Automating Multilanguage Development Processes for the High-Performance Software Lifecycle

Benjamin A. Allan[1] and Boyana Norris[2]

[1] Sandia National Laboratories, Livermore, CA, USA,
baallan@ca.sandia.gov
[2] Argonne National Laboratory, Argonne, IL, USA,
norris@mcs.anl.gov

Abstract. The component-based software engineering approach based on the Scientific Interface Definition Language(SIDL), Babel, and the Common Component Architecture have been employed successfully in high-performance multilanguage software projects dominated by the development of new code. Large scientific projects that rely on existing object-oriented (non-component) code, however, have additional automation requirements, including: (1) fully automated component wrapper generation from non-component code; (2) single expression of interfaces so that maintainers and users have no development lag or code mismatch between changes to the wrapped code and changes to the wrappers; and (3) extensible source code markup for directing the wrapping process of ambiguous cases. These issues are not currently addressed by the high-performance software development tool community. We describe a combination of extensions to the SIDL syntax, the Babel code generation processes, and the bocca tool to address these issues, providing dramatic reductions in component software development time.

1 Background and Motivating Challenges

1.1 Component-Based Software Engineering for HPC

Components There are many existing component-based application frameworks. CCA requires language interoperability through SIDL for integrating, rather than rewriting, diverse libraries into larger applications.

Currently SIDL and Babel provide the strict separation of implementation language from public functionality. SIDL provides a Java-like object model with support (automatic or manual depending on the bound implementation language) for memory management, static type safety, and exception handling. The result is restricting; each of the supported modern implementation languages (C++, Python, Java, Fortran) loses some features which do not map easily to the others. Particularly, function arguments are restricted to the subset that can be trivially and canonically mapped to SIDL.

The typical SIDL/Babel wrapping process to create a component involves:

1. Defining the function signatures needed in SIDL.
2. Generating an incomplete *implementation* file in the same language as the library being wrapped.
3. Inserting hand-written code to call the library in the implementation file.
4. Inserting hand-written, CCA-required boilerplate implementation code.
5. Compiling the implementation file.
6. Generating sources and compiling them for bindings in the implementation language and other Babel client languages desired.

The entire process involves many files, most of which the user must compile but never edit, which then must be packaged into one or several libraries and eventually installed. Bocca [1, 2] eliminates the effort needed to track all these details except for steps 1 and 3.

1.2 Multilanguage HPC Software Development Process Challenges

We next describe the social and technical barriers to adopting the CCA multi-language approach.

Transient development affects most research code projects. Software developers within the same HPC project frequently have conflicting goals. Most research code projects are developed by a constantly changing group of temporary employees for whom time to first solution and respective publications are paramount. Good software engineering practices and language interoperability are important to the software owner who expects a history of quick results over several years, incrementally built on prior results. These practices do not simplify the work of the transient contributor charged with implementing just one new feature or result, unless a good component decomposition of the software already exists.

Lack of fully automated two-way wrapping tools. A large number of one-way wrapping tools exist, with varying degrees of automation support. These tools usually target the generation of bindings for some scripting language from C and C++, or Fortran. None addresses the full needs of scientific code wrapping [3].

Incorporating CCA-compliant component implementations into code generated by the Babel middleware compiler is a nontrivial time-consuming manual process, even when bocca is used to manage the build details. The time and complexity costs of producing SIDL and component wrappers for an evolving code base must drop by orders of magnitude to make them more attractive.

The source code does not (in most cases) contain enough information to uniquely determine how it should be mapped to SIDL or other language binding generators. This extra information must be associated with the source code in a manner which is convenient to the developers and maintainers of the code and easy to understand and extend for new developers of the code.

2 An Example Use Case

We describe a layered set of tools to address the challenges presented in Section 1. The result is a simplified and accelerated workflow for wrapping libraries. We assume the source language is C++ for this example (the process is similar for all languages supported by Babel).

1. Annotate the C++ source with structured comments to resolve ambiguous SIDL wrapping decisions.
2. Invoke a C++-to-SIDL interface generator with the input C++ sources as arguments to generate a set of wrapper SIDL files annotated for automatic implementation generation.
3. Invoke the normal bocca build processes as needed.
4. Review the results and iterate from step 1 as needed.

Step 1 requires the user to provide answers to the questions: (i) What objects and methods are wrapped or ignored? (ii) How are methods grouped into SIDL interfaces and classes? (iii) How are native arrays mapped to SIDL? (iv) In what SIDL packages are the wrapping classes and interfaces placed?

Step 2 requires SIDL interface generator tools (SIGs) which can process the original project source code and automatically create complete, ready for compilation SIDL files. Each programming language community has its own formatting conventions for associating structured comments with code entities that can be processed to yield documentation, typically in HTML format. Similarly, a new structured comments language can be used to direct generation of SIDL.

Enhancements to Babel beyond the features of the current version 1.2 are necessary as a prerequisite to step 3, as we will describe in Section 3.1.

3 Analysis and Solution of the SIG Problem

Previous attempts at solving the interface generator problem have been monolithic, monolingual, and reliant on tools exotic to the typical scientific programmer. Typical of the approaches [4] is one that works by parsing source in a single language, resolving mapping ambiguities by applying XSLT stylesheet-based transformations, and using Babel as it currently exists. Both SIDL and complete wrapper implementation code are generated by the tool.

There are a number of important drawbacks to prior approaches. The generated wrappers are not capable of retaining, in the binding of SIDL to the same source language, the feature richness of that language . Minor changes in the SIDL/Babel tool invalidate the implementation code produced. Working in new languages, such as XSLT, requires a significant investment of time. Finally, the SIDL and Babel bindings are always completely distinct from the core project source, and thus they are easily neglected as development of the core code features moves forward.

3.1 A New Approach

We propose to enable the automated generation of the *complete implementation* of SIDL classes, which requires extensions to the SIDL language. We will accomplish this by writing a SIG for each source language supported by Babel, which will work as described in the use case in Section 2. We need the following three additions to SIDL and Babel to enable development of maintainable SIGs and to enable the unification of the Babel-generated interfaces and the feature-rich native interfaces of the wrapped code.

Typed opaque objects for SIDL. SIDL supports passing of opaque items (64 raw bits) from one language to the same language through intermediate languages. Passing array or object references as opaque causes all type information to be lost, forcing the called code to apply potentially incorrect type casting to recover the native type from the raw bits. We can easily add native type information to SIDL, preserving it as the opaque crosses language boundaries.

Eliminating the native mismatch overheads of SIDL/Babel. Methods in HPC languages may require function argument or return types that cannot be easily coerced into any other language. SIDL can be extended to allow such methods to appear only in the binding of the language in which these types are defined. We thus allow the full power of SIDL object-orientation to be used as the basis for a library or wrapping in those languages which do not have good support for object-oriented programming. This allows the SIDL wrapping of a code to become more than just an overhead to support language interoperability, adding robust software engineering practices to legacy codes without having to rewrite them.

SIDL markup. We must be able to give hints in SIDL so that Babel can generate the complete wrapper implementation without human intervention. SIDL has a primitive annotation capability. We can extend the Babel implementation code generators to use expanded annotations syntax to derive complete, ready-to-compile implementation code.

References

1. Elwasif, W., Norris, B., Allan, B., Armstrong, R.: Bocca: A development environment for HPC components. In: Proceedings of HPC-GECO/CompFrame Workshop, Montreal, Canada. (2007)
2. CCA Tutorial Working Group: CCA Tutorials. http://www.cca-forum.org/tutorials/ (2008)
3. Dahlgren, T., Epperly, T., Kumfert, G., Leek, J.: Babel User's Guide. CASC, Lawrence Livermore National Laboratory, Livermore, CA. babel-0.9.4 edn. (2004)
4. Rasmussen, C.E., Sottile, M.J., Shende, S., Malony, A.D.: Bridging the language gap in scientific computing: The Chasm approach. Concurrency and Computation: Practice and Experience **17**(2–4) (2006) 151–162