Toward The Automated Generation of Components from Existing Source Code

Daniel Quinlan
Qing Yi
Gary Kumfert
Thomas Epperly
Tamara Dahlgren
Center for Applied Scientific Computing
Lawrence Livermore National Laboratory
P. O. Box 808, Livermore CA, 94551 USA
{dquinlan,yi4,kumfert,tepperly,dahlgren}@llnl.gov

Markus Schordan
Institute of Computer Languages
Vienna University of Technology
Argentinierstrasse 8/4/13, A-1040 Vienna, Austria
markus@complang.tuwien.ac.at

Brian White
Computer Systems Laboratory
Cornell University, Frank H. T. Rhodes Hall
Ithaca, NY 14853, USA
bwhite@csl.cornell.edu

Abstract

A major challenge to achieving widespread use of software component technology in scientific computing is an effective migration strategy for existing, or legacy, source code. This paper describes initial work and challenges in automating the identification and generation of components using the ROSE compiler infrastructure and the Babel language interoperability tool. Babel enables calling interfaces expressed in the Scientific Interface Definition Language (SIDL) to be implemented in, and called from, an arbitrary combination of supported languages. ROSE is used to build specialized source-to-source translators that (1) extract a SIDL interface specification from information implicit in existing C++ source code and (2) transform Babel's output to include dispatches to the legacy code.

1 Introduction

Contemporary multi-disciplinary, multi-scale, multi-physics simulations are increasingly becoming large, composite applications consisting of new and existing components implemented in different programming languages by disparate teams. These factors present several challenges to developers of such systems that, if dealt with manually, are time-consuming and error prone. This paper addresses an automation technology for the extraction and implementation of components within the context of a component architecture tailored for scientific computing.

Component technology is industry's answer to at least two of the three major concerns plaguing large-scale componentization efforts; namely, interoperability of software written in different languages, interoperability of software running on different platforms, and maintenance and evolution of large composite systems with multiple third party dependencies. Component architectures from industry include CORBA [3], Microsoft COM [14], and Sun's Enterprise Java Beans (EJB) [5]. These architectures establish the framework in which compliant components interact. For instance, EJB assumes all components are implemented in Java (or JNI), thereby leaving the language interoperability issue to component developers to address. Unlike commercial applications, large scale numerical simulations have additional constraints unique to the scientific computing domain such as high performance, a wide variety of often one-of-a-kind computing platforms, and a need to migrate a substantial body of code to a new programming paradigm.

The Common Component Architecture (CCA) Forum [1] is working to deliver component technology suitable for large scale numerical simulations. Babel provides the Scientific Interface Definition Language (SIDL) and associated language interoperability tools that undergirds CCA-compliant frameworks. Current best practice for migrating legacy source code to CCA components does not require modifying existing code but may involve substantially

rethinking the interfaces and writing additional code by hand that bridges Babel's language bindings to the legacy code.

In this paper we address the automated generation of the bridging code between C/C++ libraries and their CCA-compliant component wrappers. The work employs the ROSE compiler tools to build specialized translators that use a library's header files to generate the SIDL file and bridging code required by the CCA framework. Such an automated approach is critical to converting legacy codes to components in sufficient numbers to achieve the economy of scale that makes component technology so effective in other domains. Currently our work is limited to C and C++ libraries. Our ongoing work includes adding a FORTRAN90 frontend to the ROSE infrastructure, which will enable us to apply similar techniques to convert FORTRAN libraries to components in the future.

The major technical challenges of automatically mapping libraries to SIDL code come from two aspects. First, to allow all-to-all interoperability among its supported languages, Babel defines SIDL with a narrow intermediate type-system and inheritance model. The ROSE translators thus must extract the necessary (often implicit) information embedded in the library and map the C/C++ type system into one expressible in a proper SIDL file. Second, programming in the CCA component model has a more event-driven and less imperative feel than traditional programming due to its focus on services. For instance, a CCA component rarely explicitly creates all the lower-level components it depends on to function. Instead, it typically registers what capabilities it provides and which capabilities it depends on to the CCA framework in response to an event, which usually takes the form of a creation request, but can also be a connection or disconnection request. Then the CCA component typically is inactive until one of its provided capabilities is invoked.

Our present work has addressed the first challenge by translating C++ types into equivalent SIDL types when possible, and conservatively using the *opaque* type in SIDL (indicating no information is known about the type) if no suitable translation is available. The work to address the second challenge is still ongoing. Specifically, our future work will include techniques to automatically cluster global functions and classes into different components (our current implementation simply places all global types in a library into a single component). Further, we will automate the generation of CCA components, which are independent units of composition that implement the $gov.cca.Component$ interface. More details are provided in Section 3.4.

Although C and C++ are only two of the modern languages supported by Babel and CCA, the their type system represents one of the most complex and comprehensive type systems in existing statically-typed languages. For example, generating SIDL specifications for a C/C++ library requires the translation of overloaded functions and operators, classes with multiple inheritance, C++ templates, function pointers, and variable number of arguments, many of which can be ignored when translating smaller languages such as Java and FORTRAN, which do not have multiple inheritance or C++ templates. We thus expect that many techniques we develop for translating C/C++ libraries to CCA components will similarly apply to Java and FORTRAN as well. Further, much of the design principles we developed are language independent, and can apply in general to all modern programming languages.

2 Infrastructure

Our component generation infrastructure includes both ROSE [21, 24] and Babel [16]. ROSE is a compiler infrastructure that offers mechanisms for analyzing C++ source code and for building source-to-source translators, which in this paper are used to process library code and generate component implementations. Babel is an Interface Definition Language (IDL)-based language interoperability tool akin to CORBA but tailored for the scientific computing community. In the following two sections we describe ROSE mechanisms in simplifying the development of translators and Babel capabilities in aiding the generation of components.

2.1 ROSE

The ROSE infrastructure allows building source-to-source translators by offering a front-end for parsing C++ code and generating an Abstract Syntax Tree (AST), a mid-end for restructuring the AST representation of the source code, and a back-end to unparse C++ source code from the AST.

We use the Edison Design Group (EDG) C++ front-end [2] to parse C++ programs. After invoking the EDG parser on an input C++ program, we then translate the C-style EDG internal representation of the program into an object-oriented abstract syntax tree (AST), Sage III, which we have developed as a revision of the Sage II [13] intermediate representation. Current work includes collaboration with Rice to add F90 support to ROSE through use of the Open64 compiler infrastructure.

The mid-end supports restructuring of the Sage III AST. The programmer can add code to the AST by specifying a source string using C++ syntax, or by manually constructing subtrees of the AST. A program transformation consists of any required program analysis and a series of AST restructuring operations each of which specifies a location in the AST where a code fragment should be inserted, deleted, or replaced.

The back-end unparses the AST and generates C++ source code. Header files can either be unparsed where they are included in source files, or `#include` directives can be generated for the header files. This feature is important when transforming user-defined data types, for example, when adding compiler-generated methods.

2.2 Babel

Compared with other IDL technologies, Babel/SIDL has several features critical for scientific computing such as intrinsic support for dynamically allocated, arbitrarily strided multidimensional arrays and complex numbers. It also supports overloading method names. Whereas CORBA emphasized remote method invocation, Babel emphasizes fast, in-process language interoperability [12]. Babel also has extensive FORTRAN 77/90/95 support, even allowing Babel arrays to be manipulated as native FORTRAN 90 arrays [16]. The Babel team is currently developing remote method invocation (RMI) capabilities.

Babel includes two parts: the code generator and the runtime library. The code generator parses SIDL and generates client and/or server bindings in C, C++, FORTRAN 77, FORTRAN 90, Python, and Java. The runtime library contains base classes of the object model which are themselves defined in SIDL and additionally, bits and pieces needed to enhance portability and support interoperability.

IDLs are fundamentally different than typical programming languages. IDLs define types without providing code to implement them. Often, IDL resembles stripped down C++ header files. In SIDL (scientific IDL), users can define new types (classes and interfaces), name operations on their types, specify arguments of their operations, and designate different scopes to avoid symbol name collision. Unlike C++, each argument is explicitly annotated as `in`, `out`, or `inout` to indicate whether data is being passed as an input, produced as an output, or used as input and output for the operation. SIDL has only declarative statements and no mechanism for defining states or algorithms.

Babel's main purpose is to enable scientific library designers to make their code language independent and thus reach a broader audience [18]. *Babelizing* an existing library typically involves writing a SIDL interface specification, running the Babel code generator to generate implementation bindings (called *Impls* for short) in the same language as the library, and hand coding the empty Impls to dispatch to the existing software. Though Babelizing code requires manual programming, customers find it easier than generating a single language wrapper by hand. Some even welcome the opportunity to craft a modern object-oriented interface over their legacy procedural code.

More details about Babel-generated Impls are needed for discussion in later sections. Recall that implementation details of software are intentionally inexpressible in SIDL. Therefore, when Babel first generates Impls, the bodies of the methods, member functions, subroutines, or procedures (depending on what programming language Babel's generating Impls for) are empty. The library developer needs to fill in the Impls with an actual implementation or code to dispatch their existing code. Since Impls are generated code that contain hand-edited fragments, these fragments are located in *splicer blocks*. Contents of splicer blocks are preserved across multiple runs of Babel as SIDL specifications evolve. Automatically generating contents for the splicer blocks is one of the challenges for this joint work.

3 Automated Code Generation, Transformations, and Analysis

The processing steps for automatically translating a library into components are summarized in Figure 1. Essentially, the steps are

1. SIDL and C++ implementation information extraction.
   This step involves using ROSE Translator T1 to process the *library example* to generate SIDL code and *Non-SIDL C++ Information* for ROSE Translator T2.

2. Stub and Impl generation.
   Babel is called using the SIDL file from step 1 to generate all stubs for clients in all supported languages as well as the corresponding C++ *Impl* files with empty *splicer blocks*.

3. Implementation bridge generation.
   The ROSE Translator T2 is called using *Non-SIDL C++ Information* and the C++ *Impl* files from step 2 to insert dispatching code into the initially empty *splicer blocks*.

To simplify the process each of these steps can be fully contained within a single program.

The input to the ROSE Translator T1 in Figure 1 includes two objects: a target library and a simple *library example* program. The *library example* program is constructed by hand and must include all the header files required to define the target library's implementation. Only the library's header files must be seen, although more sophisticated analysis is possible by processing the entire library as described in Section 3.3. The *library example* program can be as simple as a one line file that includes a single header file. For example, a file containing the line `#include <A++.h>` is a sufficient *library example* program for processing the A++ library.

Given the target library and an example program, the ROSE Translator T1 generates two outputs: SIDL interface files and Non-SIDL C++ information. The SIDL interface

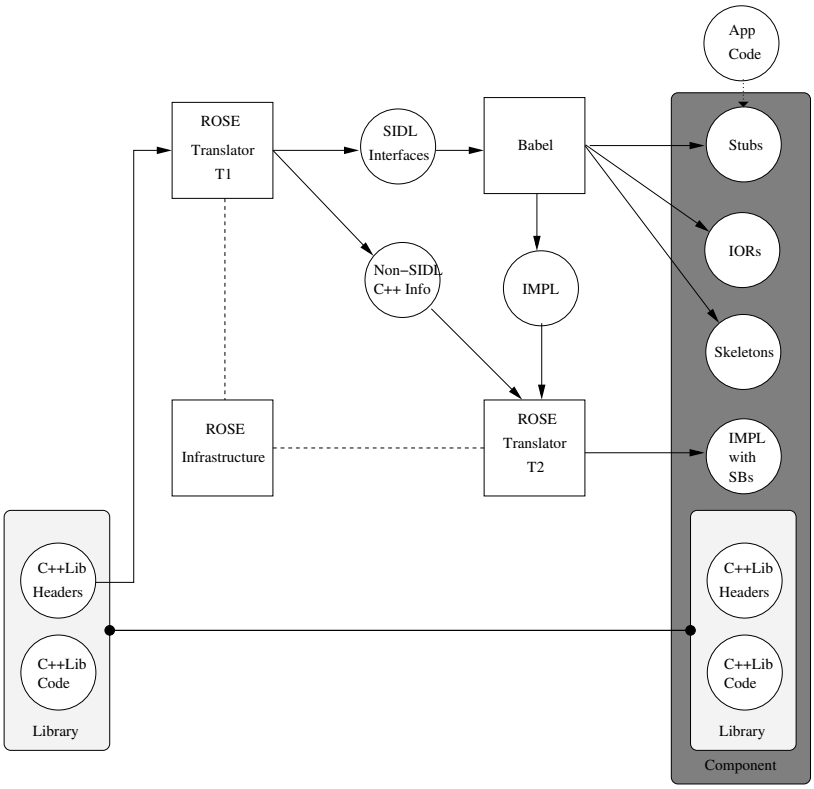

Figure 1. Component Generation Process with ROSE-Babel infrastructure. Circles represent (generated) files and arrows show the data-flow of these files. The dashed lines show that the translators T1 and T2 use the ROSE infrastructure. The library is not modified and becomes part of the generated component.

files are later used as inputs to Babel, which then generates components specifications and Impl files with empty splicer blocks. In greater detail, the ROSE Translator T1 performs the following substeps of step 1:

1.a. Constructing AST.
The AST at this point represents all library declarations. Only classes, structs, functions and member functions are of particular interest, but the AST also contains all comments, pragmas, variables, typedefs, etc.

1.b. Collecting information about classes and functions

Class definitions. Builds the list of library classes, each of which is translated into SIDL classes.

Member functions. This step builds a list of all member functions, each of which is put into its associated SIDL class previously constructed (preceding step).

Non-member functions. Builds a list of all non-member function, each of which is put into a SIDL class called "Global".

1.c. Generating *Non-SIDL C++ Information*
A file containing the list of `#include` directives is generated. The file is specified on the command-line as library specific data to be read by the ROSE Translator T2.

The above steps preserve the original structure of the library. The generated SIDL code does not re-organize the library other than presenting a list of global functions, classes, and member functions. To add more structures to the generated code, the following information can be used: directory and filename information, function name prefix information, pragmas in the library headers to specify mapping of functions to SIDL interface classes, and an alternative external annotation mechanism for specifying the mapping of functions to SIDL interface classes. These heuristics are part of our ongoing research.

The ROSE Translator T2 requires two inputs: the Impl files from Babel and the non-SIDL C++ information from ROSE Translator T1. Since the SIDL language doesn't

permit the specification of `#include` directives, the non-SIDL information includes the list of `#include` directives present in the library example program processed by T1. These directives are required for declaring the library interface in the new Impl files generated by The ROSE Translator T2, which performs the following substeps of step 3:

3.a. Inserting the list of `#include` directives into the Impl file's appropriate splicer block.

3.b. Inserting code to map each input parameter of each Impl function to the appropriate parameter of the library's function call.

3.c. Inserting library function calls into the appropriate Impl function's splicer block.

Through library annotations or analysis, we can exploit SIDL specific features that are not present in C++ (e.g., specification of side-effects to function parameters via `in`, `out`, and `inout`), see Section 3.3. A third ROSE Translator could automate analysis of the library, using side-effect analysis to verify the correctness of parameter annotations or to use an in or out specification in lieu of the default inout. This narrower specification of parameters enables subsequent compiler optimization.

3.1 Generation of SIDL

Because the set of C++ features is much larger than those present in SIDL, mapping from C++ to SIDL requires some complex translation. Much information could be lost in this process, although it could conceivably be saved in the *Non-SIDL C++ Information* and used within the marshaling of function parameters between the Impl functions (generated by Babel) and the target library's function calls. The following issues have been considered in the existing translation of C++ code to SIDL:

C++ overloaded functions. Additional information is required within SIDL to support overloaded operators.

C++ overloaded operators. All overloaded operators are given unique names within the generation of SIDL. These names are mapped back to the respective overloaded operators within the transformation of the Impl files.

C++ function pointers. These are handled using a SIDL opaque. Some function pointers will be replaced by a SIDL interface.

Multiple inheritance. SIDL supports only single inheritance for classes and multiple inheritance of interfaces (similar to Java and Objective C). Through a level of indirection (supported in the interface parameter marshaling), multiple inheritance models in C++ can be reduced to single inheritance models appropriate for representation in SIDL.

C++ templates. There is no C++-like templating mechanism available as part of the SIDL interface. However, each template instantiated internally in the target library is represented as a *template-instantiation class* within ROSE, which can be translated to any non-template class within SIDL. This permits the use of templates within C++ libraries so long as they are instantiated over a closed set of parameterized types. This detail requires that the *library example* program triggers instantiate (uses) all templates.

SIDL support for arrays. SIDL supports arrays of specific types, but functions passing pointers to data and an integer describing its length can skip the use of the SIDL array abstractions. This avoids a translation ambiguity.

Variable arguments. C++ methods with variable numbers of arguments, using the ellipsis ... in their declaration must be converted to a method with a SIDL array containing a generic argument base class.

SIDL's opaque type is necessary for low level routines with application programming interfaces (APIs) that require address pointers. For example, an opaque would have to be used for the ANSI C routine `signal` which requires a function pointer as an argument because our tool cannot change the underlying implementation to use a functor approach. A routine such as ANSI C's `malloc`, would need to use opaque as a return value. Certain device drivers might also require particular addresses as arguments.

3.2 Transformation of Impl files

Babel generates both stubs for other languages to call and Impl files to invoke the implementation of the library functions. Instead of generating new Impl files, which is handled by Babel, we transform the Impl files generated by Babel by inserting calls to the associated library functions and marshaling all parameters.

3.3 Library Analysis

An optional step is to process the target library implementation and analyze each function in the library to determine the side-effects upon their parameters. The side-effect analysis has been implemented as a result of collaborations with Cornell and will permit a verification of (`in`, `out`, and

`inout`) annotations and or the generation of such annotations. The correct classification of interface function parameters is mostly a performance issue. The current conservative default classification is to classify all function parameters as `inout`. An additional processing step using ROSE could automate much of the classification of function parameters. In some cases, lack of sufficient program analysis, in particular pointer alias analysis, may require the process be semi-automatic rather than fully automatic. For example, side-effect analysis could signal that a potential alias between a locally-modified variable v and a parameter p prevents declaring p as an in parameter rather than inout. This ambiguity could be resolved by an annotation, specified through ROSE's annotation language, which declares that v does not alias p.

3.4 Future Extension: CCA Componentization

The CCA component framework has two requirements: *components* and *ports*. A component is an independent unit of composition that must implement the `gov.cca.Component` interface. Ports are capabilities, or services, of a component that must be specified in SIDL as extensions of `gov.cca.Port`.

Our translator will need to generate a set of classes implementing `gov.cca.Component`. This interface has one method, `setServices`, that must notify the framework about which ports the component can provide (known as *provides ports*) and which ports the component requires (known as *uses ports*). Identifying the mapping from the original set of C++ classes to SIDL classes implementing `gov.cca.Component` is a major challenge. In some cases, it might be best to treat each concrete C++ class as a component, and in other cases, the whole multi-class library should be considered a single component.

Our translator will also need to generate a set of ports based on the C++ classes in the original API. The first step will be to create a port for each C++ class involved. Determining better methods for choosing which C++ classes should be included in each type of port will need to be explored. *Provides ports* are basically services provided to the library's user; hence, they can be gleaned from the interfaces of a class. However, a *uses port* indicates services provided by the component's client that are needed by the component. Designating something as a *uses port* of a component means that the component needs exactly one instance of the port corresponding to the initial C++ class, which is very challenging to determine from the C++ header files. The underlying code may be able to handle zero through many instances.

4 Related Work

Our aim is to automate the generation of components from legacy scientific applications. This process includes two phases: extracting component interfaces and producing implementations that bind the interface specifications with the original software.

A number of research efforts have aimed at extracting components from existing software. Specifically, many clustering techniques [19, 20, 15] have been developed to analyze the function calls within a library system and to identify reusable components within the library. We currently focus on making all individual classes in a library reusable. Our work can be combined with the clustering techniques to provide better component interfaces for libraries. Beck and Eichmann [10] have also explored the extraction of interfaces from source code. They have focused on reducing the interfaces (and code) to only those methods that are actually needed by a user. Their solution is language-specific; whereas we focus on extracting language-neutral specifications and automating the library bindings.

To automate the second phase of generating components, Babel provides the translation from the interface specification to implementation stubs. Our work then generates dispatching code that fills in these stubs. Prior efforts have developed several systems that support automatic bridging of pairs of different languages. For example, SWIG [6, 9], a wrapper and interface generator, supports automatic bindings between C/C++ and common scripting languages such as Tcl, Python and Perl. In contrast, we leverage Babel's intermediate representation, so component developers do not have to be concerned with providing a point-to-point mapping for their users. Furthermore, Babel's RPC-like mechanism will enable future remote access to the libraries wrapped by our infrastructure. Chasm [22, 23], another point-to-point adapter generator, employs static compiler analysis to automatically connect C++ applications to FORTRAN 90 libraries. Our work, on the other hand, automates the connection of C++ libraries with applications written in a variety of scientific computing programming languages. By leveraging SIDL, which has been adopted as the specification language for scientific components by the Common Component Architecture Forum [1, 8], we enable the automatic generation of CCA-compliant components from existing libraries.

Similar to our work, Rational Rose [4], a commercial general-purpose graphical modeling tool, supports round-trip engineering from user applications to both CORBA and COM specifications [11, 17] using the Unified Modeling Language (UML) [7] as the intermediate representation. Our work, on the other hand, does not require the translation from yet another intermediate language. By virtue of

using Babel/SIDL, we also have an intermediate layer that is tailored for scientific computing.

5 Conclusions and Future Work

This paper presents work on the automated generation of components. The work is incomplete presently but shows both the automation of the SIDL description and the connection of the generated code (from SIDL) back to the library. Both pieces are essential to automate the connection to an arbitrary library. To enable automatic generation of CCA components, additional analysis and transformations of the resulting SIDL objects is necessary to properly define and designate ports and implement the CCA required `setServices()` method.

All known "automatic" language wrapping tools require some degree of hints, pragmas, structured comments, or the like just to enable a one way connection from the calling language to the existing code. To "automatically" Babelize an existing C++ code is even more challenging, but offers more capabilities if successful. Taking the entire body of Babelized code and packaging it up as a component actually involves analysis of how the code is used and creation of new functions in the interface. The ultimate goal is to break any large existing code up into useful constituent CCA components.

Although the current ROSE infrastructure is limited to C and C++, the essential motivations are language independent. Analogous reverse mappings of FORTRAN to SIDL would require FORTRAN-specific analysis and techniques.

Specific details to C and C++ are addressed separately. It is conceivable that all C++ language feature could be mapped to SIDL without extension, but with some library specific translation support. Still, such tools could be made easy to build in the future, perhaps even automatically generated.

6 Acknowledgments

This work was performed under the auspices of the U. S. Department of Energy by the University of California, Lawrence Livermore National Laboratory under Contract No. W-7405-Eng-48. UCRL-CONF-208403

References

[1] Common Component Architecture forum. http://www.cca-forum.org.

[2] Edison Design Group. http://www.edg.com.

[3] Object Management Group's CORBA Component Model. Available online from the OMG http://www.omg.org/.

[4] Rational Rose UNIX. http://www-3.ibm.com/software/awdtools/developer/rose/unix/.

[5] Sun Microsystems' Enterprise JavaBeans Downloads and Specifications. Available online from Sun http://java.sun.com/products/ejb/docs.html.

[6] SWIG. http://www.swig.org.

[7] Unified Modeling Language. http://www.uml.org.

[8] R. Armstrong, D. Gannon, A. Geist, K. Keahey, S. Kohn, L. McInnes, S. Parker, and B. Smolinski. Toward a common component architecture for high-performance scientific computing. In *Proceedings of the 8th High Performance Distributed Computing*, 1999.

[9] D. M. Beazley. SWIG: An easy to use tool for integrating scripting languages with C and C++. In *Proceedings of the 4th Annual Tcl/Tk Workshop*, Monterey, CA, July 1996.

[10] J. Beck and D. Eichmann. Program and interface slicing for reverse engineering. In *Proceedings of the 15th International Conference on Software Engineering (ICSE '93)*, pages 509–518, May 1993. Baltimore, Maryland.

[11] N. Bereny. Rose 101: Component Modeling with Rose 98. *Rose Architect*, January 1999.

[12] D. E. Bernholdt, W. R. Elwasif, J. A. Kohl, and T. G. W. Epperly. A component architecture for high-performance computing. In *Proceedings of the Workshop on Performance Optimization via High-Level Languages and Libraries*, New York, NY, June 2002.

[13] F. Bodin, P. Beckman, D. Gannon, J. Gotwals, S. Narayana, S. Srinivas, and B. Winnicka. Sage++: An object-oriented toolkit and class library for building fortran and C++ restructuring tools. In *Proceedings. OONSKI '94*, Oregon, 1994.

[14] D. Box. *Essential COM*. Addison-Wesley Professional, 1997.

[15] Y. Chiricota, F. Jourdan, and G. Melancon. Software components capture using graph clustering. In *Proceedings of the 11th International Workshop on Program Comprehension (IWPC 2003)*, pages 217–225, May 2003.

[16] T. Dahlgren, T. Epperly, and G. Kumfert. *Babel User's Guide*. Lawrence Livermore National Laboratory, Livermore, CA, 0.8.8 edition, 2003.

[17] J. Hammond. CORBA and Rational Rose – An Insider's View. *Rose Architect*, April 1999.

[18] S. Kohn, G. Kumfert, J. Painter, and C. Ribbens. Divorcing language dependencies from a scientific software library. In *10th SIAM Conference on Parallel Processing*, Portsmouth, VA, March 2001.

[19] R. Koschke and T. Eisenbarth. A framework for experimental evaluation of clustering techniques. In *Proceedings of the 8th International Workshop on Program Comprehension (IWPC 2000)*, pages 201–210, June 2000.

[20] B. S. Mitchell and S. Mancoridis. Modeling the search landscape of metaheuristic software clustering algorithms. In *Proceedings of the 7th Annual Genetic and Evolutionary Computing Conference (GECCO '03)*, pages 2499–2510, July 2003.

[21] D. Quinlan, M. Schordan, B. Miller, and M. Kowarschik. Parallel object-oriented framework optimization. *Concurrency and Computation: Practice and Experience*, 2003.

[22] C. E. Rasmussen, K. A. Lindlan, B. Mohr, and J. Striegnitz. CHASM: Static analysis and automatic code generation for improved Fortran 90 and C++ interoperability. In *Proceedings of the Los Alamos Computer Science Symposium 2001 (LACSI'01)*, October 2001. Santa Fe, New Mexico.

[23] C. E. Rasmussen, M. J. Sottile, S. S. Shende, and A. D. Malony. Bridging the language gap in scientific computing: the Chasm approach. (in submission).

[24] M. Schordan and D. Quinlan. A source-to-source architecture for user-defined optimizations. In *JMLC'03: Joint Modular Languages Conference*, volume 2789 of *Lecture Notes in Computer Science*, pages 214–223. Springer Verlag, Aug. 2003.