ILU: Inter-Language Unification via Object Modules

Bill Janssen, Mike Spreitzer

Xerox Corporation
Palo Alto Research Center

15 August 1994

1 Introduction

A key characteristic of modern software engineering is the multiplicity of computing environments and languages. In theory, this variety allows an engineer to choose tools and techniques particularly well suited to a specific problem. Unfortunately, making a choice of any particular language frequently prohibits the use of packages implemented in other languages. This phenomenon restricts both the productivity and creativity of software designers, by partitioning the universe of available software "parts" into disjoint regions.

It also tends to force software tool&die vendors, who wish to build widely useful sub-systems, to employ languages such as C -- a low-level language which lacks many useful engineering features, such as automatic memory management, a useful macro facility, an exception model, an object system, module interfaces, and a concurrent processing model (threads). Competent software engineers who must work in C wind up implementing some or all of these features as part of their package -- often in a way which is incompatible with other implementations provided as part of other packages. This balkanization of technology is also apparent in the networking world, at several levels, but notably with the emergence of several competing standards for remote procedure calls. Xerox is still using their XNS system, with the Courier RPC protocol [Xerox81]. The OSF is promoting their Distributed Computing Environment (DCE) [OSF91]. Sun Microsystems and others are using ONC RPC [SMI86]. Novell has its own 'network operating system' called NetWare. An industry consortium, the Object Management Group (OMG), is devising a standard object architecture called CORBA [OMG91] that includes an RPC model. The odds seem good that few (if any) of these systems will be able to work with any of the others.

Our answer to this problem is called Inter-Language Unification. ILU is based on a notion of *modules* that represent encapsulated environments. A program is constructed of modules. Each module has a well-defined interface description, which identifies the appearance of the module, with respect to other modules. Each interface is described in a generic interface description language that allows definition of a extensible set of programming-language types, exceptions, and constants. Each module may be implemented with whatever combination of programming languages and environments are appropriate for the problem. A set of specified translation protocols allow automatic translation of data as it passes through any interface. Network interfaces and RPC protocols are treated as just one form of translation protocol, and are thus subsumed in this approach.

Since the interface specification language is programming-language independent, it forms a useful tool for design of application protocols that are inherently inter-modular, such as printing, filing, and information discovery protocols. This design can proceed independently from the actual implementation of any particular module. Different modules can export different parts of the protocol.

From the module interface description, a set of tools automatically generate renderings of this interface in each of several target languages. New target languages may be added without disturbing existing modules. A programmer working in any particular target language may view any module as being implemented in that target language, by referring to the appropriate rendering of the interface. We call this property *apparent homogeneity*, as it creates the illusion of a homogeneous universe of modules.

2 ILU Basics

The ILU model is based on the notion of modules, described with object-oriented interfaces. Each module may export a set of types, including object classes, which are defined in the interface description of the module -- each module has only one interface. The interface is intended to fully describe the external behavior of the module, without regard to the programming language or environment used to implement the module. All calls into the module are performed by invoking some *method* on some *instance* of some object class, exported from that module. A module may exist in the same address space as its user, or may be in some other address space, possibly on some other computer.

ILU defines a module description language, ISL. ISL was designed with several constraints: it had to provide for modern software engineering constructs, particularly objects and exceptions; any description written in it had to have reasonable interpretations for each of several target languages -- the languages C [ANSI89], Common Lisp [Steele90], C++ [Ellis90], and Modula-3 [Nelson91a] were chosen as a reasonably broad target set; and it had to be able to reasonably describe existing interfaces of various kinds, particularly network RPC interfaces such as ONC RPC or Xerox Courier.

The actual language is modeled after that used by Modula-3 to describe interfaces. Each interface description has a header, naming the interface and specifying a version, followed by a list of type declarations, exception declarations, and constant declarations. An interface may ‘‘import’’ other interfaces, and use elements defined in them.

ISL has several primitive scalar types: 16-, 32, and 64-bit signed and unsigned integers, 8-bit bytes, 32-, 64, and 128-bit IEEE floating point values (the 128 bit format uses a form of ‘extended double-precision’ defined by Sun), 16-bit ISO 10646/Unicode [ISO90] and 8-bit ISO Latin-1 characters, and Boolean values. These types can be combined with several type constructors: arrays (fixed-length multi-dimensional arrays), sequences (lists of some type), records (C structs), unions (values that may be one of several different types), and enumerations (a type that allows one of a specified set of abstract values). Support for pointer-constructed data is provided in that arrays, unions, sequences, and record fields may be specified as *optional*, which defines an implicit union of the specified field type and the NULL type.

ISL supports termination model exceptions, which causes the calling stack to be unwound out of the module in which the exception is raised, when it is raised. Optionally, an exception may have a type, which allows a value of that type to be associated with an instance of the exception.

The ISL class language does not attempt to be a full class implementation language, or even a "full" specification language. We assume that the ISL specification will be rendered in some appropriate class language, for each target language, that will actually flesh out the details of the class's implementation. The abstract class system defined for ILU is a multiple-inheritance system. The only effect of this inheritance is a conceptual one -- it is used to indicate that the subclass provides all the methods described by its superclasses, plus any other methods described directly in the subclass description. No inheritance of code is required, though it may be possible to use the inheritance hierarchy for greater leverage, including code reuse, in some target languages. ILU object types have no slots or instance values associated with them.

Instances of a class are created in a *server* module -- these are called *true* instances -- and given out to *client* modules, which hold *surrogate* instances. Client modules may pass surrogate instances to other modules, which become in turn clients of the instances' servers. In general, methods are invoked on objects in client modules, and serviced in server modules. It is important to realize that the terms ''client'' and ''server'' are used solely to describe a module's relationship to an instance; a module could be a client of some other module with respect to one instance, but a server of that other module with respect to a different instance.

Methods may be defined on each class. Each method has a procedural signature which allows the designer to specify the types of its arguments, its return type (if any), and what exceptions it might raise (if any). Method arguments may be specified as either *in*, *out*, or *inout*. Methods may be tagged as *functional*, indicating that any call to that method with the same arguments will return the same result; this allows implementation of caching on the client side. Methods which neither return a value nor raise an exception may be tagged as *asynchronous*, indicating that no acknowledgment of the method request will be made to the caller.

Since the ILU class language is an artificially limited language, it is reasonable to assume that the implementation of a true class will have additional private generic functions and instance variables that can not be exported across module boundaries. If a method is invoked with other instances as arguments, in some cases these instances will have to be true instances in the same module as the method *discriminant*, the instance to which the method's 'message' is 'sent'. To provide for this case, method arguments that are object types may be marked as *sibling*, implying that their true instance must be in the same module as that of the method discriminant.

For purposes of resource management, it is useful for a server module to know when no clients have surrogate instances of a true instance, to perform distributed garbage collection. In a networked environment, ILU uses a simple reference-counting + timeout scheme. A complication of this scheme is that servers must be able to do periodic liveness checks on *dormant* clients, that is, clients which have not communicated with the server module for some period of time; this requires that each client module export some "callback" class, so that a server can "ping" the client.

3 Current State of ILU

ILU is currently at version 1.6. A parser for ILU ISL has been implemented. Stub generators and language-specific support libraries have been written for the ANSI C target language, using the language mapping prescribed by OMG's CORBA standards 1.1 and 1.2, and for Modula-3, FORTRAN 77, C++, Common Lisp, Python, and GNU Emacs Lisp, using our own language mappings. Support for the script language Tcl is in progress.

All the language-specific support libraries share a common kernel library, written in ANSI C using POSIX system calls, which implements an object database, support for garbage collection, support for binding, an abstract threads system, an abstract data normalization system, and an abstract data communication system. The kernel library does not currently support multiple languages in the same address space, pipes, or authentication. The abstract modules in the kernel are designed using an object protocol to support extension and modification. A concrete data normalization module supporting ONC RPC's XDR conventions [SMI86] has been implemented. Concrete communication modules using ONC RPC conventions over TCP/IP and UDP/IP, and in-memory conventions, have been implemented. Work is underway on a concrete data normalization module using OSF DCE RPC's NDR formats.

The kernel library and Common Lisp implementation have been ported to the Silicon Graphics Iris/Indigo IRIX5 architecture, in addition to running on the Sun. The kernel library and ANSI C implementation have been ported to the Linux operating system, running on top of PPCR. The kernel and C++ implementation have been ported to the Macintosh.

Many people at Xerox PARC have contributed to the ILU project: Mark Stefik, Doug Cutting, Bill Schilit, Carl Hauser, Antony Courtney, Farrell Wymore, Denis Severson, Jan Pedersen, Ramana Rao, Doug Terry, Marvin Theimer, Bill van Melle, George Robertson, Scott Minneman, Ian Smith, Don Kimber, and Gordon Kurtenbach are among them.

4 References

[Xerox 81] [Xerox Corporation]; COURIER: THE REMOTE PROCEDURE PROTOCOL; Xerox, Stamford, CT, 1981, <order number XSIS 038112>.

[OSF91] [Open Systems Foundation]; INTRODUCTION TO OSF DCE (Revision 1.0), Chapter 3.2: "Remote Procedure Call"; OSF, ??, 31 December 1991.

[SMI86] [Sun Microsystems Inc.]; NETWORKING ON THE SUN WORKSTATION, "Remote Procedure Call Specification"; SMI, Mountain View, CA, 1986, <part # 800-1324-03 B>.

[ANSI89] [American National Standards Institute, Inc.]; AMERICAN NATIONAL STANDARD FOR INFORMATION SYSTEMS -- PROGRAMMING LANGUAGE -- C; ANSI, ??, 1989, <X3.159-1989>.

[Steele90] Guy L. Steele; COMMON LISP, THE LANGUAGE (Second Edition); Digital Press, 1990, <ISBN 1-55558-041-6>.

[Ellis90] Margaret Ellis and Bjarne Stroustrup; THE ANNOTATED C++ REFERENCE MANUAL; Addison-Wesley, Reading, MA, 1990, <ISBN 0-201-51459-1>.

[Nelson91a] Greg Nelson; SYSTEMS PROGRAMMING WITH MODULA-3; Prentice-Hall, 1991, <ISBN 0-13-590-464-1>.

[ISO90] [Internation Standards Organization]; DRAFT INTERNATIONAL STANDARD -- INFORMATION TECHNOLOGY -- UNIVERSAL CODED CHARACTER SET (UCS); ISO, ??, 1990, <ISO/IEC DIS 10646>.

[OMG91] [Object Management Group]; THE COMMON OBJECT REQUEST BROKER: ARCHITECTURE AND SPECIFCATION; OMG, Framingham, MA, 10 December 1991.

5 Addresses and Information

Bill Janssen <janssen@parc.xerox.com> (415) 812-4763 FAX: (415) 812-4777
Mike Spreitzer <spreitze@parc.xerox.com> (415) 812-4833

Xerox Palo Alto Research Center, 3333 Coyote Hill Rd, Palo Alto, CA 94304

ILU URL: ftp://parcftp.parc.xerox.com/pub/ilu/ilu.html