be linked together into libraries or programs. Libraries can be linked against other libraries, and eventually multiple programs. The main difference is that a program has additional instructions to bootstrap itself, do some interaction with the operating system, receive an argument list, and call `main()`. To see all this in action, try building a "hello world" type program in your favorite language, and run the "compiler" with an additional flag such as **`-v`**, **`--verbose`**, or whatever.

For example, this is what I get from a g77 compiler.

```
% g77 hello_world.f
% ./a.out
Hello World!  % g77 -v hello_world.f
Driving:  g77 -v hello_world.f -lfrtbegin -lg2c -lm -shared-libgcc
Reading specs from /usr/local/gcc/3.2/lib/gcc-lib/i686-pc-linux-gnu/3.2/specs
Configured with:  ../gcc-3.2/configure --prefix=/usr/local/gcc/3.2
Thread model:  posix
gcc version 3.2
```

```
/usr/local/gcc/3.2/lib/gcc-lib/i686-pc-linux-gnu/3.2/f771 hello_world.f
-quiet -dumpbase hello_world.f -version -o /tmp/ccp2GBGE.s
GNU F77 version 3.2 (i686-pc-linux-gnu)
compiled by GNU C version 3.2.
as --traditional-format -V -Qy -o /tmp/ccEiIsHc.o /tmp/ccp2GBGE.s
GNU assembler version 2.11.90.0.8 (i386-redhat-linux) using BFD version
2.11.90.0.8
/usr/local/gcc/3.2/lib/gcc-lib/i686-pc-linux-gnu/3.2/collect2 -m elf_i386
-dynamic-linker /lib/ld-linux.so.2 /usr/lib/crt1.o /usr/lib/crti.o /usr/local/gcc/3.2/
-L/usr/local/gcc/3.2/lib/gcc-lib/i686-pc-linux-gnu/3.2 -L/usr/local/gcc/3.2/lib/gcc-li
/tmp/ccEiIsHc.o -lfrtbegin -lg2c -lm -lgcc_s -lgcc -lc -lgcc_s -lgcc /usr/local/gcc/3.2
/usr/lib/crtn.o
```

For the purposes of this discussion, we will make a big distinction between linking to build a library and linking to build and executable. Even though these transformations have similar names, they perform very different kinds of transformations to the code.

15.2.2 Circular Dependencies and Single-Pass Linkers

Almost all linkers are single pass. This means that when linking an executable, linkers will run through the list of libraries exactly once trying to resolve symbols Ever get libraries listed in the wrong order and an executable wouldn't get built? Ever have to list the same libraries over and over again to build an executable? These are both side-effects of single pass linkers. The symbols in question are essentially jumps in the instruction code corresponding to subroutines that are defined elsewhere. When linking a final executable, all these symbols need to be resolved. When linking libraries, multiple undefined symbols are commonplace.

Having to list libraries over and over again in the link line when compiling the final executable typically indicates a circular dependency between libraries. Circular dependencies are much better kept within a single library. Even though linkers are single-pass between libraries, they exhaustively search within them.

This is important because all the files generated by Babel have a circular dependency in each Babel type. The stub makes calls on the IOR, the IOR calls the Skel, the Skel calls the Impl, but the Impl also may make calls on a Stub. Just like C++ has a `this` object, and Python has a `self`, Babel objects have a stub for them to call methods on themselves and dispatch properly through Babel's IOR layer.

15.2.3 IOR as single point of access

When building a Babelized library, its also important to note if your code has dependencies to other Babel types not in your library. These types often appear as base classes, argument types, or even exception types. Your library will need stubs corresponding to all these types, so it is best to put these in your library as well. We call these external stubs.

Many have tried to minimize replication of Babel stubs by removing the external stubs and letting the library link directly against the stubs in an external library. This is a mistake because the external library may be implemented in a different language, and the stubs may be for a different language binding. By bundling the external stubs specific to your implementation with the implementation's library, you are ensuring that the only access your library has with any other Babelized library is through the IOR. This is a good thing. The Babel IOR is the same for all language bindings and essentially forms the binary interface by which all Babel objects interact.

15.3 Dynamic vs. Static Linking

Most UNIX users are very comfortable with statically linked libraries (e.g. `libXXX.a`). Most are aware of "shared object files" in UNIX (with the form `libXXX.so`) though few actually build them. Even fewer still are familiar with dynamically linked libraries, called DLL's in Microsoft (after the common `.dll` suffix), which involve actually selecting and loading dynamic libraries at run time based on their string name. MacOSX uses the novel suffix `libXXX.dynlib`. (In most UNIX systems, including Linux and Solaris, .so "shared object files" are actually dynamically linked libraries.) This section serves as a quick overview of how Babel handles both static and dynamic libraries, including runtime loading.

15.3.1 Linkers and Position Independent Code (PIC)

In a static library, the linker simply copies needed compilation units from the library to the executable. The static library can subsequently be deleted with no adverse affects to the executable. This also causes common libraries to be duplicated in every executable that links against it, and for the resulting executables to be quite large.

In a shared library, the linker simply inserts in the executable enough information to find the library and load it when the executable is invoked. This typically happens before the program ever gets to `main()`. This keeps executables small and allows commonly used libraries to be reused without copying, but it also means that the executable can fail if the library is renamed, moved, deleted, or even if the user's environment changes sufficiently.

A necessary (but not sufficient) condition for shared libraries to work is that all the compilation units (`*.o`) contained must be explicitly compiled as *position independent code*(PIC). Position independent code has an added level of indirection in critical areas since details (such as addresses to jump to in subroutine calls) are not known until runtime. Even though shared libraries are very useful, PIC causes a small but measurable degradation in performance, making static linked libraries with non-PIC code a viable option for performance-critical situations.

A dynamic-linked library is a shared library with one added feature, it can be loaded explicitly by the user at runtime by passing the string name into `dlopen()`. Dynamic-linked libraries (DLL's) also require compilation as PIC, though many compilers (including GCC) have special commands for each[1].

15.3.2 Tracking Down Problems

When tracking down problems with Babel libraries, to UNIX tools **nm** and **ldd** are your friends. **nm** will print the list of linker symbols in a file, including details such as whether the symbol is defined or not. **ldd** lists dynamic dependencies of a shared libraries or executables, indicating where it will look for these symbols when loaded.

Recall the Fortran hello world example in section 15.2.1. Even though we may think this is all done with static linking, using these tools we find out the truth.

```
% ldd a.out
 libg2c.so.0 => /usr/local/gcc/3.2/lib/libg2c.so.0 (0x400180000)
 libm.so.6 => /lib/i686/libm.so.6 (0x4004a000)
 libgcc_s.so.1 => //usr/local/gcc/3.2/lib/libgcc_s.so.1 (0x4006d000)
 libc.so.6 => /lib/i686/libc.so.6 (0x40076000)
 /lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x40000000)
```

Here, we clearly see that five libraries are shared libraries that will be loaded after the executable is invoked, but before we get to the main program. Some of these libraries make sense: `libg2c` is a Fortran to C support library, `libc` is the C standard library, but why is `libm` listed... its a library of transcendental functions (e.g. sin(), cos()) why would it be included? The answer becomes obvious when using ldd on `libg2c`. The fortran support library has dependencies on the math library, so our FORTRAN executable inherits that dependency too.

```
% nm a.out | grep ' U '
 U __cxa_atexit@@GLIBC_2.1.3
 U __libc_start_main@@GLIBC_2.0
 U do_lio
 U e_wsle
 U exit@@GLIBC_2.0
 U f_exit
 U f_init
 U f_setarg
 U f_setsig
```

[1] `-fpic` for SO's, `-fPIC` for DLL's

```
U s_stop
U s_wsle
```

nm (and grep) shows us 11 symbols that are were left undefined in our final hello world application. A little more nm—greping about will help us find that symbols starting with `f_` are defined in libg2c.

15.4 SIDL Library Issues

As mentioned in Section 5.5, the Babel toolkit includes the SIDL runtime library. The library provides a base interface, class, and exception as the foundation. This is how Babel provides object-oriented features to non-object-oriented languages. In order to support the runtime system and build the SIDL library, it also provides DLL and Loader classes.

Babel generated code depends critically on `babel_config.h` to correctly define a lot of platform specific details. One detail that changes too frequently to encode in `babel_config.h` is whether or not the software is being compiled is position independent code (PIC). This detail is commonly added to the compilation instruction using the flags (e.g. `-fPIC -DPIC` [2]). The first flag tells the compiler to generate position independent code. The second defines the preprocessor macro `PIC`. Looking now at `babel_config.h`, we see that either `SIDL_DYNAMIC_LIBRARY` or `SIDL_STATIC_LIBRARY` are defined depending on whether or not `PIC` is defined.

As described in Section 15.3.1, Babel tends to focus on static libraries and dynamic linked libraries; not worrying much about shared libraries. The main reason is that for every last drop of performance, people would want static libraries. To support Java and Python (and the CCA model) dynamic loading is required. There's no real benefit to doing shared libraries that can't be dynamically loaded, so in developing Babel, we focus on the other two linkage situations.

15.5 Language Bindings for the `sidl` Package

The implementation and C stubs for the `sidl` package are stored in `libsidl.so` and `libsidl.a`, shared and static libraries that are installed when you install babel. You can determine the directory where these libraries are stored by running `babel-config --libdir`. Normally, running `babel-config --libdir` will yield something like `/usr/lib` or `/usr/local/lib`; however, your system administrator may have chosen a different directory by specifying a `--prefix` when they configured Babel (see Section 2.1.1). The IOR header files and C stub header files are installed in the directory shown by `babel-config --includedir`.

Babel also provides precompiled stubs for the `sidl` package for the C++, F77, F90, Java and UC++ language bindings. These libraries are also installed in `babel-config --libdir`, and they are named `libsidlstubs_cxx.so`, `libsidlstubs_ucxx.so`, Codelibsidlstubs_f77.so, and `libsidlstubs_f90.so`. Similarly named static archives and libtool `.la` files are also inalled in `babel-config --libdirst`. The header files for these languages are installed in subdirectories of `babel-config --includedir` named `Cxx`, `F77`, `F90`, and `UCxx`.

15.6 SCL Files for Dynamic Loading

If you generate a dynamic-linked library containing implementations of SIDL classes, you must also generate a SIDL Class List file (SCL file). An SCL file contains metadata about zero or more dynamic-linked libraries; for each dynamic-linked library, the SCL file has the list of SIDL classes implemented in that library. The `sidl.Loader.findLibrary` method searches SCL files when trying to find the implementation (or some other aspect) of a SIDL class.

The SCL file is an XML file with three kinds of elements. The top level element is `scl` which contains zero or more `library` elements. The `library` element has several attributes, and it contains zero or more `class` elements. The `library` element has three required attributes, `uri`, `scope` and `resolution`, and two optional attributes, `md5` and `sha1`. The `uri` is a local filename including path or a network `url` indicating where the library is stored. The `scope` attribute allows developers to suggest whether the library should be loaded in a `local` or the `global` namespace. The developer can suggest `lazy` or `now` symbol resolution using the `scope` attribute. The `md5` and `sha1` are optional message digests to confirm that the library has not been modified or replaced. The `class` element has two required elements, `name` and `desc`. The `name` field is the name of the class, and `desc` indicates what kind of

[2] The actual command to the compiler varies, `-fPIC` is understood by GCC

information is held in the library. Each class contained in the dynamic-linked library should be listed in the SCL file. For now, the only `desc` values with standardized meanings of `ior/impl`, `java` and `python/impl`. `ior/impl` indicates the dynamic-linked library contains the IOR object and implementation for the class, and `java` indicates that the library has the Java JNI wrapper object code. `python/impl` has the Python skeletons and implementation libraries.

Here is an the SCL file for the SIDL runtime library installed in `/usr/local`.

```
<?xml version="1.0" ?>
<scl>
  <library uri="/usr/local/lib/libsidl.la" scope="global" resolution="now" >
    <class name="SIDL.BaseClass" desc="ior/impl" />
    <class name="SIDL.ClassInfoI" desc="ior/impl" />
    <class name="SIDL.DLL" desc="ior/impl" />
    <class name="SIDL.Loader" desc="ior/impl" />
    <class name="SIDL.Boolean" desc="java" />
    <class name="SIDL.Character" desc="java" />
    <class name="SIDL.DoubleComplex" desc="java" />
    <class name="SIDL.Double" desc="java" />
    <class name="SIDL.FloatComplex" desc="java" />
    <class name="SIDL.Float" desc="java" />
    <class name="SIDL.Integer" desc="java" />
    <class name="SIDL.Long" desc="java" />
    <class name="SIDL.Opaque" desc="java" />
    <class name="SIDL.SIDLException" desc="ior/impl" />
    <class name="SIDL.String" desc="java" />
  </library>
</scl>
```

It's worth noting that the `uri` can be a libtool metadata file (`.la`) when the library is located on the local file system or a dynamic-linked library file (`.so` or another machine dependent suffix). You cannot have a libtool `.la` when the library is remote (e.g., `ftp:` or `http:`) because libtool expects the files references in the `.la` file to be local and in particular directories.

15.7 Deployment of Babel Enabled Libraries

At this point, there is no standard — or even recommended — model for deploying Babel enabled libraries. Below are a few examples of how our developer-customers are currently packaging their code.

Server Source Only With this option your users are expected to have Babel installed on their system. In this mode, developers simply include a SIDL file and their corresponding implementation files. The user in this case must build the software, call Babel to generate the client bindings in the language of choice, and link it all together into a final application.

Client and Server Source This option tries to hide Babel as much as possible. In this mode, the developer pre-generates many different client language bindings and distributes them along with their code and the sources for the Babel runtime library. Then the user has a "batteries included" package that's ready to run out of the box. The user may not even be aware that Babel has been used unless they pay careful attention to how the package was built.

Server Libraries Only Finally, in this mode only the SIDL file and the precompiled shared library files are distributed. This is not an open-source solution, though users still need to build the language bindings to access the shared library.

Chapter 16

Troubleshooting

Contents

16.1 Introduction

This appendix provides an overview of common problems that Babel users have encountered. Additional insights may be found in Chapter 17.

16.2 Common Errors

This section focuses on common errors encountered by Babel users. The errors have been separated into those related to SIDL parsing, XML parsing, and compilation.

SIDL Parsing Errors

- ***Babel: Error: when trying to resolve remaining args...Error : ÄnArgumentf̈ails to resolve as a symbol or file.*** For a symbol, Babel attempts to find it in the repository(ies) specified on the command line or, if none specified, in the default repository. Check the repository being used to ensure that XML exists for the appropriate version of the symbol. If it is not present, generate the XML for it first then try again.

XML Parsing Errors

Compilation Errors

16.3 Common Warnings

This section focuses on common warnings encountered by Babel users. Again, warnings have been separated into those related to SIDL parsing, XML parsing, and compilation.

SIDL Parsing Warnings

- ***Babel: Warning: When creating repository...File R̈epository+Fileïs not a repository directory”.*** First verify that the specified directory is actually a repository directory. That is, that it contains symbol interfaces defined by XML files. If not, correct the repository option then try again.

XML Parsing Warnings

Compilation Warnings

Chapter 17

Lessons Learned

Contents

17.1 Introduction

This appendix focuses on providing tips, tricks, and advice submitted by Babel/SIDL users. We have generally provided the information verbatim.

17.2 Compilation Consistency is Key

Steve Smith, 24 September 2001

Basically ”be consistent” is the biggest lesson we found.

When compiling C++ codes, you may have conflicts if you use different compile options. Under KCC we found -no_exceptions caused problems if parts were compiled with/without the flag. There are most likely other compile flags which turn features on/off which would cause similar problems. This caused a core dump immediately when core file was loaded. This is somewhat obvious but if you are linking together several different codes from a variety of developers you need to examine the compile flags very carefully. This problem is probably more likely with C++ due to the greater number of code generation options (e.g. RTTI, exceptions etc).

A much more subtle problem occurred when we had a C shared library which called functions in a C++ shared library. We initially used gcc to create the C shared library and KCC to create the C++ shared library. The application would core dump when a dynamic cast was attempted. This was solved by using the ”cc” compiler wrapper that is part of the KCC distribution (which uses the native ”cc”). So you need to be aware of not only what is in your .so and how it is compiled but all the .so’s that you are using.

If you have several versions of a library, say during a debugging process, make sure you are using the correct versions of things. The ”ldd” command is very useful for making sure you getting the shared libraries that you think you should be linking to. Along these lines, keep your LD_LIBRARY_PATH as simple as possible when debugging.

In retrospect this does not look like a large number of problems, but figuring out the second problem took a long time since I focused on how the C++ library was being created rather than where the real problem was being introduced. It wasn’t until after I had exhausted a long list of other potential conflicts that I started messing with the C library compilation.

Part IV

Appendices

Appendix A

Reserved Words

Contents

A.1 Introduction

This appendix lists SIDL's reserved words. Other words and constructs that are problematic in particular language bindings are also listed.

A.2 Reserved Words

Table A.1 lists all the words that are part of the SIDL grammar and cannot be used as a package, enum, interface, class, or argument name.

A.3 Suggested Things To Avoid

Since SIDL maps onto many other languages there are a great number of words and constructs that are harmless in SIDL, but cause great trouble in generated language bindings. We list known problems in Table A.2.

In addition, the following should be avoided:

- Reserved words in all of the supported languages. This is a long list only some of which appear here.
- Methods with the same name as a class (this is a constructor in C++).
- Packages, Classes, Interfaces, Methods or Arguments that differ only by case. Not all languages are case sensitive but, since Babel's focus is language interoperability, Babel must make allowances.

Table A.1: SIDL Reserved Words

RESERVED WORD	ROLE
abstract	optional modifier for *class*
array	datatype
bool	builtin datatype
char	builtin datatype
class	user defined datatype
copy	(future) argument modifer
dcomplex	builtin datatype
double	builtin datatype
enum	user defined datatype
extends	inheritance mode
fcomplex	builtin datatype
final	package and method modifier
float	builtin datatype
implements	inheritance mode
implements-all	inheritance mode
import	bring other packages into current scope
in	argument mode
inout	argument mode
int	builtin datatype
interface	user defined datatype
local	(future) method modifier
long	builtin datatype
oneway	(future) method modifier
opaque	builtin datatype
out	argument mode
package	scoping construct
static	method modifier
string	builtin datatype
throws	exception declaration
version	assign version number to package
void	declares method as not returning a type

Table A.2: Other words/constructs to avoid

WORD	**C**	**C++**	**Java**	**Python**	**word**	**C**	**C++**	**Java**	**Python**
abstract			X		lambda				X
and		X		X	long	X	X	X	
and_eq		X			mutable		X		
asm	X	X			namespace		X		
assert				X	native			X	
auto	X	X			new		X	X	
bitand		X			not		X		X
bitor		X			not_eq		X		
bool		X			null			X	
boolean			X		operator		X		
break	X	X	X	X	or		X		X
case	X	X	X		or_eq		X		
catch		X	X		package			X	
char	X	X	X		pass				X
class		X	X		print				X
compl		X			private		X	X	
const	X	X	X		protected		X	X	
const_cast		X			public		X	X	
continue	X	X	X	X	raise				X
def				X	register	X	X		
default	X	X	X		reinterpret_cast		X		
del				X	return	X	X	X	X
delete		X			short	X	X	X	
do	X	X	X		signed	X	X		
double	X	X	X		sizeof	X	X		
dynamic_cast		X			static	X	X	X	
elif				X	static_cast		X		
else	X	X	X	X	strictfp			X	
enum	X	X			struct	X	X		
except				X	super			X	
exec				X	switch	X	X	X	
explicit		X			synchronized			X	
export		X			template		X		
extends			X		this		X	X	
extern	X	X			throw		X	X	
false		X	X		throws			X	
final			X		transient			X	
finally			X	X	true		X	X	
float	X	X	X		try		X	X	X
for	X	X	X	X	typedef	X	X		
friend		X			typeid		X		
from				X	typename		X		
global				X	union	X	X		
goto	X	X	X		unsigned	X	X		
if	X	X	X	X	using		X		
implements			X		virtual		X		
import			X		void	X	X	X	
inline		X			volatile	X	X	X	
instanceof			X		wchar_t		X		
int	X	X	X		while	X	X	X	X
interface			X		xor		X		
is				X	xor_eq		X		

Appendix B

SIDL Grammar

Contents

B.1 Introduction

This appendix provides an overview of the Scientific Interface Definition Language (SIDL) grammar. For simplicity, the grammar is described in extended BNF.

B.2 Backus-Naur Form

The grammar described here was extracted from the JavaCC productions defined in the Babel source code. Since the comments associated with the productions appeared to be sufficiently descriptive, they have been retained to serve as the explanation of the key productions.

```
/*
 * The following lexical tokens are ignored.
 */
SKIP : {
   < " " >
 | < "\n" >
 | < "\r" >
 | < "\t" >
 | < "//" (~["\n","\r"])* ("\n" | "\r" | "\r\n") >
 | < "/**/" >
 | < "/*" (~["*"])+ "*" ("*" | ~["*","/"] (~["*"])* "*")* "/" >
   { checkComment(image, input_stream.getBeginLine(),
                  input_stream.getEndLine()); }
 | < "[" >
 | < "]" >
}

/*
 * The following lexical states define the transitions necessary to
 * parse documentation comments.  Documentation comments may appear
 * anywhere in the file, although they are only saved if they preceed
 * definition or method productions.  Documentation comments are
 * represented by "special tokens" in the token list.
```

```
 */
SPECIAL_TOKEN : {
   < T_COMMENT : "/**" > : BEGIN_DOC_COMMENT
}

<BEGIN_DOC_COMMENT> SKIP : {
   < " " >
 | < "\t" >
 | < "*/" >                   : DEFAULT
 | < ("\n" | "\r" | "\r\n") > : LINE_DOC_COMMENT
 | < "" >                     : IN_DOC_COMMENT
}

<LINE_DOC_COMMENT> SKIP : {
   < " " >
 | < "\t" >
 | < "*/" >      : DEFAULT
 | < "*" (" ")?> : IN_DOC_COMMENT
 | < "" >        : IN_DOC_COMMENT
}

<IN_DOC_COMMENT> SPECIAL_TOKEN : {
   < "*/" >                   { trimMatch(matchedToken); } : DEFAULT
 | < ("\n" | "\r" | "\r\n") > { trimMatch(matchedToken); } : LINE_DOC_COMMENT
}

<IN_DOC_COMMENT> MORE : {
   < ~[] >
}

/*
 * The following keywords are the lexical tokens in the SIDL grammar.
 */
TOKEN : {
   < T_ABSTRACT       : "abstract" >
 | < T_CLASS          : "class" >
 | < T_COPY           : "copy" >
 | < T_ENUM           : "enum" >
 | < T_EXTENDS        : "extends" >
 | < T_IMPORT        : "import" >
 | < T_IN             : "in" >
 | < T_INOUT          : "inout" >
 | < T_FINAL          : "final" >
 | < T_IMPLEMENTS     : "implements" >
 | < T_IMPLEMENTS_ALL : "implements-all" >
 | < T_INTERFACE      : "interface" >
 | < T_LOCAL          : "local" >
 | < T_ONEWAY         : "oneway" >
 | < T_OUT            : "out" >
 | < T_PACKAGE        : "package" >
 | < T_REQUIRE        : "require" >
 | < T_STATIC         : "static" >
 | < T_THROWS         : "throws" >
 | < T_VERSION        : "version" >
 | < T_VOID           : "void" >

 | < T_ARRAY          : "array" >
 | < T_RARRAY         : "rarray" >
 | < T_BOOLEAN        : "bool" >
```

```
 | < T_CHAR           : "char" >
 | < T_DCOMPLEX       : "dcomplex" >
 | < T_DOUBLE         : "double" >
 | < T_FCOMPLEX       : "fcomplex" >
 | < T_FLOAT          : "float" >
 | < T_INT            : "int" >
 | < T_LONG           : "long" >
 | < T_OPAQUE         : "opaque" >
 | < T_STRING         : "string" >

 | < T_IDENTIFIER     : <T_LETTER> (<T_LETTER> | <T_DIGIT> | "_")* >
 | < T_VERSION_STRING : <T_INTEGER> ("." <T_INTEGER>)+ >
 | < T_INTEGER        : (["-","+"])? (<T_DIGIT>)+ >
 | < T_DIGIT          : ["0"-"9"] >
 | < T_LETTER         : ["a"-"z","A"-"Z"] >

 | < T_CLOSE_ANGLE    : ">" >
 | < T_CLOSE_CURLY    : "}" >
 | < T_CLOSE_PAREN    : ")" >
 | < T_COMMA          : "," >
 | < T_EQUALS         : "=" >
 | < T_OPEN_ANGLE     : "<" >
 | < T_OPEN_CURLY     : "{" >
 | < T_OPEN_PAREN     : "(" >
 | < T_SEMICOLON      : ";" >
 | < T_SCOPE          : "." >

 | < T_COLUMN_MAJOR   : "column-major" >
 | < T_ROW_MAJOR      : "row-major" >

 | < T_CATCH_ALL      : ~[] >
}

/**
 * A SIDL Specification contains zero or more version productions followed
 * by zero or more import productions followed by zero or more package
 * productions followed by the end-of-file.  Before leaving the specification
 * scope, resolve all references in the symbol table.
 */
Specification ::= ( Require )* ( Import )* ( Package )* <EOF>

/**
 * A SIDL Require production begins with a "require" token and is followed
 * by a scoped identifer, a "version" token, and a version number.  The
 * scoped identifier must be not defined.  The version number is specified
 * in the general form "V1.V2...Vn" where Vi is a non-negative integer.
 */
Require ::=
  <T_REQUIRE> ScopedIdentifier
  <T_VERSION> ( <T_INTEGER> | <T_VERSION_STRING> ) <T_SEMICOLON>

/**
 * A SIDL Import production begins with an "import" token and is followed
 * by a scoped identifier which is optionally followed by a "version" token
 * and a version number.  The scoped identifier must be defined and it must
 * be a package.  The version number is specified in the general form
 * "V1.V2...Vn" where Vi is a non-negative integer.  A particular package
 * may only be included in one import statement.  The import package name
 * is added to the default search path.  At the end of the parse, any import
```

```
 * statements that were not used to resolve a symbol name are output as
 * warnings.
 */
Import ::=
  <T_IMPORT> ScopedIdentifier
  [ <T_VERSION> ( <T_INTEGER> | <T_VERSION_STRING>) ] <T_SEMICOLON>

/**
 * The SIDL package specification begins with a "package" token followed by
 * a scoped identifier.  The new package namespace begins with an open curly
 * brace, a set of zero or more definitions, and a close curly brace.  The
 * closing curly brace may be followed by an optional semicolon.  The package
 * identifier must have a version defined for it, and it must not have been
 * previously defined as a symbol or used as a forward reference.  The parent
 * of the package must itself be a package and must have been defined.  The
 * symbols within the curly braces will be defined within the package scope.
 */
Package ::=
  [ <T_FINAL> ] <T_PACKAGE> ScopedIdentifier
  [ <T_VERSION> ( <T_INTEGER> | <T_VERSION_STRING> ) ]
  <T_OPEN_CURLY> ( Definition )* <T_CLOSE_CURLY> [ <T_SEMICOLON> ]

/**
 * A SIDL Definition production consists of a class, interface, enumerated
 * type, or package.
 */
Definition ::= ( Class | Enum | Interface | Package )

/**
 * A SIDL class specification begins with an optional abstract keyword
 * followed by the class token followed by an identifier.  The abstract
 * keyword is required if and only if there are abstract methods in the
 * class.  The class keyword is followed by an identifer.  The identifier
 * string may not have been previously defined, although it may have been
 * used as a forward reference.  The identifier string may be preceeded
 * by a documentation comment.  A class may optionally extend another class;
 * if no class is specified, then the class will automatically extend the
 * SIDL base class (unless it is itself the SIDL base class).  Then parse
 * the implements-all and implements clauses.  The interfaces parsed during
 * implements-all are saved in a set and then all those methods are defined
 * at the end of the class definition.  The methods block begins with an
 * open curly-brace followed by zero or more methods followed by a close
 * curly-brace and optional semicolon.
 */
Class ::=
  [ <T_ABSTRACT> ] <T_CLASS> Identifier
  [ <T_EXTENDS> ScopedIdentifier ]
  [ <T_IMPLEMENTS_ALL> AddInterface ( <T_COMMA> AddInterface )* ]
  [ <T_IMPLEMENTS> AddInterface ( <T_COMMA> AddInterface )* ]
  <T_OPEN_CURLY> ( ClassMethod )* <T_CLOSE_CURLY> [ <T_SEMICOLON> ]

/**
 * The SIDL enumeration specification begins with an "enum" token followed by
 * an identifier.  The enumerator list begins with an open curly brace, a set
 * of one or more definitions, and a close curly brace.  The closing curly
 * brace may be followed by an optional semicolon.  The enumeration symbol
 * identifier must have a version defined for it, and it must not have been
 * previously defined as a symbol.  Forward references are not allowed for
 * enumerated types.  This routine creates the enumerated class and then
```

```
 * grabs the list of enumeration symbols and their optional values.
 */
Enum ::=
  <T_ENUM> Identifier <T_OPEN_CURLY> Enumerator ( <T_COMMA> Enumerator )*
  <T_CLOSE_CURLY> [ <T_SEMICOLON> ]

/**
 * The SIDL enumerator specification consists of an identifier followed
 * by an optional assignment statement beginning with an equals and followed
 * by an integer value.  This routine adds the new enumeration symbol to
 * the list and then returns.
 */
Enumerator ::= Identifier [ <T_EQUALS> <T_INTEGER> ]

/**
 * A SIDL interface specification begins with the interface token followed
 * by an identifier.  An interface may have an extends block consisting of
 * a comma-separated sequence of interfaces.  The methods block begins with
 * an open curly-brace followed by zero or more methods followed by a close
 * curly-brace and optional semicolon.  Interfaces may be preceeded by a
 * documentation comment.  The identifier string may not have been previously
 * defined, although it may have been used as a forward reference.  If the
 * interface does not extend another interface, then it must extend the base
 * SIDL interface (unless, of course, this is the definition for the base
 * SIDL interface).
 */
Interface ::=
  <T_INTERFACE> Identifier [ <T_EXTENDS> AddInterface
  ( <T_COMMA> AddInterface )* ]
  <T_OPEN_CURLY> ( InterfaceMethod )* <T_CLOSE_CURLY> [ <T_SEMICOLON> ]

/**
 * This production parses the next scoped identifier and validates that
 * the name exists and is an interface symbol.  Then each of its methods
 * are checked for validity with the existing methods.  If everything
 * checks out, then the new interface is added to the existing object.
 */
AddInterface ::= ScopedIdentifier

/**
 * This production parses the SIDL method description for a class method.
 * A class method may start with abstract, final, or static.  An error is
 * thrown if the method has already been defined in the class object or if
 * the method name is the same as the class name.  An error is also thrown
 * if a method has been defined in a parent class and (1) the signatures
 * do not match, (2) either of the methods is static, (3) the existing method
 * is final, or (4) the new method is abstract but the existing method was
 * not abstract.
 */
ClassMethod ::= [ ( <T_ABSTRACT> | <T_FINAL> | <T_STATIC> ) ] Method

/**
 * This method parses a SIDL method and then checks whether it can be
 * added to the interface object.  An error is thrown if the method has
 * already been added to the interface object or if the method name is
 * the same as the interface name.  An error is also thrown if a previous
 * method was defined with the same name but a different signature.
 */
InterfaceMethod ::= Method
```

```
/**
 * The SIDL method production has a return type, a method identifier,
 * an optional argument list, an optional communication modifier, and
 * an optional throws clause.  The return type may be void (no return
 * type) or any valid SIDL type.  The method is built piece by piece.
 */
Method ::=
  ( <T_VOID> | [ <T_COPY> ] Type() ) Identifier [ <T_IDENTIFIER> ]
  <T_OPEN_PAREN> [ Argument ( <T_COMMA> Argument )* ] <T_CLOSE_PAREN>
  [ <T_LOCAL> | <T_ONEWAY> ] [ <T_THROWS> ScopedIdentifier
  ( <T_COMMA> ScopedIdentifier )* ] <T_SEMICOLON>

/**
 * Parse a SIDL argument.  Arguments begin with an optional copy modifier
 * followed by in, out, or inout followed by a type and a formal argument.
 * The argument is returned on the top of the argument stack.  This routine
 * also checks that the copy modifier is used only for symbol objects.  For
 * all other types, copy is redundant.
 */
Argument ::= [ <T_COPY> ] ( <T_IN> | <T_OUT> | <T_INOUT> )
 (Type Identifier | Rarray)

/**
 * A SIDL type consists of one of the standard built-in types (boolean,
 * char, dcomplex, double, fcomplex, float, int, long, opaque, and string),
 * a user-defined type (interface, class, or enum), or an array.  This
 * production parses the type and pushes the resulting type object on
 * the top of the argument stack.
 */
Type ::=
  ( <T_BOOLEAN>
  | <T_CHAR>
  | <T_DCOMPLEX>
  | <T_DOUBLE>
  | <T_FCOMPLEX>
  | <T_FLOAT>
  | <T_INT>
  | <T_LONG>
  | <T_OPAQUE>
  | <T_STRING>
  | Array
  | SymbolType )

/**
 * Parse an array construct and push the resulting type and ordering
 * on top of the stack.  Only dimensions one through MAX_ARRAY_DIM
 * (inclusive) are supported.
 */
Array ::=
  <T_ARRAY> <T_OPEN_ANGLE> Type [ <T_COMMA> ( <T_INTEGER>
  [ <T_COMMA> ( <T_COLUMN_MAJOR> | <T_ROW_MAJOR> ) ]
  | ( <T_COLUMN_MAJOR>| <T_ROW_MAJOR> ) ) ] <T_CLOSE_ANGLE>

/**
 * Parse an rarray construct and push the resulting type and ordering
 * on top of the stack.  Only dimensions one through MAX_ARRAY_DIM
 * (inclusive) are supported.  And don’t forget the indicies!
 */
```

```
Rarray ::= <T_RARRAY> <T_OPEN_ANGLE> Type [ <T_COMMA> <T_INTEGER> ]
       <T_CLOSE_ANGLE> Identifier
       <T_OPEN_PAREN> Identifier ( <T_COMMA Identifier )*
       <T_CLOSE_PAREN>

/**
 * This production parses a scoped identifier and verifies that it is
 * either a forward reference or a symbol that may be used as a type
 * (either an enum, an interface, or a class).
 */
SymbolType ::= ScopedIdentifier

/**
 * All SIDL scoped names are of the general form "ID ( . ID )*".  Each
 * identifier ID is a string of letters, numbers, and underscores that
 * must begin with a letter.  The scope resolution operator "." separates
 * the identifiers in a name.
 */
ScopedIdentifier ::= Identifier ( <T_SCOPE> Identifier )*

/**
 * A SIDL identifier must start with a letter and may be followed by any
 * number of letters, numbers, or underscores.  It may not be a reserved
 * word in any of the SIDL implementation languages (e.g., C or C++).
 */
Identifier ::= <T_IDENTIFIER>
```

Appendix C

Extensible Markup Language (XML)

Contents

C.1 Introduction

This appendix describes the XML representation of SIDL interfaces. Since the format of an XML file is dictated by a Document Type Declaration (DTD) file, the description will focus on the DTD associated with SIDL.

C.2 SIDL Document Type Declaration (DTD)

Babel relies on several DTDs to describe and enforce the layout of conformant XML files. The DTD of primary importance for Babel is `SIDL.dtd` because it describes the requisite tags and attributes corresponding to SIDL files. The contents of the DTD are given below.

```
<?xml version="1.0" encoding="UTF-8"?>
<!--
   File:        sidl.dtd
   Package:     sidl XML
   Release:     $Name:  $
   Revision:    @(#) $Id: SIDL.dtd,v 1.3 2005/01/20 05:32:25 epperly Exp $
   Description: DTD for the sidl XML database representation

   Copyright (c) 2000-2003, The Regents of the University of Calfornia.
   Produced at the Lawrence Livermore National Laboratory.
   Written by the Components Team <components@llnl.gov>
   UCRL-CODE-2002-054
   All rights reserved.

   This file is part of Babel. For more information, see
   http://www.llnl.gov/CASC/components/. Please read the COPYRIGHT file
   for Our Notice and the LICENSE file for the GNU Lesser General Public
   License.

   This program is free software; you can redistribute it and/or modify it
   under the terms of the GNU Lesser General Public License (as published by
   the Free Software Foundation) version 2.1 dated February 1999.
```

```
   This program is distributed in the hope that it will be useful, but
   WITHOUT ANY WARRANTY; without even the IMPLIED WARRANTY OF
   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the terms and
   conditions of the GNU Lesser General Public License for more details.

   You should have recieved a copy of the GNU Lesser General Public License
   along with this program; if not, write to the Free Software Foundation,
   Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA

   This file describes the DTD for a sidl symbol represented in XML format.
   The root element is <Symbol>.

   PUBLIC ID "-//CCA//sidl Symbol DTD v1.1//EN"
-->

<!--
   Symbol Element

   Symbol is the root element for all sidl XML schema.  The Symbol contains a
   SymbolName (fully qualified symbol name and version), Metadata, Comment,
   and one of Class, Enumeration, Interface, or Package.
-->

<!ENTITY % symbols "Class | Enumeration | Interface | Package">
<!ELEMENT Symbol (SymbolName, Metadata, Comment, (%symbols;))>

<!--
   SymbolName Element

   A SymbolName represents a fully qualified symbol name along with its
   version.  It is of the form:

   <SymbolName name="sidl.SomeName" version="1.3.4"/>
-->

<!ELEMENT SymbolName EMPTY>
<!ATTLIST SymbolName name     CDATA #REQUIRED
                     version  CDATA #REQUIRED>

<!--
   Metadata Element

   The Metadata element contains any useful descriptive data about the symbol.
   The time and date of creation is required, but all other information is
   optional.  The date and time must follow the ISO-8601 standard.  The
   entries in the metadata element are (key,value) pairs.
-->

<!ELEMENT Metadata (MetadataEntry)*>
<!ATTLIST Metadata date CDATA #REQUIRED>

<!ELEMENT MetadataEntry EMPTY>
<!ATTLIST MetadataEntry key   CDATA #REQUIRED
                        value CDATA #REQUIRED>

<!--
   Comment Element
```

```
   Comment elements support a very simple HTML description using the
   html-lite.dtd HTML subset.  See html-lite.dtd for more details.
-->

<!ENTITY % html-lite PUBLIC "-//CCA//sidl HTML DTD v1.0//EN" "html-lite.dtd">
%html-lite;

<!ELEMENT Comment %html-block;>

<!--
   Package Element

   The Package element contains the symbols that exist within a package.
   In the PackageSymbol element, note that the name is relative to the
   package (thus, sidl.Class is represented by Class within package sidl).

   A true final attribute indicates that this package is not reentrant. It
   defaults to true to handle old XML files. In previous versions, all
   packages were non-reentrant.
-->

<!ELEMENT Package (PackageSymbol)*>
<!ATTLIST Package final (false | true) "true">

<!--
If the version attribute isn’t provided, the symbol has the same version
as the containing package. This is to provide backward compatibility with
previous released versions of the DTD. Someday the version may become
REQUIRED, so always include it.
-->
<!ELEMENT PackageSymbol EMPTY>
<!ATTLIST PackageSymbol name CDATA                               #REQUIRED
                        type (class | enum| interface | package) #REQUIRED
                        version CDATA                            #IMPLIED>

<!--
   Enumeration Element

   The Enumeration element consists of a collection of Enumerator elements
   that describe a relative symbol name, its integer value, and whether the
   value was assigned by the parser or in the sidl input file.
-->

<!ELEMENT Enumeration (Enumerator)+>

<!ELEMENT Enumerator (Comment)?>
<!ATTLIST Enumerator name     CDATA          #REQUIRED
                     value    CDATA          #REQUIRED
                     fromuser (false | true) #REQUIRED>

<!--
   Class Element

   The Class element consists of a class extended by this class, a block
   of interfaces implemented by this class, and a block of methods declared
   or defined by this class.  The methods block does not include methods
   declared or defined by parents.  The elements AllParentInterfaces and
   AllParentClasses include all parents of this class.
-->
```

```
<!ELEMENT Class (Extends, ImplementsBlock,
                 AllParentClasses, AllParentInterfaces,
                 MethodsBlock)>
<!ATTLIST Class abstract (false | true) #REQUIRED>

<!ELEMENT Extends (SymbolName)?>

<!ELEMENT ImplementsBlock (SymbolName)*>

<!--
   Interface Element

   The Interface element constsis of a block of interfaces that this
   interface extends (element ExtendsBlock) and a block of methods
   declared by this interface (element MethodsBlock).  The methods block
   element contains only those method declared or re-declared by this
   interface and does not include all those methods defined by the
   parent interfaces.  The AllParentInterfaces element block includes
   all parent interfaces that this interface extends.
-->

<!ELEMENT Interface (ExtendsBlock, AllParentInterfaces, MethodsBlock)>

<!ELEMENT ExtendsBlock (SymbolName)*>

<!--
   AllParentClasses and AllParentInterfaces Elements

   These elements define a collection of zero or more SymbolName elements
   that are the parent classes and parent interfaces of a class or interface.
-->

<!ELEMENT AllParentClasses (SymbolName)*>

<!ELEMENT AllParentInterfaces (SymbolName)*>

<!--
   MethodsBlock Element

   The MethodsBlock element defines a collection of zero or more methods
   that belong to a sidl interface or class.
-->

<!ELEMENT MethodsBlock (Method)*>

<!--
   Method Element

   The Method element defines a single method in a class or interface.
   The method is defined by a return type (the Type element), a return
   mode (the copy attribute of Method), a method name, an argument list,
   a throws clause, definition mode modifiers, and communication mode
   modifiers.
-->

<!ELEMENT Method (Comment, Type, ArgumentList, ThrowsList)>
<!ATTLIST Method shortname     CDATA                                #REQUIRED
                 extension     CDATA                                #REQUIRED
```

```
                 copy          (false  | true)                        #REQUIRED
                 definition    (normal | abstract | final | static) #REQUIRED
                 communication (normal | local    | oneway)           #REQUIRED>

<!ELEMENT ArgumentList (Argument)*>

<!ELEMENT ThrowsList (SymbolName)*>

<!--
   Argument Element

   The sidl Argument element defines a sidl argument, which consists
   of a copy modifier, a parameter passing mode (in, inout, or out),
   a parameter type, and a formal parameter name.
-->

<!ELEMENT Argument (Type)>
<!ATTLIST Argument copy (false | true)     #REQUIRED
                   mode (in | inout | out) #REQUIRED
                   name CDATA              #REQUIRED>

<!--
   Type Element

   The Type element describes a sidl type, which may be a built-in type
   such as boolean or int, an array, or a user-defined symbol.  If the
   type description is a primitive type, then no sub-elements are allowed.
   If the type is a symbol, then the single sub-element must be a symbol
   name.  If the type is an array, then the single sub-element must be
   an array element
-->

<!ELEMENT Type (SymbolName | Array)?>
<!ATTLIST Type type (void     | boolean | char    | dcomplex | double |
                    fcomplex | float   | integer | long     |
                    opaque   | string  | symbol  | array    ) #REQUIRED>

<!ELEMENT Index EMPTY>
<!ATTLIST Index name CDATA #REQUIRED>

<!ELEMENT Array (Type?,Index?)>
<!ATTLIST Array order (unspecified | column-major | row-major) #REQUIRED
dim   CDATA "0" >
```

Babel assumes that comments will conform to the HTML-lite comment format. So, Babel relies on `comment.dtd` to validate whether SIDL documentation comments follow the HTML-lite comment format, which is described in `html-lite.dtd`. The most current versions of all of these DTDs can also be found in the source distribution in the `babel/compiler/gov/llnl/babel/dtds` directory.

NOTE: Any XML interface description that complies with the SIDL DTD can be used as input to Babel.

Appendix D

Glossary

abstract

OOP concept: Abstract describes something that is declared but not fully defined. For example, an abstract method is a method that is declared as a part of a class, but has no implementation. It cannot be called, it is only meant to be inherited by derived classes.

SIDL keyword: Abstract is an optional modifier for both *`class`*es and *`method`*s. An abstract method is a method that has no implementation, it's a way of declaring a method that every subclass must implement for itself. An abstract class has one or more abstract methods, and therefore cannot be instantiated.

array

Datastructure: An array is a fixed size, numerically indexed, set of variables. Arrays have in language support in almost all modern programming languages.
Babel: Babel has built in support for arrays of every data type, including objects. Babel allows these arrays such that they may be shared by differing languages.

BLAS

Basic Linear Algebra Subprograms. BLAS is a famous library for doing matrix and vector algebra. More information may be found at: http://www.netlib.org/blas/

BNF

BackusNaur Form. BNF is a formal way to describe computer languages and other formal languages.

bool

Definition: bool is a short form of the word boolean. A boolean is a logical data type that holds 1 bit of data, i.e. it is either true or false. It is used for Boolean Algebra.
SIDL keyword: bool is a data type built into SIDL, an instance of which is either true or false. For efficiency sake, the underlying storage of bool is not 1 bit.

borrowed arrays

Babel: A borrowed array is a SIDL array that does not manage its own data. The data is provided by some third party, who is also in charge of deallocating the data. It is useful for sending data through Babel, but the developer must beware in case the third party deallocates the array data before the program has finished with it.

CCA

Common Component Architecture http://www.ccaforum.org/

char

Definition: char is a short form of the word character. A character is a letter, number, puctuation mark, or other such symbol use in writing. In programming, a character is often defined by the 8 bit ASCII encoding.
SIDL keyword: char is a data type built into SIDL. It stores 1 byte of data, or enough for 1 ASCII character.

class

OOP concept: A class is a definition for a particular kind of object. It may define the data and methods that will be included in an actual instance of the object.
SIDL keyword: class is a SIDL keyword. In SIDL a class definition only defines methods. Methods may be static or instance methods. (They are instance methods by default.) If any instance method in a class is declared abstract, the class cannot be instantiated as an object, and is called an abstract class. Otherwise, it can be instantiated and is called a concrete class.

concrete class

OOP concept: A concrete class is a class where all the class's instance methods have implementations. (ie. there are no abstract methods) A concrete class may be instantiated as an object.

COM

Common Object Model http://www.microsoft.com/ Microsoft's IDL based language interoperability suite.

component

OOP concept: Components are "plug-and-play" software libraries designed with standard, clearly defined interfaces. They are the epitome of modular design. Because components communicate only through well-defined interfaces, when an application needs to be modified, a single component can be modified (or exchanged for a similar component), without fear of disrupting the other components making up the application.

component architecture

OOP concept: A component architecture defines the specifics of setting up a system for programming with components in that architecture. For example, how components are imported and how they communicate are some of the questions that must be answered in a component architecture design.

copy

SIDL keyword: copy is a SIDL keyword. It is planned that in future version of babel it will be used as a parameter modifier for parameters passed to RMI functions, currently however, this feature is unimplemented.

CORBA

Common Object Request Broker Architecture http://www.omg.org CORBA allows different programs by different vendors to communicate though an IDL interface specification. In CORBA this glue code is called the "Broker."

dcomplex

Definition: The sum of a real number and an imaginary number is called a complex number. Babel supports complex numbers as a basic type via the basic types "fcomplex" and "dcomplex."
SIDL keyword: dcomplex is a data type built into SIDL. The name is short for "double complex." It stores a complex number via 2 64-bit floating point variables, one for the real part, and one for the imaginary part.

dense

Definition: A dense array is an array where all the dimensions are "densely packed," or, in terms of memory addressing, there are no "spaces" between array elements. For example, if a one-dimensional SIDL array of 10 elements is created, it will be densely packed. However, if a slice of the array is taken with a stride of 2, the resulting array will use the same data as the original array. However, the new array will be only five elements long, and will only consist of the even elements of the original array. This is not densely packed. Example:
Array 1: 0 1 2 3 4 5 6 7 8 9
Array 2: 0 – 2 – 4 – 6 – 8 –

developer

Babel: There are two anticipated user types for Babel, both are kinds of programmers. The person referred to as the "developer" is the person developing a Babelized library. The "user" is the person who writes a program using a Babelized library.

DLL

Definition: Dynamically Linked Library. A type of library that can be linked to dynamically at runtime by passing its name as a string to the dlopen() function.

double

Definition: A double is a 64-bit floating point number.
SIDL keyword: SIDL support double as a basic type.

DTD

Document Type Definition. Defines the grammar of the XML files. http://www.w3.org/2002/xmlspec/

dynamic linking

Definition: The action of dynamically linking to DLLs at runtime.

enum

Definition: Enum is a shortend form of the word enumeration. An enumeration is used to assign numbers to a set of variable names, that is, enumerate the set of variable names.
SIDL keyword: enum is a reserved word in SIDL. It is used for defining enumerations. In Babel, enumerations are a way of binding integer constants to names.

enumeration

In Babel, enumerations are a way of binding integer constants to names. See subsection 5.3.

exception

Definition: The idea of an exception is that if a method encounters a problem it cannot handle, it interrupts its execution and “throws” and exception. Hopefully some function up the call stack will “catch” the exception and know what to do about the problem. It is a useful form of error handing that SIDL supports. Exception is not a reserved word in SIDL (but *`throw`* is).

extends

OOP concept: See inheritance.
SIDL keyword: extends is a SIDL reserved word. It is used to declare “like-type” inheritance. For example, a class may extend another class, or an interface may extend multiple interfaces, but a class cannot extend an interface, nor can an interface extend a class.

external stubs

When building a Babelized library, its also important to note if your code has dependencies to other Babel types not in your library. These types often appear as base classes, argument types, or even exception types. Your library will need stubs corresponding to all these types, so it is best to put these in your library as well. We call these external stubs. See subsection 15.2.3

external types

External Types are variable or object types that are not defined in the current class. In a class foo.Bar, sidl.Integer, or sidl.BaseClass would be external types.

fcomplex

fcomplex is a data type built into SIDL. The name is short for “float complex.” It stores a complex number via 2 32-bit floating point variables, one for the real part, and one for the imaginary part.

final

final is a SIDL reserved word. It is a method modifier. A final method is inherited by subclasses, but its implementation can never be overwritten. It is the "final" version of the implementation.

float

float is a data type built into SIDL. It is a 32-bit floating point number. float is short for floating point.

full name

Overloaded Babelized methods called from non-object oriented languages, such as C and FORTRAN 77, have 2 method names. The full name consists of the concatenation of the package name, class name, method name and type extension. The short name is missing the type extension. See subsection 5.6.

fundamental types

Fundamental types are the basic types that SIDL supports natively. bool, int, char, long, float, double, fcomplex, dcomplex, opaque, and string.

glue

Most of the code that Babel generates is "glue" code. "Glue" code sits between the caller and the implementation to allow communication between them. We use the term glue to refer to the stub, IOR, and skel files.

HTML

Hypertext Markup Language http://www.w3.org/MarkUp/

implementation

In Babel, the implementation is the code placed in the server side Impl files. It is the code that Babel used glue code to allow you to call to.

implements

implements is a SIDL reserved word. It is used when a class inherits from one or more interfaces. However, in this case the word "to implement" is not quite taken seriously. If a class implements an interface it inherits its methods, and may be cast to that interface, but if the programmer actually wished to implement any of the interface methods, he must redeclare them in the SIDL class. Any un-redeclared method is assumed abstract and will not appear in the Impl files. If there are any abstract methods in a class, that class is automatically abstract.

implements-all

implements-all is a SIDL reserved word. It takes the place of "implements." It is used when a class inherits from one or more interfaces, and the programmer definitely wants to write implementation code for each method in the named interfaces. If the programmer uses "implements-all" he does not have to redeclare the interface methods. See Section 5.6

import

import is a SIDL reserved word. It is used to bring other packages into scope. Packages may be accompanied by a version number.

in

in is a SIDL reserved word. Each parameter passed though Babel must be declared as in, out, or inout. Each of these modes has certain rules and implication associated with it. In means "pass this variable by value to the implementation." See Section 5.2.

independent arrays

Independent arrays are arrays that manage their own data. When all the references to an independent are deleted, the array data is garbage collected. The other kind of array is a borrowed array.

inheritance

In normal object-oriented programming, inheritance is the ability of a "super" or "parent" class or interface to pass its characteristics (methods and instance variables) on to its subclasses, allowing subclasses to reuse these characteristics.

Of course, in SIDL we cannot define instance variables, so in SIDL inheritance only refers to method inheritance. In SIDL inheritance is is declared with the reserved words *`extends`* and *`implements`*.

inout

inout is a SIDL reserved word. Each parameter passed though Babel must be declared as in, out, or inout. Each of these modes has certain rules and implication associated with it. Inout means "pass this variable by reference to the implementation. The implementation may do whatever it wants with the reference, but it should return something. Possibly a new variable." See Section 5.2.

instance method

An instance method is a method that must be associated with an object instance. These methods probably rely on some state in the instance, so they cannot be divorced from it. In Object Oriented languages, you call these methods on an instance, in Babelized non-OO languages like C, you pass an instance in as the first argument to one of these methods.

int

int is a data type built into SIDL. It is a 32-bit integer variable int is short for integer.

int32_t and int64_t

The ANSI C standard way of declaring an integer that is definitely 32 or 64 bits.

interface

An interface is a declaration of a set of methods with no information given about their implementation. All interface methods are abstract. An interface cannot be instantiated. However, a class may inherit from multiple interfaces. The purpose of interfaces is to give objects that are conceptually similar but internally different a common interface so that code may treat them the same, or seamlessly exchange them.

interprocess

Interprocess means "between processes." It is normally used to refer to "interprocess communication," where two or more processes find some way to communicate. Interprocess communication is one of the goals of babel with RMI.

IOR

Intermediate Object Representation. IOR code is where Babel does all its work maintaining arrays, Babel objects, reference counting, etc.

JNI

Java Native Interface. The JNI is what allows Java to call to C and C++. It is referred to as calling native code because while Java runs in a virtual machine, but C and C++ run on the real machine, or run "natively."

language interoperability

Language interoperability is Babel's main purpose. Language interoperability technology allows different computer languages to call each other methods and communicate despite problems with calling conventions and differing variable types.

local

A method (or other identifier) is considered local if it is defined or declared in the current class or method. Sometimes a more specific term like, "local to the method" or "local to the class" is used.

long

long is a data type built into SIDL. It is a 64-bit integer variable long is short for long integer. Note: Python sometimes has trouble with longs, see Section 11.7 for more details.

method

Method is the word commonly used in Java for what is called, in some other languages, a function, subroutine, or procedure. Methods are a piece a code that is called by a name. Instance methods depend on an object instance, and are allowed to read and manipulate that objects data. A static method does not depend on an instance, and therefore can only access class data and what data is passed in to the method.

namespace

A namespace is a way of logically divvying up globally accessible names. This helps in avoiding conflicts between globally accessible methods, classes, data, etc. They are mainly a feature of C++.

non-strided

A non-strided array is a dense array. See the glossary entry for dense.

Object model

The Object Model is the of rules that regulates the definition, creation, and use of classes and objects in a language. To read about the SIDL object model see Section 5.6

OMG

Object Management Group http://www.omg.org/

oneway

oneway is a SIDL reserved word. It is currently unimplemented, but it planned to be used as a method modifier.

opaque

opaque ia a data type build into SIDL. The word opaque is an adjective meaning “not transparent.” In SIDL, an opaque is a 64-bit variable that cannot be touched or modified by the holder. It is normally used to hold pointers that cannot be understood by the current language or in the current context.

out

out is a SIDL reserved word. Each parameter passed though Babel must be declared as in, out, or out. Each of these modes has certain rules and implication associated with it. Out means “pass this (null) variable by reference to the implementation. The implementation is expected to fill the reference with a new variable to be passed back to the client.” See Section 5.2.

package

A package is a container and namespace for conceptually linked classes and interfaces. Generally it is good practice to have one package per SIDL file.

PIC

Position Independent Code is for making dynamically loadable libraries. PIC contains and extra level of indirection to allow the correct methods to be found dynamically at runtime.

preprocessing

Code preprocessing is a step, prior to compilation, where various simple, automatic code modifications are made. For example, int C, #include files are included, and #define macros are textually duplicated throughout the code. In some cases, such as Babel FORTRAN 90, method names are “mangled” to reduce their size under the method name character limit.

private data

Private data is data that is only accessible locally, inside an object. In Babel, all Babel object data is private and cannot be accessed by other SIDL objects.

process

A process is a running program that exists in its own memory space and can therefore run in parallel with other processes.

reference counting

Reference counting is the form of garbage collection used in Babel. Each object keeps a "reference count." When that count reaches zero, the object is destroyed and the memory reclaimed. In some languages teh counting is handled automatically, in some, like C, the developer must explicitly add and subtract from the reference count. (Using the functions addRef and deleteRef.) The internal implementation of deleteRef literally has an if statement that says "If the count is 0, free this memory," so if the reference count of an object goes below one, all references to the object are immediately invalid.

reverse engineering

Reverse Engineering is the practice of inspecting the behavior of an existing program to understand more about how it works. Babel does not support this, or any forms of inspecting or modifying compiled code.

shared library

A shared library is a set of methods that may be used by multiple different programs without recompilation of the library.

short name

Overloaded Babelized methods called from non-object oriented languages, such as C and FORTRAN 77, have 2 method names. The full name consists of the concatenation of the package name, class name, method name and type extension. The short name is missing the type extension. See subsection 5.6.

SIDL

Scientific Interface Definition Language. The language used by Babel to describe how Babel glue code should be generated. See Chapter 5.

single process

A single process program is a program that only uses one process to complete its work. One of the features of Babel is that it is able to facilitate language interoperability in a single process, which saves the extra overhead of interprocess communication.

skeleton

The Babel skeleton code is the opposite of the Babel stub code. The Stub code facilitates the method call from client to IOR, and the skeleton code facilitates the method call from IOR to implementation.

SO

Shared Object. A Unix catch all term for shared and dynamically loadable libraries.

SPMD

Single Program Multiple Data. The term used to describe parallel programs that use multiple processes running the same code working on different data to solve a problem.

state (of an object)

Object state refers to the data that an object holds. For example, if an object holds one integer, that integer holds the objects state. It is assumed that instance methods modify or use an object's state in some way. If a method does not use the object state in any way, it should probably be a static method.

static

A static method is a method that does not depend on an object instance to run. It should have no need of any data of any particular object, it should only depend on the data that is passed into it. As such, unlike instance methods, it does not need to run on an instance of the class it is associated with. In Babelized C, this means the first argument to the function is not an object instance. In Java, this means the function not called on an object, but referenced by the class name.

static linking

Static linking refers to the practice of linking code at compile time, rather than dynamically at runtime. It has a speed advantage over dynamically linked code, but lack flexibility.

string

string is a data type built into SIDL. It stores a set of characters. It has no predefined length.

stub

The Babel stub code is the opposite of the Babel skeleton code. The Stub code facilitates the method call from client to IOR, and the skeleton code facilitates the method call from IOR to implementation.

SWIG

Simplified Wrapper and Interface Generator http://www.swig.org/ SWIG is a language interoperability tool that is not IDL based, but has certain other drawbacks.

tarball

Tarball is a common way to refer to a set of directories and files organized into a single file using the Unix tar command. It is often gzipped.

throws

throws is a SIDL reserved word. It is used the tell SIDL that a method may throw the named SIDL exception, and code should be generated to pass it to the client.

type

A type describes what sort of information a variable stores, and usually how much space that information takes up. Classes and interfaces are user defined types, there are also fundamental types like int and bool.

URL

Uniform Resource Locater. Often thought of as a pointer to a web resource.

user

There are two anticipated user types for Babel, both are kinds of programmers. The person referred to as the "developer" is the person developing a Babelized library. The "user" is the person who writes a program using a Babelized library.

version

version is a reserved word in Babel that is used to declare a version for a given package, or to declare what version of a given package should be used.

virtual

Virtual is the opposite of final. All SIDL methods are virtual by default. A virtual method is a method that may be overridden in subclasses.

VM

Virtual Machine

void

a reserved word in Babel, used to state that a function has no return type.

VPATH

If you want to build software in a separate directory from where the tarball was untarred, this is called a "VPATH build". VPATH builds are useful if you want to build Babel multiple times with various compilers, flags, or you have a shared file system across multiple platforms. It separates the code you generate from things that you were given.

XML

Extensible Markup Language. http://www.w3.org/XML/ A standardized data exchange format.

Bibliography

[1] Babel homepage. `http://www.llnl.gov/CASC/components/babel.html`.

[2] David E. Bernholdt, Wael R. Elwasif, James A. Kohl, and Thomas G. W. Epperly. A component architecture for high-performance computing. In *Proceedings of the Workshop on Performance Optimization via High-Level Languages (POHLL-02)*, New York, NY, June 2002.

[3] CCAFE homepage. `http://www.cca-forum.org/~baallan/ccafe`.

[4] Bradford Cobb, Gary Hook, Christopher Strauss, Ashok Ambati, Anita Govindjee, Wayne Huang, and Vandana Kumar. AIX linking and loading mechanisms. `http://www-1.ibm.com/servers/esdd/pdfs/aix_11.pdf`, May 2001.

[5] Common Component Architecture (CCA) Forum homepage. `http://www.cca-forum.org`.

[6] Tammy Dahlgren, Tom Epperly, and Gary Kumfert. *Babel User's Guide*. CASC, Lawrence Livermore National Laboratory, version 0.8.4 edition, April 2003.

[7] Guy Eddon and Henry Eddon. *Inside Distributed COM*. Microsoft Press, Redmond, WA, 1998.

[8] Eric Eide, Jay Lepreau, and James L. Simister. Flexible and optimized IDL compilation for distributed applications. In *Proceedings of the Fourth Workshop on Languages, Compilers, and Run-time Systems for Scalable Computers*, 1998.

[9] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*, July 1996. Available at `http://java.sun.com`.

[10] Michi Hennig and Steve Vinoski. *Advanced CORBA Programming with C++*. Professional Computing. Addison-Wesley, 1999.

[11] International Organization for Standardization, Geneva. *ISO/IEC 14882 Standard for the C++ Programming Language*, 1998.

[12] Bill Janssen, Mike Spreitzer, Dan Larner, and Chris Jacobi. *ILU Reference Manual*. Xerox Corporation, November 1997. Available at `ftp://ftp.parc.xerox.com/pub/ilu/ilu.html`.

[13] Scott Kohn, Gary Kumfert, Jeff Painter, and Cal Ribbens. Divorcing language dependencies from a scientific software library. In *10th SIAM Conference on Parallel Processing*, Portsmouth, VA, March 2001.

[14] Scott Meyers. *More Effective C++: 35 New Ways to Improve your Programs and Designs*. Professional Computing. Addision-Wesley, 1996.

[15] Scott Meyers. *Effective C++: 50 Specific Ways to Improve your Programs and Designs*. Professional Computing. Addision-Wesley, 2 edition, 1998.

[16] Microsoft Corporation. *Component Object Model Specification (Version 0.9)*, October 1995. See `http://www.microsoft.com/oledev/olecom/title.html`.

[17] Object Management Group. *The Common Object Request Broker: Architecture and Specification*, February 1998. Available at `http://www.omg.org/corba`.

[18] SciDAC: Scientific Discovery through Advanced Computing. `http://www.science.doe.gov/scidac`.

[19] SCIRun homepage. `http://www.sci.utah.edu`.

[20] John Shirley, Wei Hu, and David Magid. *Guide to Writing DCE Applications*. O'Reilly & Associates, Inc., Sebastopol, CA, 1994.

[21] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, 3 edition, 1997.

[22] U. S. Department of Energy (DOE) homepage. `http://www.energy.gov`.

[23] Norm Walsh. *DocBook*. O'Reilly, 2000.

[24] XCAT homepage. `http://www.extreme.indiana.edu/xcat`.