Automating Interoperabilty For Heterogeneous Software Components

Alan Kaplan
Clemson University, Department of Computer Science
Box 341906, Clemson, SC 29634-1906 USA
kaplan@cs.clemson.edu,

Bradley Schmerl
Clemson University, Department of Computer Science
Box 341906, Clemson, SC 29634-1906 USA
schmerl@cs.clemson.edu,

Jack C. Wileden
University of Massachusetts, Department of Computer Science
Box 34610, Amherst, MA 01003-4610 USA
wileden@cs.umass.edu,

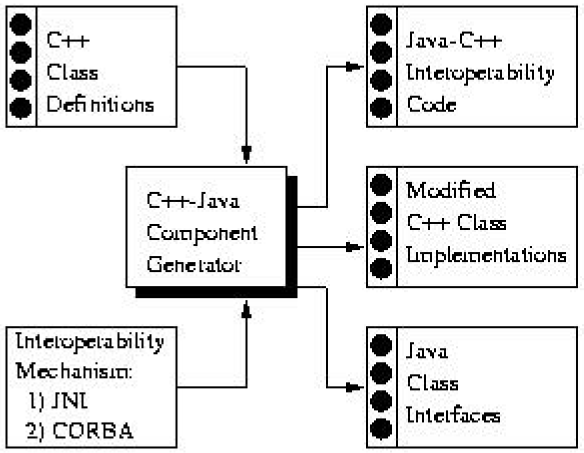
This position paper addresses Section 4, Research Issues and Directions, in the *Proposed Outline for Handbook of CBSE*. Specifically, this paper discusses research issues and directions related to the problem of assembling and integrating software components that have been constructed in different programming languages. This problem is frequently termed the *interoperability problem*. We claim that the interoperability problem is a fundamental concern in the area of component-based software engineering.

Interest in interoperation among software components developed using multiple programming languages is growing rapidly in the software engineering and programming languages community. Reusable software libraries and integration with legacy systems are two common interoperability problems faced by software developers. The expected growth and acceptance of the Internet along with the advent of new programming languages strongly suggest that interoperability will become an even greater issue in the coming years. Although various software engineering tools and programming language constructs supporting interoperation have been proposed and used in the past, these approaches do not meet the demands imposed by today's rapidly evolving heterogeneous computing environment. They are generally difficult to use and prone to error, often forcing developers to waste valuable time dealing with the complexities of a particular interoperability mechanism.

Various software engineering tools and programming language constructs supporting interoperation have been proposed and used in the past. Examples of language-based approaches include the C++ *extern* construct [1] and the Java Native Interface (JNI) [2], each of which support interoperation with the C programming language. In recent years, a myriad of alternative, sometimes competing, interoperability mechanisms (frequently referred to as *middleware* or *componentware*), has been developed. Instead of using specific language constructs, these mechanisms generally rely on a combination of specification languages or intermediaries, code generators and/or highly specialized design styles to achieve interoperability. Examples here include OMG’s CORBA [3], Xerox Parc’s ILU [4], Microsoft’s OLE/DCOM [5], ODMG’s ODL [6] and Sun’s RMI[7].

Despite, or perhaps due to, their number and variety, contemporary interoperability mechanisms are difficult to use and applications that require interoperability mechanisms are difficult to engineer and maintain. First, the choice of a specific interoperability mechanism often becomes inextricably intertwined with the application. This not only hinders development, but makes it extremely difficult, if not infeasible, to change the interoperability mechanism at a later point during the application’s lifetime. Second, applications that require access to pre-existing components often force programmers to re-engineer these components and/or the application itself, thus further reducing the flexibility and robustness of an application. Third, contemporary approaches to interoperability demand far too much programmer involvement in low level details to be appealing to most software developers. Although some of these approaches provide a modicum of automated support, in general they are not well-integrated and require manual intervention, thus making them tedious to use and prone to error. As a result, they force software developers to waste valuable time dealing with the complexities of a particular interoperability mechanism, instead of focusing on the problem domain.

We are currently working on several projects that attempt to address these issues [8,9,10]. The details of these projects are beyond the scope of this position paper. However, they each share the overall goal of trying to hide the underlying interoperability mechanism for heterogeneous software components. For example, we are currently developing and experimenting with a tool that automates interoperability between Java and C++ components. Specifically, the tool allows engineers to create Java class interfaces to existing C++ class libraries using either the Java Native Interface or CORBA. The figure below provides a conceptual architecture of this tool:


C++ Class Definitions
C++-Java Component Generator
Interoperability Mechanism: 1) JNI 2) CORBA
Java-C++ Interoperability Code
Modified C++ Class Implementations
Java Class Interfaces

The purpose of this tool is to provide Java interfaces to existing C++ classes. The tool takes as input the C++ class interface and implementation definitions. The user of the tool also indicates whether JNI or CORBA should be used as the underlying interoperability mechanism. The tool analyzes the C++ class interface and produces a corresponding Java class. Specifically, all public C++ methods are provided in its Java counterpart. Note that the tool does not simply translate from C++ to Java. For example, instance variables defined in the C++ class are not created in the generated Java class. Similarly, private method members are not created in the generated Java class. The tool essentially produces a Java *proxy* for the C++ class. The tool also automatically generates the required Java-C++ interoperability code (as dictated by JNI or CORBA). Finally, the tool may modify the implementation of a C++ class; however, the C++ class interface remains unchanged. Currently, our tool is only in a prototype stage. We are in the process of refining its capabilities and experimenting with techniques to make it more robust. The tool presently only allows for invoking C++ from Java. We intend to support the invoking Java from C++ in the near future. We are also discovering interesting phenomena that arise when trying to provide transparent interoperation between two seemingly similar object-oriented languages. For example, C++ allows:

- pointers to pointers to pointers and so on
- multiple inheritance
- virtual and non-virtual methods

At this stage, it is not clear how to provide corresponding constructs in Java. Our approach is unique compared to existing approaches in that:

- It does not require the use of separate languages and/or type systems (such as CORBA's IDL) to achieve component interoperability. Our tool, working much like a compiler or translator, analyzes the component interfaces and generates the necessary code allowing heterogeneous components to interoperate.
- It allows the possibility of employing different interoperability mechanisms.This allows programmers to develop components without having to commit to a particular interoperability mechanism. For example, an application may choose to integrate components into a single address space (e.g., using JNI) or combine components in a distributed environment (e.g., using CORBA).
- Components provide the same interface independent of an underlying interoperability mechanism. In other words, the decision to interoperate has minimal impact on the component's interface. While contemporary approaches pollute the component code space with interoperability-specific code, our tools ensures that a component's interface remains interoperability code-free.

In summary, it is our position that interoperability is a fundamental concern in component-based software engineering. Although advances in programming language technology have yielded numerous improvements in the construction of components, it is clear that no single programming language will ever dominate software engineering practices. Therefore, new tools and techniques are needed to help software engineers interoperate among heterogeneous components. In this position paper, we have briefly outlined several related research issues and directions. We have also outlined some of our own work that addresses this area.

REFERENCES

1. Stanley B. Lippman. *C++ Primer*, chapter 4, pages 213-214. Addison-Wesley, second edition, 1993.

2. Sun Microsystems Inc. Java native interface, May 1997.

http://java.sun.com/products/jdk/1.1/docs/guide/jni/spec/jniTOC.doc.html

3. Object Management Group. *The Common Object Request Broker: Architecture and Specification*, August 1997. Revision 2.1.

4. Bill Janssen and Mike Spreitzer. ILU: Inter-language unification via object modules. In *Workshop on Multi-Language Object Models*, Portland, OR, August 1994 (in conjunction with OOPSLA’94).

5. Kraig Brockschmidt. *Inside OLE, 2nd Edition*. Microsoft Press, 1995.

6. R.G.G. Cattell, Douglas Barry, Dirk Bartels, Mark Berler, Jeff Eastman, Sophie Gamerman, David Jordan, Adam Springer, Henry Strickland, and Drew Wade, editors. *The Object Database Standard: ODMG 2.0*. Series in Data Management Systems. Morgan Kaufmann, San Francisco, CA, 1997.

7. Sun Microsystems Inc. Java remote method invocation, May 1997. http://java.sun.com:80/products/jdk/1.2/docs/guide/rmi/spec/rmiTOC.doc.html.

8. A. Kaplan, J. V.E. Ridgway, and J.C. Wileden. Why IDLs are not ideal. In *Proceedings of the Ninth IEEE International Workshop on Software Specification and Design*, Ise-Shima, Japan, April 1998.

9. Alan Kaplan and Jack C. Wileden. Toward painless polylingual persistence. In *Seventh International Workshop on Persistence Object Systems*, Cape May, NJ, May 1996.

10. Daniel J. Barrett, Alan Kaplan, and Jack C. Wileden. Automated support for seamless interoperability in polylingual software systems. In *The Fourth Symposium on the Foundations of Software Engineering*, San Francisco, CA, October 1996.

Clemson University, Department of Computer Science
Box 341906, Clemson, SC 29634-1906 USA
kaplan@cs.clemson.edu, http://www.cs.clemson.edu/~kaplan

Alan Kaplan
Last modified: Thu Mar 18 11:29:34 EST 1999