Sweet Harmony: The Talk/C++ Connection

Harley Davis Pierre Parquier Nitsan Séniak

Ilog, S.A.
2, avenue Galliéni
94253 Gentilly, France
talk-support@ilog.fr

Abstract

A tight, transparent, and portable integration between C++ and LISP is desirable and feasible. This paper describes the C++ interface supplied with [6], a modern LISP dialect which extends the proposed ISLISP standard with a module system [3], a metaobject protocol, and an extensive set of libraries. The interface parses C++ header files and generates C++ stub interface files, as well as TALK modules which implement proxy classes and TALK foreign function definitions for C++ functions. The programmer then has nearly complete access to the functionality of a C++ library.

1 Introduction

LISP and C++ [2] provide complementary services: C++ is efficient, statically typed, relatively small, and widely used; while LISP is dynamic, productive, flexible, and extensible. Instead of battling for dominance, we should try to take advantage of each language, using each where appropriate and joining the two together to rapidly produce efficient, working software. Such a strategy allows LISP to leverage off the considerable efforts currently being invested in developing large C++ libraries.

For example, distributed object systems are becoming available in C++; by connecting to such systems directly LISP can immediately gain the benefits of distribution. Similarly, graphic libraries written in C++ provide high performance. However, it is difficult to rapidly construct complex graphic applications because of the long turnaround involved in testing and experimenting with various possibilities. LISP, as a dynamic language, is ideal for exploratory programming. By making a C or C++ graphic library available in LISP, both high performance and high productivity can be maintained.

The TALK/C++ connection is an attempt to bridge the gap between these two languages. It goes much further than typical LISP foreign function interfaces [9] by merging the object models in LISP and C++. Specifically, it provides the following facilities:

- C++ classes are mirrored by equivalent TALK classes.
- C++ class data members are seen as slots in these classes.
- C++ class constructors are available to the TALK programmer. The TALK function `delete` deletes a C++ object by calling the appropriate destructor function in C++.
- Instances of these classes created in C++ are visible in TALK as TALK objects correctly typed to the extent that the C++ class provides type information.
- Because mirrored C++ classes use an extension of the TALK metaobject protocol, object inspection and browsing tools automatically work for C++ objects. (See [7] for an excellent overview of the capacities of a metaobject protocol.)
- C++ functions — normal, overloaded, and member — are available to the TALK programmer as generic functions which discriminate on the classes and number of their arguments.
- TALK functions can be passed as callbacks to the C++ library.

The entire interface to a C++ library can thus be made available to the TALK programmer. Programming such a library in TALK rather than in C++ can provide a number of benefits:

- Because of its interactive development environment, TALK provides an ideal medium in which to prototype applications using a C++ library.
- TALK's dynamic typing and automatic memory management relieve the programmer from worrying about low-level details, meaning that programs are more likely to be completed on time.
- TALK provides a portable mechanism for creating higher-level libraries and executables, saving the programmer porting costs.
- TALK's powerful high-level class libraries provide a ready-made toolset for common programming abstractions such as heterogenous lists, vectors, hash tables, and formatted i/o.

The TALK/C++ interface works best when extending or using a relatively independent C++ library. Mixed-language

LISP 94 - 6/94 Orlando, Florida USA
© 1994 ACM 0-89791-643-3/94/0006..$3.50

programming at the application level is not currently well-supported because of the difficulties of debugging such an application.

The TALK/C++ interface provides three possible levels of integration with a C++ library. In the best case, the C++ library provides a dynamic typing scheme for its major classes. Then the interface provides a completely non-intrusive encapsulation of a C++ library. The programmer does not need to modify the C++ library in order to generate its interface in TALK. If no dynamic typing scheme exists, there are two options: First, the programmer can modify the library to use a simple dynamic typing scheme provided by the interface. In this case, the binding is complete but slightly intrusive. If the library cannot be modified, the final possibility is to generate an interface using static typing, in which case the exact class of arguments and return values for C++ functions must be specified. In this worst case, the binding is non-intrusive but incomplete, unless the C++ program has no derived classes.

A C++ object is represented by a handle in TALK. This handle is a TALK object of the appropriate mirror TALK class containing a pointer to the C++ object.

C++ objects are in general not amenable to garbage collection by the TALK memory manager. They must be allocated and deleted explicitly by the programmer, as in C++. However, this can be done in TALK: Each class has at least one constructor which calls the C++ operator `new`, and the TALK generic function `delete` deletes a C++ object. The handle objects themselves can be garbage collected — but the programmer must be careful about keeping a handle to a deleted C++ object. The consequences of using such a handle are undefined, but are unlikely to be useful. We considered the generalized use of handle finalizers to automatically destroy C++ objects after handle collection, but this solution was rejected as being incompatible with C++ semantics and thus too dangerous [5]. However, for certain simple classes without side-effects, it is feasible to automate the deletion process when the handle is collected.

2 Generating a Binding

A C++ library consists of a set of header files and a set of object files. The C++ module parser is used to analyze these header files and generate a TALK interface from such a library:

1. The parser scans the header files to extract relevent information such as the names and inheritance of classes, data member names and types, type definitions declared using `typedef`, and the prototypes of operators and functions.

2. The parser presents this information to the user, who selects a subset of the interface to be made available in TALK. The user must also clarify any ambiguities and specify the dynamic typing scheme to be used for the interface. The parser can also function automatically to produce a complete first-cut interface using reasonable defaults.

3. The parser then generates a TALK module containing the appropriate declarations.

4. The compilation in TALK of this module generates a stub file containing functions which create the C++ classes, access the data members, and call the appropriate functions. All of these stubs are specified as having linkage `extern "C"`. This file is compiled by the local C++ compiler and associated with the TALK module. The stub file generation is key to the approach; by using them, we avoid portability problems associated with trying to second-guess C++ compilers' object layout and name mangling schemes.

5. The generated TALK module itself defines the mirror TALK classes, which are instances of a special metaclass, as well as foreign function definitions for the generated stubs. In TALK, slot access is provided by storing an anonymous reader and writer function for each slot in the class object (see [1]). The metaclass for C++ objects in TALK supports anonymous foreign functions which can be used as slot accessor functions.

6. The TALK interface module can be either linked with a TALK library or executable, or loaded dynamically on most systems. Dynamic loading of C++ code presents certain portability problems due to the possibility in C++ of defining top-level forms which must be executed when the code is loaded.

```
#include "IndexedCollection.h"

class OrderedCollection : public IndexedCollection {
public:
        int size_; // fake slot for example
        classType isA() const
           { return OrderedCollectionClass; }
        char* className() const
           { return "OrderedCollection"; }
        int     includes (const Object*) const;
        Object* remove  (const Object*);
        Object* forEach (funPtr, void* = 0) const;
        Object* at      (int i) const;

        // Constructor
        OrderedCollection(int = BaseCollectionSize);

        // Destructor
        ~OrderedCollection() { delete _vec; }
};
```

Figure 1: A Typical C++ Header File.

2.1 Example

Below we present an example inspired by the NIH class library [4], a freely available C++ library implementing a number of SmallTalk-like classes. This example is provided to give a flavor of the interface; this article does not document the various macros in the interface in depth.

Figure 1 shows part of the C++ header file `OrderedCollectior` Figure 2 shows the automatically generated TALK interface code for this header file, plus some handwritten code which defines new functionality for this class in TALK.

The TALK forms implementing the interface are designed to match as closely as possible the C++ program structure.

```
;;; Generated code
(cpp-include "IndexedCollection.h")

(define-cpp-type <OrderedCollection>*
  talk-class: <OrderedCollection>
  cpp-type: "OrderedCollection"
  interface: NihObject)

(define-cpp-type <Object>*
  talk-class: <Object>
  cpp-type: "Object"
  interface: NihObject)

(define-cpp-function-type funPtr (<Object>* void*)
   void)

(define-cpp-class <OrderedCollection>
      (<IndexedCollection>)
  ((size_ type: <integer>)
   (includes (<Object>*) <integer>)
   (remove (<Object>*) <Object>*)
   (forEach (funPtr optional: void*) <Object>*)
   (at (<integer>) <Object>*))
  constructor: (optional: <integer>))

;;; Handwritten code
(defcallback remove-callback funPtr (obj arg)
  (printf "Removing %A from %A.\n" obj arg)
  (->remove obj arg))

(defgeneric OrderedCollection-p (object))
(defmethod OrderedCollection-p
    ((object <OrderedCollection>))
  t)
(defmethod OrderedCollection-p
    ((object <object>))
  ())

(defun empty-collection (obj)
  ;; OBJ is an OrderedCollection.
  ;; Remove all elements from an
  ;; ordered collection.
  (assert (OrderedCollection-p obj))
  (->forEach obj #'remove-callback))
```

Figure 2: Generated TALK Interface File

The form `define-cpp-type` defines a foreign data type interface for each C++ type. This is necessary because the C++ notion of type does not correspond its notion of class; types are declined by various modifiers. In this example, we define a type named `<OrderedCollection>*` corresponding to the C++ type `OrderedCollection*` — in other words, a pointer to an instance of `OrderedCollection`. The module parser generates one such `define-cpp-type` form for each type encountered in the C++ header files. The parser also generates types for each `typedef` encountered. This type is then specified when declaring function argument and return types, and is used by the stub generator to correctly dereference pointers and do numeric conversion when necessary.

Because callbacks present a somewhat more complicated problem, as we shall see later, types which are pointers to functions are declared using the form `define-cpp-function-type`. This allows such types to be specified for callback functions written in TALK.

The form `define-cpp-class` defines a proxy TALK class for a C++ class. The syntax of `define-cpp-class` is reminiscent of `defclass`, but allows the specification of member functions in addition to data slots. In this case, the slot `size` is a data member while the other slots — `includes`, `remove`, `forEach`, and `at` — correspond to member functions. Member functions declare their argument types and return type; each such type must be defined by `define-cpp-type`. Basic types are predefined by the system; other types are generated by the parser. Proxy class declarations also contain signature declarations for constructor functions. While C++ constructors are called indepedently of memory allocation, TALK proxy constructors always call `new` to allocate a new instance in the heap; TALK has no provision for heap allocation of foreign data.

Finally, the form `defoverloaded` — not used in this example — defines a top level overloaded function signature.

The handwritten code TALK following the automatically generated declarations shows a simple use of these declarations. The TALK programmer can write normal functions, generic functions, and methods which specialize on proxy classes. A callback, specified by the form `defcallback`, allows the programmer to write TALK functions which can then be passed to C++ functions which take functions as arguments. Because these functions are exactly typed in C++, a callback definition must mention the function type (defined by `define-cpp-function-type`) to which it conforms.

3 Issues

A number of orthogonal issues provide a challenge to a complete and efficient interface in TALK to C++ libraries. Among other issues, we can cite:

Naming: If the TALK interface to a C++ library is to be transparent and simple to use, we need a set of simple and universal naming conventions for the TALK entities generated by the interface. We must also avoid name collisions with predefined or user-defined TALK names.

Typing: C++ provides no default dynamic typing mechanism. Some C++ libraries offer no dynamic typing at all. TALK has dynamic typing for all objects. In addition, C++ types include several distinctions not recognized in TALK— such as the distinction between

signed and unsigned integers as well as const pointers and objects.

Representation: The representation of a C++ object is decided upon by the compiler, and is not directly accessible; only the public class interface provides a portable means of accessing and creating objects. TALK objects have a particular representation and are strongly connected to the memory manager.

Arguments: C++ supports call-by-value and call-by-reference argument passing styles. Using call-by-value has the unfortunate effect of copying the value during argument passing, and can thus be very inefficient for even small objects. Therefore, most C++ programs use either call-by-reference or call-by-value for object pointers rather than objects themselves. The TALK interface must be able to support both argument passing styles.

Callbacks: C++ libraries — especially graphics libraries — allow the programmer to dynamically specify actions through the use of callbacks, passing C++ functions as arguments. TALK functions are not called using the same protocol as C++ functions, and require data manipulation when passed foreign objects.

Operators: C++ libraries are defined not only in terms of classes, slots (members), and functions, but also in terms of overloaded operators. For a complete integration with C++, the interface must provide some way to access these operators.

In this section, we will describe each of these problems in detail, and the approach used in the TALK/C++ interface to deal with them.

3.1 Naming

To simplify the use of the TALK interface to a C++ library, we need a set of naming conventions. We have chosen the following rules:

- The name of the TALK surrogate for a C++ class always has the same name as that class in C++, with angle brackets added. This is necessary because all TALK class names are enclosed in angle brackets.
- Top level C++ functions are represented by special TALK generic functions of the class `<overloaded-function>`. TALK overloaded functions discriminate on the number of arguments passed as well as the class of the arguments, simplifiying the treatment of C++ overloading.
- All C++ member functions are represented in TALK by special generic functions (of the class `<member-function>`) with the same name as the C++ function, prefixed by the sequence `->`. The arrow prefix was chosen to avoid any conflict with predefined or user-defined TALK function names, and to remind users of the C++ arrow operator which accesses object pointer members. TALK member functions follow C++ semantics by looking up an overloaded function in the scope of the class of their first argument.

3.2 Typing

Because dynamic typing is useful for many applications, such as persistance, many C++ libraries provide some sort of *ad hoc* dynamic typing feature. Indeed, this is so widespread that the C++ standardization committee has approved a standard dynamic typing mechanism known as RTTI (Run Time Type Identification). However, few compilers as yet implement this standard, and even fewer libraries can use it.

TALK objects are systematically dynamically typed. The function `class-of` can be applied to any TALK object to discover its type. If we are to provide C++ objects in TALK, these objects must be dynamically typed. Each time that TALK sees a C++ object, it must be able to determine the TALK class corresponding to the object so that its identity can be determined and `class-of` (and hence all dynamic typing operations) can be applied to the object.

Almost all of the *ad hoc* dynamic typing schemes for various C++ libraries share a common trait: Given a C++ object belonging to a dynamically-typed class, it is possible to obtain a unique string naming the class. Sometimes it is necessary to go through a class object in order to obtain the string, but the important point is that the string is obtainable. It is thus possible to write a small procedure for each such dynamic typing system which takes a C++ object and returns the name of its class. By defining this function as having `extern "C"` linkage in C++ and using the foreign function interface in TALK, we can thus provide a TALK typing function for each C++ dynamic typing system. It is then straightforward to produce a table mapping from the class name returned by C++ for an object to a TALK class corresponding to the C++ class.

For example, in the NIH class library, such a function would have the following definition in C++ and TALK:

```
// In C++
extern "C" {
const char const* NihTypeOf(Object* obj)
  { return obj->className(); }
}

;;; In Talk
(defextern NihTypeOf (ptr) ptr)
```

The foreign function interface system must know how to type the arguments and return value of foreign functions, as well as the slots of C++ classes. Therefore, the declaration of a new C++ dynamic typing scheme in TALK involves adding a new argument/return value converter to the foreign function interface system. For the NIH class library, this declaration would be:

```
(define-cpp-interface NihObject typer: NihTypeOf)
```

As demonstrated in the example above, the interface `NihObject` could then be used in C++ foreign function and class declarations to indicate that an argument, return value, or slot is a pointer to an NIH object[1].

The previous discussion applies to C++ libraries which implement some sort of dynamic typing. There remains the case of libraries which do not support such typing for some

[1] For arguments and return values which are not pointers, see the discussion below on passing by reference.

or all of their classes. We would like to make these classes too accessible in TALK, even if it means functioning in a degraded mode. The TALK/C++ interface provides two solutions for such cases.

1. If the library source code can be modified by the user, we provide a C++ base class and declaration macro which implement a cheap dynamic typing system. The cost is one virtual function, and the need to put the declaration macro in the definition of each C++ class which is to be accessible from TALK.

2. It is possible to use the TALK/C++ interface with a purely statically typed system. In this case, foreign function arguments, return types, and data member types must be declared with the exact TALK class these values will receive in TALK. If the corresponding C++ classes form a hierarchy, this will often mean that C++ objects seen in TALK will only be known to TALK as an instance of a base class. This is because, in this case, TALK has no way of determining the exact class of the C++ object, and must assign the object a prespecified TALK class. If the C++ classes do not form a hierarchy, the statically typed mode is in fact nearly as good as the dynamically typed mode; the only disadvantage is that C++ objects are blindly assigned their TALK types and so debugging becomes somewhat harder in case of mismatch between formal parameter type and actual argument type.

3.3 Representation

C++ objects have a hidden representation. C++ compilers are relatively free to use memory resources in whatever way is most appropriate for a given class[2]. This is why we need to generate stub functions which provide a portable if slightly expensive means of accessing objects.

TALK objects are allocated by the TALK memory manager, and have a special layout in memory which allows the memory manager to recognize them as TALK objects. We therefore choose to allocate a handle object for each C++ object encountered by TALK. This handle is a proxy TALK object created with a single field pointing to the actual C++ object. When the C++ object is passed from C++ to TALK, TALK looks up the pointer and its type in a weak hash table of handles. If no existing handle is found for the object, one is created with the appropriate type and stored in the table. The type of the handle is determined using the typing procedure defined for the interface specified for the argument or member involved in the transfer from C++ to TALK. When the object is passed from TALK to C++, the handle is dereferenced and the actual C++ pointer is passed to the C++ function. When calling between TALK functions, the handle is passed directly.

The handle approach does not allow TALK's garbage collector to safely collect C++ objects. Since C++ allows pointers to the interior of objects and pointer mangling, TALK cannot safely deallocate objects allocated by C++. Additionally, C++ destructor functions may not necessarily be called at arbitrary times. Therefore, just as in a C++ program, the deallocation of C++ objects must be managed by the programmer or by C++'s stack management[3]. Handles, on the other hand, can be collected. Collecting a handle does not delete its associated C++ object. It is therefore possible to have two different handles for the same C++ object, but not at the same time.

In any case, the TALK programmer sees what are apparently full-fledged TALK objects. All object-based tools function with these objects: inspectors, editors, class browsers, and so on work without modification. The TALK metaobject protocol provides a transparent reflective layer insulating the tool developer from different object representations. Because the generic function mechanism is completely separate from object representation issues, methods can be defined in TALK for C++ objects. The inheritance of the mirror TALK classes follows that of C++.

Some C++ types are not represented by handles. Integral types are represented by TALK small integers and bignums, and enums are represented by symbols with a special foreign type (defined by `defenum`). The exact treatment of these objects is outside the scope of this paper.

3.4 Argument Passing Style

Although C++ provides it, most C++ programs do not pass objects directly with call-by-value since the copying involved may be quite expensive. The TALK/C++ interface does not support call-by-value with objects because objects passed this way do not retain their identity over time, and, in any case, this style is little-used. However, the interface does support both call-by-value for pointers to objects and call-by-reference.

Like most other LISPs, TALK itself supports only call-by-value for object pointers and some immediate values. There is no notion of call-by-reference. Therefore, all C++ objects which are seen by TALK are seen as pointers. This means that call-by-reference arguments and return types must be referenced and dereferenced appropriately. This is done in the generated stub function. For example, the following C++ foreign function declaration, in which the single argument is passed by reference, generates the stub function shown below:

```
// C++ call-by-reference function
Color* call_by_ref(Point& r) { return r.Color(); }

;;; Generated Talk declaration
(define-cpp-type <Point>&
  talk-class: <Point>
  cpp-type: "Point"
  style: &)

(define-cpp-type <Color>*
  talk-class: <Color>
  cpp-type: "Color"
  style: *)

(defoverloaded call_by_ref (<Point>&) <Color>*)

/* Generated stub */
extern "C" {
```

[2] Although there are some restrictions on data member ordering, these are not sufficient to provide portable low-level access to C++ objects

[3] However, it is possible to imagine collecting objects when the programmer explicitly gives Talk the right to do so, and when the objects are allocated by the Talk interface to the C++ constructor.

```
extern Color *Ilt_call_by_ref(Point *v0)
  { return call_by_ref(*v0); }
}
```

The ampersand character in the `style:` keyword for the type definition for `<Point>&` indicates that arguments declaring this type use call-by-reference, and so the stub generator knows to generate the dereferencing operator in the stub. When the TALK function `call_by_ref` is called with an instance of `<Point>` (always a pointer to such an instance in TALK), the pointer is passed to the stub `Ilt_call_by_ref`, which dereferences the pointer and calls the original C++ function `call_by_ref`.

3.5 Signature Ambiguity

There are a number of cases where it is possible to overload a C++ function with signatures that are ambiguous for TALK, because TALK, having a much simpler object and typing model than C++, simply cannot reproduce dynamically all of the information necessary for distinguishing between overloaded function signatures. In all of these cases, the module parser can either present the user with a choice, or take a default resolution. Here are some of the ambiguous cases:

- C++ functions may be overloaded with both pointer and reference signature for the same class. TALK proxy objects are always pointers; there is no dynamic notion of a reference. By default, the pointer version is therefore preferred.
- C++ distinguishes between `const` and non-`const` values and pointers. TALK does not make such a distinction. When a C++ function is overloaded on both `const` and non-`const` types, the module parser prefers the `const` version because it is safer.
- C++ has a multitude of numeric types, each of which can be further modified by signedness. TALK has only characters, small integers (30 or 61 bits), bignums, and boxed doubles. When a C++ function is overloaded on several integral types, the parser prefers the `signed long` version since it most closely matches a native TALK type. For float types, `double` is preferred since it most closely matches a native TALK type. Other numeric types are ordered for preference by the size of their range, always preferring the signed version above the unsigned version.

These heuristics are not always sufficient for the parser to choose a best signature. For example, consider the following declaration:

```
void bar(long, short);
void bar(short, long);
```

In this and similar cases, the parser merely chooses the first signature and signals a warning. The user can modify the parser's choice if the other signature is more useful.

3.6 Callbacks

Many C++ libraries use callbacks in order to allow the programmer to specify action to be taken in certain situations. Callbacks correspond to the LISP notion of closures or functional objects, although, unlike LISP functions, they may not close over variables. To provide a complete interface to a C++ library, we must be able to program such callback procedures in TALK. Exported TALK functions have a different calling protocol than C++ functions; it is not sufficient merely to pass a pointer to the TALK function's code.

In particular, TALK functions expect two extra arguments: The functional object itself, which in TALK contains the function's captured lexical environment, and the number of arguments to allow the function to dynamically test the number of arguments and implement optional and multiple arguments. We therefore arrange to pass an intermediate stub C++ function as the callback function. This stub takes the arguments expected by the callback's prototype, and then calls the TALK function passing the functional object and correct number of arguments. In TALK, the arguments to the callback must be handled correctly if they are foreign data. When calling a C++ function which accepts a callback argument, the programmer must pass a pointer to the stub function. To generate a stub function for a TALK function, the defining form `defcallback` must be used instead of `defun`.

3.7 Operators

While operators are called differently than functions in C++. they are fundamentally functions. The TALK/C++ interface provides access to them via their C++ function names, in which the word `operator` is prefixed to the operator name. For example, the operator + can be accessed in TALK using the function `operator+`. Since many of these operators correspond to TALK generic arithmetic operations, methods can be defined for the TALK proxy classes on these operations which simplifies the use of the operators. For example, given an overloaded C++ definition for + for a class `MyClass`. the following TALK code will access this definition using the TALK function +:

```
;;; Generated code
(define-cpp-class <MyClass> ...)

(defoverloaded operator+ (<MyClass>& <MyClass>&)
   <MyClass>&)

;;; Handwritten code
(defmethod binary+ ((arg1 <MyClass>)
                    (arg2 <MyClass>))
  (operator+ arg1 arg2))

(+ (MyClass) (MyClass))
```

4 Future Work and Current Status

Almost all major LISP implementations include a foreign function interface. The TALK/C++ interface goes much further than these interfaces because it automatically provides nearly complete access to a C++ library at both the function and data levels. Perhaps the closest approach is that of esh, described in [8], which generates a close binding between a C library and a Scheme implementation. However, the problem is considerably more difficult for C++ because of the object-oriented nature of C++ and the not inconsiderable syntactic and semantic complexity of C++ compared to C.

Currently, the C++ connection does not handle templates or multiple inheritance. Additionally, exception han-

dling works rather poorly when LISP function calls are interlaced with C++ function calls.

The TALK/C++ interface is included with the commercial product ILOG TALK 3.0. It has been used to automatically generate an interface to ILOG DBLINK, a portable database interface, and ILOG VIEWS, a complete graphics library with over 300 classes and 5000 method signatures.

5 Conclusion

LISP and C++ both have strengths and weaknesses. What C++ gains in efficiency and static checking, it loses in flexibility and programmer productivity; with LISP, the situation is inverted. Both languages have their place in a large application, and it should be possible to use both effectively. In addition, as dynamic languages gain in popularity, it will be necessary to deal with a certain amount of legacy code. It would be excessive to expect C++ to provide an interface to the various LISPs. Therefore, it is up to the LISP community to find solutions. The TALK/C++ connection provides a non-intrusive encapsulation for C++ libraries, giving the TALK programmer complete access to the functionality of a C++ library in the comfort of a LISP environment.

References

[1] Bretthauer, H., Kopp, J., Davis, H.E., and Playford, K.J. Balancing the EuLisp Metaobject Protocol. *Lisp and Symbolic Computation*, 6(1/2):119–138, 1993.

[2] WG21/N0355 Standards Committee, editor. *Working Paper for Draft Proposed International Standard for Information Systems — Programming Language C++*. International Standards Organization, 1993.

[3] Harley Davis, Pierre Parquier, and Nitsan Séniak. Talking about Modules and Delivery. In *ACM Conference on Lisp and Functional Programming*. ACM SIGPLAN, ACM Press, 1994.

[4] Keith E. Gorlen, Sanford M. Orlow, and Perry S. Plexico. *Data Abstraction and Object-Oriented Programming in C++*. John Wiley and Sons, 1990.

[5] Barry Hayes. Finalization in the Collector Interface. In *International Workshop on Memory Management*, pages 277–298. ACM SIGPLAN, Springer-Verlag, 1992.

[6] Ilog. *Ilog Talk Reference Manual*, 1994.

[7] Gregor Kiczales, Jim de Rivières, and Daniel Bobrow. *The Art of the Metaobject Protocol*. MIT Press, 1991.

[8] John R. Rose and Hans Muller. Integrating the Scheme and C Languages. In *ACM Conference on Lisp and Functional Programming*, pages 247–259. ACM SIGPLAN, ACM Press, 1992.

[9] Harlan Sexton. Foreign Functions and Common Lisp. *Lisp Pointers*, 1(5):11–26, 1988.