ILU:
Inter-Language Unification

(that's *eye-loo*!)

Bill Janssen
Mike Spreitzer

Xerox Corporation   Palo Alto Research Center

THE DOCUMENT COMPANY
XEROX

What's ILU?

- a technology for specifying procedural interfaces of software modules in a clear language-independent fashion

- a technology for implementing software modules that export such interfaces, in multiple languages

- a technology for using such modules in a language-independent, inter-address-space-capable, inter-machine-capable way

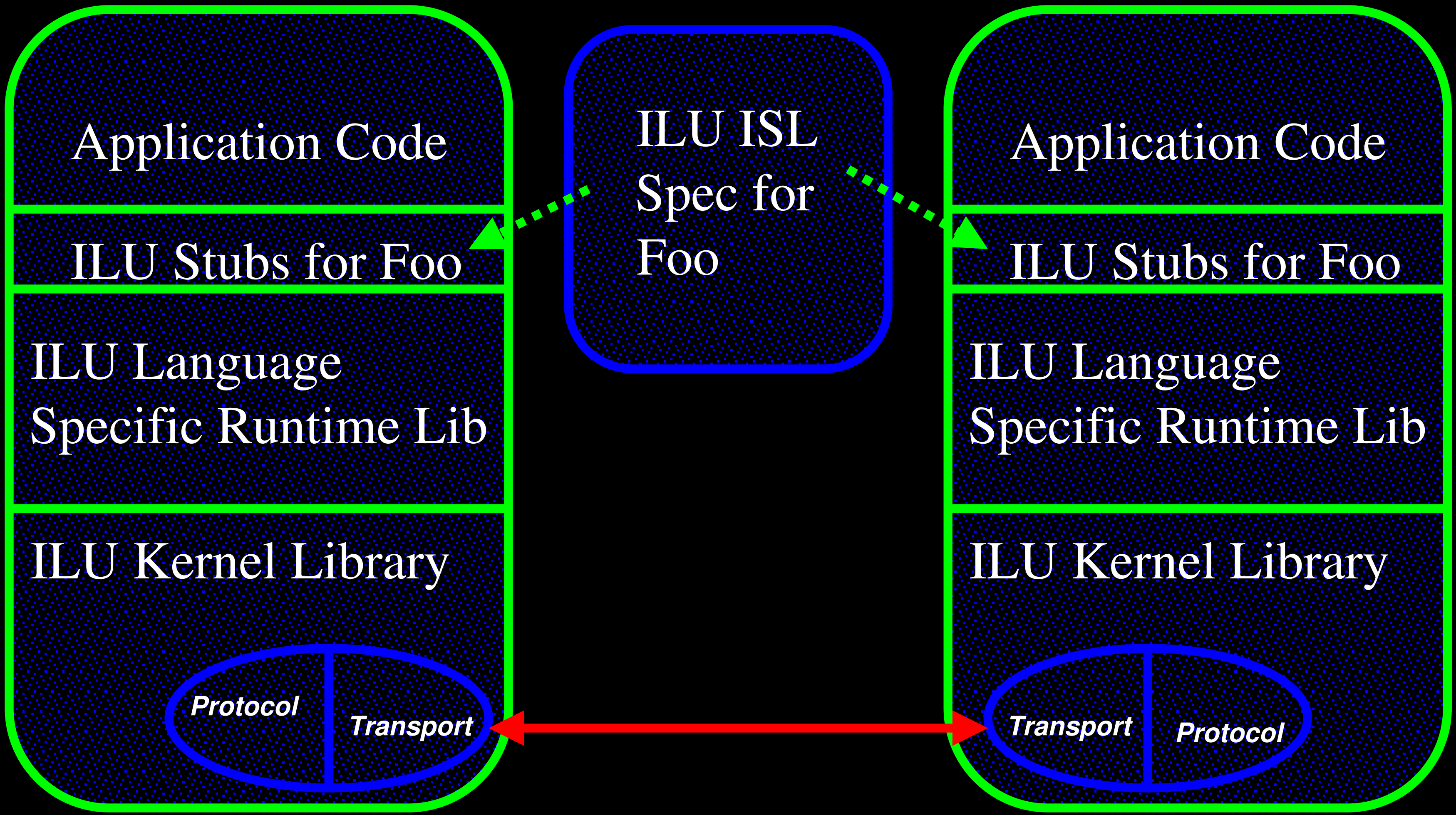
- a free OMG CORBA implementation :-)

THE DOCUMENT COMPANY
XEROX

Interoperability

- Between different programming languages
  (Common Lisp, Modula-3, ANSI C, C++, Python, ...)
- Between different operating systems
  (UNIX, Cedar, MS-Windows, PPCR, MacOS, NetWare)
- With existing clients and services and RPC protocols
  (ONC RPC, XNS Courier, DCE RPC, OMG CORBA)
- Between different binding models
  (seamless, same address space, IPC, RPC, messaging,...)

Two ILU Programs Interacting



THE DOCUMENT COMPANY
XEROX

ILU: Inter-Language Unification

A Simple Example

- The four function calculator:
  - Add
  - Subtract
  - Multiply
  - Divide (with exception on division by zero)
- Add a memory
- Add a ‘‘register tape’’

Calculator Interface

```
INTERFACE USENIX BRAND "v1";

EXCEPTION ZeroDivisor;
TYPE Calculator = OBJECT COLLECTIBLE
   METHODS
       Clear(),
       CurrentValue() : REAL,
       Add (value : REAL),
       Subtract (value : REAL),
       Multiply (value : REAL),
       Divide (value : REAL) RAISES ZeroDivisor END
   END;
TYPE Factory = OBJECT METHODS
       NewCalculator () : Calculator
   END;
```

A Calculator Server (in Python)

```
class Calculator_impl (USENIX_skel.Calculator):
   def __init__(self):
       self.value = 0
   def Clear (self):
       self.value = 0
   def CurrentValue (self):
       return self.value
   def Add (self, addend):
       self.value = self.value + addend
   def Subtract (self, subtrahend):
       self.value = self.value - subtrahend
   def Multiply (self, multiplicand):
       self.value = self.value * multiplicand
   def Divide (self, divisor):
       if divisor == 0: raise ZeroDivisor
       self.value = self.value / divisor
```

The Calculator Server (continued)

```
class Factory_impl (USENIX__skel.Factory):
   def __init__(self, server=None, instHandle="0"):
       self.IluInstHandle = instHandle
       self.calcs = []
       if server:
           self.IluServer = server
       else:
           self.IluServer =ilu.CreateServer("USENIX_Fact")
   def NewCalculator (self):
       newcalc = Calculator_impl()
       self.calcs.append(newcalc)
       return newcalc
```

The Calculator Server (finished)

```
def main():

   the_factory = Factory_impl()
   the_factory.IluPublish()
   ilu.RunMainLoop()
```

A Calculator Client (in C)

```
#include <stdio.h>
#include <stdlib.h>
#include <USENIX.h>

int main (int ac, char **av)
{
   USENIX_Factory factory;
   USENIX_Calculator calc;
   CORBA_Environment env;
   /* some other necessary declarations omitted... */

   USENIX_Initialize();

   factory = (USENIX_Factory) ILU_C_LookupObject(
                                     "0@USENIX_Fact",
                                     USENIX_Factory__MSType);
```

Calculator Client (2)

```
calc = USENIX_Factory_NewCalculator (factory, &env);

if (! ILU_C_SUCCESSFUL(&env)) {
   fprintf (stderr, "Couldn't create calculator.\n");
   fprintf (stderr, "ILU signals error <%s>.\n",
       ILU_C_EXCEPTION_ID(&env));
   exit(1);
};

if (!USENIX_Calculator_Clear (calc, &env),
       ILU_C_SUCCESSFUL(&env)) {
   fprintf (stderr, "Clear signals error <%s>\n",
       ILU_C_EXCEPTION_ID(&env);
};
```

Calculator Client (3)

```
current_val = 0.0;
do {
   printf ("%.5f\n>", current_val);
   *buf = 'q';
   line = gets(buf);
   switch (buf[0]) {
       case '+':
           val = atof(buf + 1);
           if (!((USENIX_Calculator_Add (calc, &env, val),
                   ILU_C_SUCCESSFUL(&env)) &&
    (newval = USENIX_Calculator_CurrentValue (calc, &env),
                   ILU_C_SUCCESSFUL(&env)))) {
           fprintf (stderr,
               "Operation <%s> signals <%s>.\n",
               buf, ILU_C_EXCEPTION_ID(&env)); };
       break;
```

Calculator Client (4)

```
    case 'q':
        line = NULL;
        break;
    default:
        fprintf (stderr, "Invalid operation <%s>.\n",
            buf);
        fprintf (stderr, "Valid ops are +, -, *, /, q.\n");
    };
} while (line != NULL);

return 0;
}
```

Add Functionality via Subclassing

```
EXCEPTION NoValueInMemory;

TYPE MemCalculator = OBJECT COLLECTIBLE
   SUPERTYPES Calculator END
   METHODS
      MemSet (value : REAL),
      MemGet () : RAISES NoValueInMemory END,
      MemAdd () : RAISES NoValueInMemory END,
      ...
   END;
```

Constructed Types

```
TYPE OpType = ENUMERATION
      Clear, Add, Subtract, Multiply, Divide,
      MemAdd, MemSubtract, MemMultiply, MemDivide
   END;
TYPE PossibleValue = OPTIONAL REAL;
TYPE Operation = RECORD
      kind : OpType,
      val  : PossibleValue
   END;
TYPE CalcTape = SEQUENCE OF Operation;
TYPE MemCalculatorWithTape = OBJECT COLLECTIBLE
   SUPERTYPES MemCalculator END
   METHODS
      GetTape () : CalcTape
   END;
```

ILU and CORBA

ILU Advantages:

- Network garbage collection for objects
- 64 bit integers, 128 bit floating point
- Unicode (ISO 10646 16-bit) characters
- The ‘‘optional’’ type for value references
- True ‘‘asynchronous’’ methods
- ‘‘Functional’’ methods for client-side caching
- Bulk data transfer (not in 1.7)
- ‘‘Sibling’’ arguments for enhanced private protocols
- Compatibility with existing RPC protocols
- No ‘‘any’’ or ‘‘Object’’ types as in OMG IDLis

ILU Network Garbage Collection

- Server maintains list of ‘‘interested’’ clients
- Clients register/unregister interest automatically on receipt of a ‘‘collectible’’ instance (not all object types are collectible)
- Uses pings from server to client, with timeout T1, to reap dead clients
- Uses different timeout T2 to guard against race conditions
- Integrated with language GC if possible

ISL "optional" Types

- Union of base type T with Null type

- Allows recursive data structures such as B-trees:

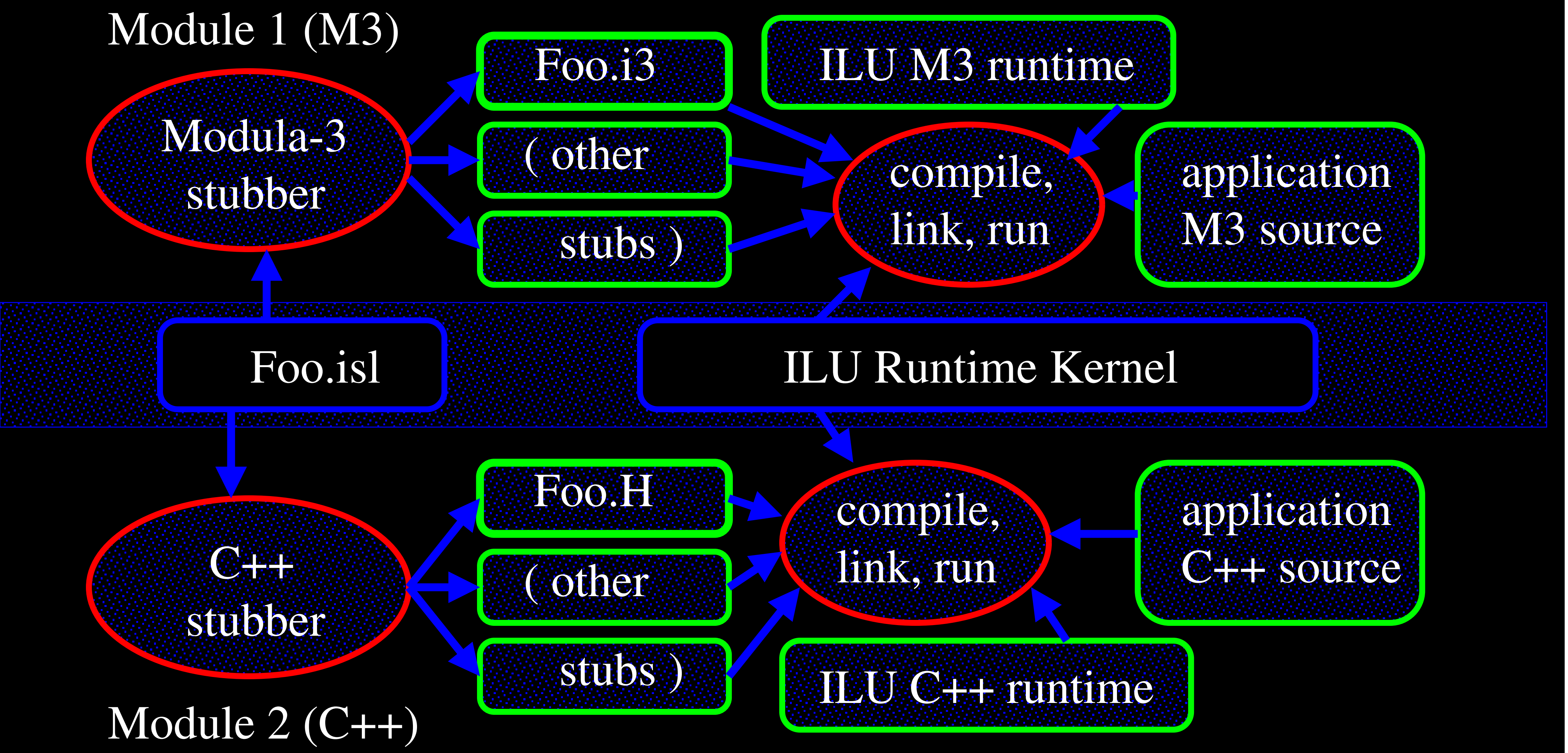
```
TYPE OptNode = OPTIONAL Node;
TYPE Node = RECORD
   data : SomeType,
   left : OptNode,
   right : OptNode
END;
```

- Maps to pointers in most pointer-based languages

Asynchronous & Functional Calls

- ASYNCHRONOUS tells server not to bother replying, and client not to wait for reply
- ASYNCHRONOUS is not the same as OMG IDL ‘‘oneway’’ -- which really means ‘‘unreliable’’
- FUNCTIONAL guarantees that method will always return same results when called with same arguments, thereby allowing client-side caching

ILU Implementation



THE DOCUMENT COMPANY
XEROX

The ILU Kernel

- Provides core ILU functionality

- Highly portable: POSIX.1 & ANSI C & select(2)
- TCP & UDP support also requires BSD sockets

- Self contained but optional event loop
- Thread safe and threaded
- Error system implemented with ANSI C macros
- Modular: new components can be added at run-time or compile-time

Replaceable Kernel Modules

- Mapping external object reference (SBH) to object via ‘‘object tables’’
- Protocols (data normalization, packet layout, etc., such as ONC RPC, DCE RPC, OMG CORBA IIOP)
- Transports (getting bytes from one place to the next, as with TCP/IP, email, UDP/IP, AppleTalk)
- Threads and Event Loops

ILU Kernel Error System

- Heap-allocated ‘‘error objects’’ are returned
- Different objects for each type of error
- Fields of the error object indicate the local state at the time of the error’s occurrence
- Errors can be chained
- Extensive macrology for manipulating error values in a type-safe manner, both from C and from non-C languages
- Heap-allocation errors handled specially
  - ‘‘soft’’ and ‘‘hard’’ allocation failures
  - user-specifiable callback routine to free allocated space and re-try malloc()

ILU Kernel Error System (2)

- Internal assertions handled with _ilu_Assert()
- Fatal errors can cause application's choice of:
  - Core dump
  - Program exit (with application-specified exit value)
  - Endless sleep loop (for ''live'' debugging)

THE DOCUMENT COMPANY XEROX

ILU Kernel Thread-Safety

- Data structures and critical sections are protected by a generic mutex construct
- A partial order is prescribed for acquiring and releasing locks
- Locking requirements of all internal and external procedures are documented
- A notation derived from that used in DEC SRC Modula-3's Trestle toolkit is used for documentation
- Language-specific interfaces are also documented as to locking requirements, using the same notation
- An interface is exported to allow specific thread systems to describe their mutex system to the ILU kernel

ILU Kernel Event Loops

- Kernel provides internal re-entrant event loop that handles file descriptors and timed events (alarms)
- Applications may use the ILU event loop and alarms for their own purposes
- Applications may instruct the kernel to use a different event loop instead of the ILU internal event loop
  - Xt, Tk
- Applications may instruct the kernel to use a different event loop in addition to the ILU internal event loop
  - XView
- Threaded runtimes may instruct the kernel to use threads instead of any event loop

ILU Kernel Symbol Management

- All exported symbols begin with "ilu_".
- All exported symbols documented in "iluxport.h".

- All internal (but global) symbols begin with "_ilu_".
- All internal (but global) symbols documented in "iluntrnl.h".

- Internal structs hidden (opaque pointers used)
- Pseudo-object style used (via function pointers in structs)
  to allow subtyping

ILU Kernel Resource Management

- Set of empty macros used to ‘‘decorate’’ function prototypes (currently somewhat scantily)

- PASS(type) means that the parameter or return value belongs to the callee or caller, respectively
- RETAIN(type) means that the parameter or return value belongs the caller or callee, respectively
- GLOBAL(type) means that neither the caller or callee owns the parameter

- OPTIONAL(type) means that a NULL pointer may be passed in place of valid value (only used for pointer values)

Current State

- Version 1.6.4 released in May 1994
- Version 1.7 in beta test
  - Support for C, C++, Common Lisp, Modula-3, Python
  - Support for CORBA's OMG IDL
  - Runs under Solaris 1 & 2, IRIX, HP-UX, Linux, etc.
- Support for Windows/NT and Windows 3.1 under way

- URL: ftp://ftp.parc.xerox.com/pub/ilu/ilu.html
- HTML manual:

  ftp://ftp.parc.xerox.com/pub/ilu/1.7beta/manual-html/manual_toc.html
- These slides: ftp://ftp.parc.xerox.com/pub/ilu/USENIX/