1 PROGRAMMING LANGUAGE INTEROPERABILITY IN DISTRIBUTED COMPUTING ENVIRONMENTS

H.–Arno Jacobsen

Institute of Information Systems
Humboldt University, Berlin
Spandauerstr. 1
D–10178 Berlin

jacobsen@wiwi.hu-berlin.de

Abstract: Distributed computing environments, such as ANSAware, CORBA, DCOM, and DCE achieve software component interoperability through specifying all public component interfaces in a common interface definition language. Programming language interoperability is achieved by mapping these interface specifications to the programming language of choice. The language mapping defines the representation of an a priori agreed upon set of data types in the target programming language. The developer is restricted to this convention. This is, however, just the first step in achieving programming language interoperability. By focusing on the CORBA standard we illustrate this mapping procedure. We develop and thoroughly analyze several design schemas for implementing cross–language method invocation with CORBA. From this discussion it will become clear that the current standard lacks support for *user extensible programming language interoperability*. The CORBA standard does not allow to efficiently and effectively interface arbitrary programming languages to CORBA middleware. Point-solutions are possible, however, portable implementations of language mappings are *not*. We propose an addition to the current standard which would alleviate this problem and provide sufficient interfaces for other ORB extensions, such as request monitoring, debugging hooks, and costum marshalling, among others.

Keywords: Programming language interoperability, language mappings, IDL–to–Sather language mapping, middleware, distributed computing infrastructure, CORBA.

1.1 INTRODUCTION

Several distributed computing infrastructures have come to age over the past few years. Examples include ANSA [1], DCE [13], OLE/COM and DCOM [3], TINA [16], CORBA [11], JavaRMI, and JavaBeans [4], as well as research prototypes, e.g., ILU [9] and Hector [2]. The key objective of these infrastructures is to leverage the heterogeneity inherent to large distributed systems and to provide a middleware insulating the distributed application from various details of the underlying computational resources.

With implementation techniques, programming languages, and programming paradigms steadily evolving, on the one hand, and the need to implement different components of the distributed application with the language and paradigm best suited for the task, on the other hand, a problem of *programming language and paradigm interoperability* emerges.

The integration of legacy code into new applications often entails similar interoperability issues, especially if the legacy code is written in a language differing from the current development language, or if the legacy code is not even fully available as source. Language interoperability becomes also important when trying to interface specialized proprietary languages, such as, for example, languages for particular business and telecommunications applications, scripting languages, domain specific development language, and research languages to the middleware platform.

Most of the enumerated distributed computing infrastructures intend to solve this interoperability problem. Language interoperability is achieved through providing a common *interface definition language* (IDL) and a *language mapping* to a set of platform–supported implementation languages. The mapping defines the representation and manipulation of a common set of data types with the means of the target programming language. This is the approach taken by ANSA, CORBA, DCE, DCOM, and TINA. We have, however, found that beyond this set of platform–supported language mappings, an extension to another programming language gives rise to major technical problems.

In this work we focus our attention on CORBA's support for programming language interoperability to better illustrate the technical issues. Similar problems arise for other platforms. The solutions we propose do conceptually carry to these platforms as well. In particular we demonstrate that CORBA dose not offer appropriate support for third parties to implement language mappings in an efficient and effective manner. Moreover, it is not possible to interface arbitrary programming languages to CORBA middleware in an inter–ORB portable manner (i.e., port from one ORB implementation to another). This is mainly due to the lack of a *portability interfaces* at the ORB–level, not mandated by the platform standard.

The often adopted solution to this dilemma is the implementation of a proper ORB (Object Request Broker) for the particular target language and pass cross–language invocations via the standardized Inter–ORB protocol to foreign language objects. OLIVETTI AND ORACLE RESEARCH LABS [10], for instance, have taken this route, with OMNIORB, to obtain a language interface to PL/SQL. A similar undertaking is pursued by FRANZ INC. to obtain an ORB language interface for Lisp[1]. This solution is a time consuming process and is not always an option for other interoperability seeking institutions. Alternatively, point–solutions are available from middleware vendors and from the research community for interfacing, scripting languages, for instance, to particular platforms (see MICO [14], and OMNIBROKER [12]). This solution is a valid option, but does not provide an inter–ORB portable approach and may only be pursued if the appropriate interfaces are available.

In this paper we exemplify the steps necessary to interface a programming language to the CORBA platform (Section 1.3). We develop and thoroughly analyze design schemas for implementing language mappings (Section 1.4). This analysis demonstrates that with the CORBA standard *user extensible programming language interoperability* is not universally possible. To fully overcome this problem we propos additions to the standard which manifest themselves in a set of *portability interfaces* that allow direct access to the ORB (Section 1.5). We sketch these interfaces and motivate their use for alternative ORB extensions. The following section reviews the key concepts of the CORBA architecture that are necessary for the further understanding of our work.

1.2 CORBA AND INTEROPERABILITY

The Common Object Request Broker Architecture (CORBA) is a standard for distributed computing which has been developed by the Object Management Group (OMG) [11], a consortium of independent companies. CORBA aims at providing a uniform communication infrastructure for building distributed applications. It supplies unifying mechanisms for interoperating software components, operating on various hardware platforms, and running under different operating systems. CORBA has also been designed to support programming language interoperability [15]. This is to allow for full flexibility in application design and development, as well as to facilitate the integration of legacy systems and legacy code into distributed applications [11]. Figure 1.1 gives an overview of the platform architecture.

Interoperability is gained by specifying all component interfaces in a universal interface definition language (IDL). IDL is a descriptive, non–algorithmic "lingua–franca" for specifying interfaces, following a C++–like syntax. Interface specifications are mapped, by a stub–compiler, to stubs in the component's programming language. These stubs are compiled and linked with the component code and with the infrastructure implementing libraries.[2] For distributed applications the stubs handle communication with the remote machine via the Object Request Broker (ORB) and perform argument packaging, marshalling, and unmarshalling.

The *Internet Inter–ORB Protocol* (IIOP) is a specific incarnation of the *General Inter–ORB Protocol* (GIOP), mapping GIOP onto TCP/IP. It is part of CORBA and

[1] FRANZ INC. — Alegro CL ORB.

[2] In OMG terminology stubs serve as object implementation proxies in the client address space, whereas skeletons serve to interface to object implementations in the servant address space. Whenever unambiguously possibly we use stubs as placeholder for both stubs and skeletons.

also standardized by the OMG [11], in an effort to achieve interoperability among different object request brokers. To be CORBA compliant an ORB must support this protocol. The ORB interoperability protocols were primarily designed for communication in distributed heterogeneous environments, (i.e., across machine and request broker boundaries). A considerable amount of computation is necessary to transform a client request and server response from a particular machine representation into the defined format and possibly back into another representation on the request's target side. In the worst case four of these transforms need to occur (e.g., two for the request and two for the reply on client and server side respectively.)

1.3 DEFINING A LANGUAGE MAPPING

1.3.1 Language mappings

An OMG IDL language mapping defines the representation of IDL statements and data types in terms of constructs and types of the programming language under consideration. This amounts to defining representations for the following constructs in the target language: literals and identifiers, basic and constructed data types, modules and interfaces, setting and retrieving of attribute values, operation signatures and parameter passing conventions, and raising and handling of exceptions.

Mappings for Ada, C, C++, COBOL, Java, and SmallTalk have been standardized. Standardization of mappings for Lisp, Eiffel, and a to be determined scripting language are in progress. Mappings for Sather [7], Modula–3 [6], JavaScript, TCL, Perl, Python, and CORBAScript have independently been proposed[3]. Although many mappings exist, it is not always clear whether the proposing institutions have supporting implementations. We attribute this to the *user extensibility problem* motivated in Section 1.1.

1.3.2 Technical issues of the language mapping design

To add a new programming language to the languages already supported by the CORBA specification, it suffices to specify an IDL–to–"*language*" mapping, where "*language*" is the language to be added. For the language mapping implementer, however, the following tasks remain to be resolved: (1) Development of a stub–compiler which maps the IDL interface specifications to stubs in the adopted programming language. (2) Interface the stubs with the proprietary middleware platform interfaces. (3) The former step often entails more subtle interoperability questions, namely, how to interoperate the newly adopted programming language with the libraries of the CORBA implementation (i.e., generated stubs need to link to CORBA implementation code, which might both be written in different languages, following different parameter passing conventions, et cetera. This constitutes the actual implementation of the more abstract notion of language interoperability at the IDL/CORBA level.)

[3]Some of these efforts are taking place in response to the OMG RFP on support for scripting languages in CORBA.

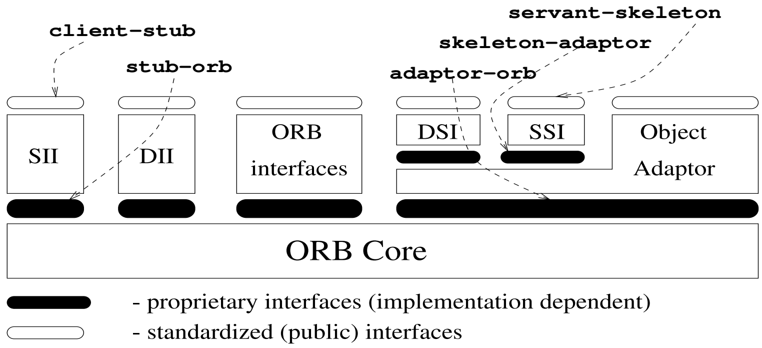

Figure 1.1 CORBA architecture with standardized and proprietary interfaces outlined.

From a software engineering point of view, these problems are all simple and well understood. This, however, presumes that necessary ORB interfaces are openly available, or, at least, presumes the source code of the CORBA system is at hand. For most CORBA implementations this is (unfortunately) not the case.

The "*client–stub*" interface of the stub is strictly defined through the respective OMG IDL language mapping. However, the CORBA standard does not address the interface of the stub–to–ORB interaction. Similarly, on the object implementation side, the "*servant–skeleton*" interface is defined in the language mappings, whereas the "*skeleton–adaptor*" and "*adaptor–ORB*" interfaces are realized in a proprietary manner. (Figure 1.1 illustrates these interfaces.) It is the availability of these proprietary interface on client, as well as, object implementation side, which is crucial for solving the above mentioned problem.

So far it has been the practice to develop CORBA implementations hand–in–hand with the stub compiler, supporting a small set of languages only. The issue of providing an open and extensible infrastructure remains neglected. In [8] we showed that most ORBs exclusively support either C++, or Java. Furthermore, most ORBs offer a single language interface only [8]. Alternate language interfaces are provided through the implementation of a proper ORB in the desired programming language, e.g., IONA supports a C++, ADA, SmallTalk, Cobol, and Java ORB [8]. Cross–language method invocation is then realized via IIOP among the different ORBs. We don't think that this is a good solution, due to the additional transformation overhead introduced. Furthermore, implementing separate ORBs for all kinds of languages will certainly not happen in the near future, especially for proprietary languages, such that programming language interoperability with CORBA remains still a goal to be achieved.

CORBA products are commonly shipped as binaries. Third party extensions are therefore limited to the standardized CORBA interfaces. As will be outlined in Section 1.4 these interfaces are not sufficient to allow for portable and efficient third party implementation of programming language mappings. We consider this non–user extensibility of the programming language interface of the CORBA architecture a weakness and hindrance for flexible distributed system design. It is a feature a CORBA implementation could easily posses, since all the necessary functionality is present in the ORB implementation. It simply needs to be revealed through an open standardized interface. This non–extensibility is a drawback which certainly prevents many users from using CORBA for their distributed application designs. For one, proprietary languages, especially scripting languages, are hard to interface to existing CORBA middleware for the reasons outlined above. Secondly, other, more specialized programming languages, such as, for example, persistent programming languages and business or telecommunications modeling languages, could greatly add to CORBA's character, but are subject to the same problems.

1.3.3 IDL–to–Sather language mapping

Sather is an object oriented programming language that has been developed at the International Computer Science Institute in Berkeley. Details may be found in the language specification[4]. The mapping of OMG IDL–to–Sather has been defined according to the requirements for language mappings outlined above. The defined Sather language interface constitutes a complete mapping of OMG IDL to Sather and is fully CORBA compliant. The exact definition and examples of the mapping may be found in [7]. The next section analysis different design schemes for implementing language mappings.

1.4 FACETS OF LANGUAGE INTEROPERABILITY WITH CORBA

1.4.1 Pseudo–stub based language transparency

This approach is currently prototyped for a commercial ORB. Due to the proprietary nature of the ORB and the non–availability of the stub–to–ORB interfaces in the standard is it difficult to directly access the ORB's communication layer. To obtain a portable solution (inter–ORB portability) we interface Sather "*pseudo–stubs*" to the C++–stubs generated from the vendor's IDL compiler. "Pseudo–stubs" are adaptors which pass calls from Sather objects to the corresponding method in the proprietary stub (cf. Figure 1.2). This is transparent to the implementer of the Sather–CORBA object. Note, that the C++–stub interface is standardized by the corresponding language mapping. In our case the IDL–to–C++ mapping. Except for the server–side code, this solution is portable. On the server–side the CORBA standard leaves room for proprietary design decisions by the ORB developer. The newly adopted, but, so far, not broadly implemented portable object adaptor (POA) alleviates these non–portability issues on the server side. Its availability will render this solution fully portable.Note, however, that the POA does not address the portability interfaces crucial for implementing a language mapping (cf. Figure 1.1).

The interface of a Sather pseudo–stub to a generated C++ stub is achieved via Sather's support for foreign language classes. In Sather an "`external C class`"

[4] `www.icsi.berkeley.edu/~sather`.

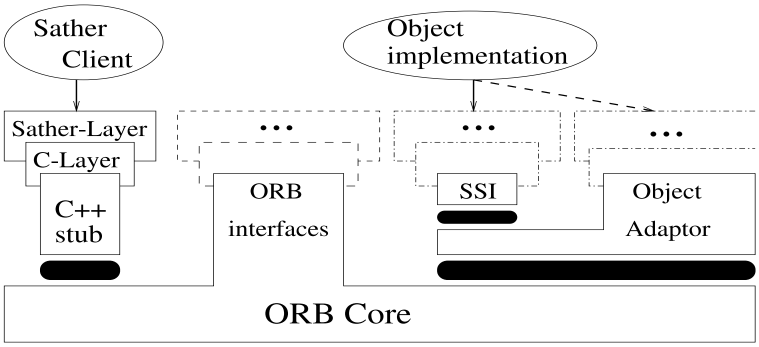

Figure 1.2 Pseudo–stub based approach to programming language transparency with CORBA.

allows for a bidirectional calling and exchange of data objects between the two languages. A thin C layer is provided to access the stubs and ORB interfaces, from within Sather Figure 1.3 and Figure 1.2 illustrate this approach in greater detail.

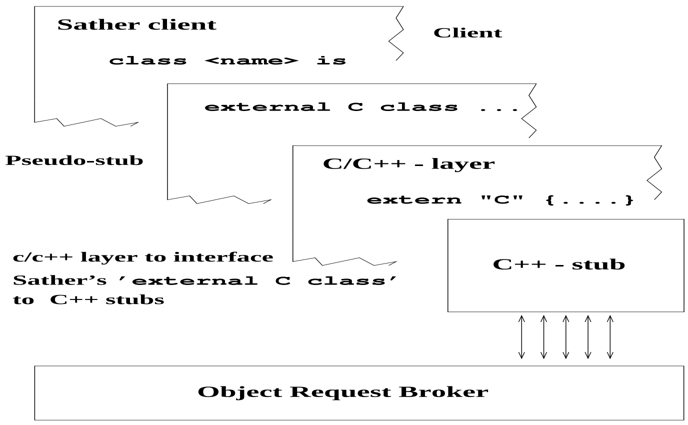

Figure 1.3 Implementation level interoperability to CORBA component.

The key advantage of this approach is that it is based on the mostly standardized parts of an already implemented language mapping, namely the C++ mapping, supported by most ORBs. Clearly, in terms of optimizing calling sequences this leaves much to be desired, since no influence can be exerted on the way the "actual" C++ stubs are emitted by the vendor's stub compiler. Performance may therefore be poor. We are currently quantifying this overhead. We also investigate the portability of this design to other ORBs. Furthermore, the advantages implicit in the IDL–to–Sather mapping, due to garbage collection in Sather, can in this design not be fully exploited, since the layer of

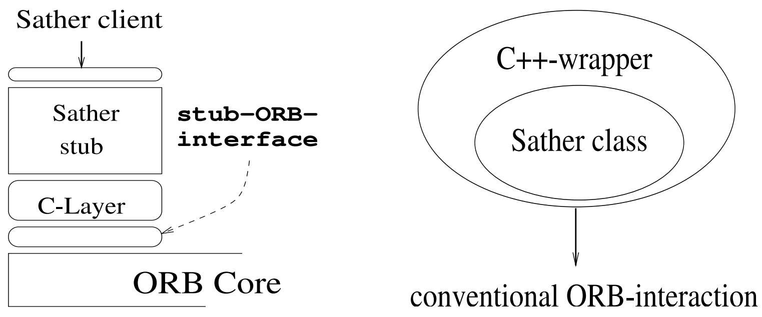

Figure 1.4 (1) Completely Integrated approach to programming language transparency with CORBA. (2) Foreign language adaptor based approach to language transparency.

C++ objects (vendor stubs) around the ORB need to be taken care of "manually", (i.e., in a manner similar to that defined by the C++ mapping.) This design may be applied to interface any programming language, that offers a foreign language interface, to a middleware platform.

1.4.2 Completely integrated language interface

This approach is currently prototyped for the public domain CORBA implementation, MICO [14], available as source code. The language interface is completely integrated into the broker architecture, just like the languages already supported by the implementation. MICO, fully supports C++. To also support the Sather language mapping we are providing a Sather interface to all interfaces of the request broker, as illustrated in Figure 1.4. For a Sather CORBA object the broker is therefore undistinguishable from an ORB written in Sather. The exact low–level implementation of the Sather/C++ language interoperability has already been discussed in the previous section (cf. Figure 1.3 for details.)

The individual tasks identified above are straight forward to realize due to the openness of MICO (mainly due to the availability of its source code). Note, however, that we do not need any internal knowledge of the implementation, open interfaces, specified in IDL, of the key APIs would suffice (cf. Section 1.5 for further thoughts on this issue.) Figure 1.4 illustrates this design.

The additional layers of Sather and C code, that make out the language interface, are thin. They corresponds to one function call in each layer to establish the language interoperability. Moreover, ORB internal optimizations for local, co–local, and remote invocations also apply to Sather CORBA object invocations. The achievable performance is therefore directly comparable to native language binding implementations. Another advantage is the possibility to provide Sather–native stubs and skeletons. Marshalling and unmarshalling may thus be implemented directly in Sather. This provides also a rich framework for further experimentation. The key drawback of this design is that it is particular to one ORB and not portable to other brokers. This design is not particular to the Sather language and the MICO broker implementation but generalizes to other languages and brokers.

1.4.3 Language interoperability through foreign language adaptors

The previous approaches encapsulate the functionality of the CORBA implementation with the language to interoperate with. This approach operates the other way around by encapsulating the target language objects with language adaptors written in the language already supported by the request broker. In this solution the extensibility problem is projected entirely onto the availability to integrate a foreign language into the languages supported by the request broker. It is therefore directly applicable to languages like C, Ada95, Sather, and Theta, for instance. The resulting encapsulated CORBA objects are portable to other broker environments, since no ORB specifics need to be exploited to make this approach work. Clearly, this approach is rather work intensive, since the CORBA object adaptors need to be hand coded for all components to be used in the distributed application. This approach is particularly well suited for interfacing legacy code to broker environments, as long as the language interoperability issue between legacy code and supported language can be resolved. Figure 1.4 depicts this design.

1.4.4 Achieving programming language interoperability via IIOP

Language transparency may also be obtained by inter–connecting different ORBs, which support the programming languages to interoperate via the Internet Inter ORB Protocol (IIOP), thus passing cross–language invocations from ORB to ORB. Conceptually simple, the problem is that most ORBs support the same set of languages, such that not too many cross–language invocation combinations are feasible [8]. Moreover, the additional format conversions involved in the IIOP protocol decrease the performance of the resulting application. Optimizations based on whether or not the individual components are local, co–local, or remote will hardly be possible, unless the involved ORBs were designed with this particular interconnection in mind. This is not to be expected for brokers from different vendors. This solution can therefore only be recommended for performance uncritical tasks. Nevertheless, have we used it in internal projects to interoperate C++ and Java components. Another problem with this solution is the support for proprietary languages which is not possible with this design, unless a CORBA compliant ORB supports the language under consideration.

Of course, it is always possible to implement a proper request broker and have it communicate with CORBA compliant systems via IIOP. An entire ORB implementation is certainly very resource consuming and is therefore not a good choice for all possible scenarios, such that other proprietary solutions will have to be taken.

A "thin" CORBA implementation, however, is a first step in this direction. By "thin" we mean an implementation which, either provides an extensible sub–set of the functionality provided by full CORBA, or supports client–side or object–implementation–side interoperability only. Such a solution would simply support the CORBA IIOP–

interface and allow to send or receive requests in the defined format. Subject to the same considerations as the here discussed solutions, its implementation cost is not as high.

1.4.5 A solution based on the dynamic invocation and skeleton interface

The dynamic invocation interface (DII) and the dynamic skeleton interface (DSI) are intended for use by components which do not have compile time knowledge of each others' interfaces. The DII and the DSI may therefore be used to generate and accept requests for operations at runtime. These interfaces are also the only direct access to the request brokers communication layer. Static invocations pass through generated stubs which interface to vendor specific proprietary broker APIs.

All fully CORBA compliant ORBs must support these interfaces. It is therefore safe to use them for implementing programming language interoperability. Like all the other ORB interfaces the DII and DSI are specified in IDL and published in the CORBA specification. Their entire functionality is thus exposed and may be used by the application designer and the language mapping implementer. We suggest using them to implement language interoperability, not at runtime, but at application development time to pass and receive requests between different programming language objects whose interfaces are available in IDL.

However, the subtle issues of how exactly to implement the programming language interoperability at the linkage level is not as yet solved, i.e., the foreign language still needs to call on the provided ORB functions (i.e., DII/DSI operations) and pass its arguments in a correct manner. Note, the CORBA standard requires the CORBA implementation to reveal language, syntax, and signature of the DII/DSI implementing functions in the languages supported as mappings. This is captured in the language mapping by *pseudo objects* representing the interfaces in the syntax of the supported languages. The exact calling code may then be generated by a support tool, given the IDL interface of the involved components.

The great disadvantage of this approach is its performance. It has been shown experimentally [5], that the DII/DSI interfaces, in current CORBA implementations, are much slower than the corresponding static interfaces. We are not using the dynamic features of these APIs, but rather exploit the fact that the DII/DSI provide a direct interface to the broker's lower level communication layer.

The key advantage of this solutions is that it is universally applicable across all CORBA compliant ORBs. It is therefore, the only truly portable solution. We are currently testing its robustness, effectiveness, efficiency, and portability[5] across different ORBs.

[5]Minor design decisions need still be taken by the ORB implementer for realizing the DII/DSI. These issues are expected to become standardized in future revisions of the CORBA standard.

1.4.6 C–language binding based

Many programming languages provide C interface. CORBA specifies a language binding for C. Interoperability could thus be achieved through integration based on the programming language's C interface and the ORB's language binding for C, if available. This solution is closely related to some of the above approaches and therefore not further discussed. However, the IDL–C binding is not widely supported, the major drawback of this approach [8].

1.4.7 Manually coded interface adaptors

In this solution the language mapping implementer provides a layer of interfaces on top of the proprietary ORB APIs revealing their syntax and operation signatures to the outside. A stub compiler can then generate code which calls against these functions. Clearly, such a solution cannot be recommended, since, unless the source code of the ORB implementation is available, a manual capture of all its functionality is rather difficult. This would have to be done by reverse engineering the generated stub code[6]. For one, the task is rather work intensive, but more importantly, may interact with potential copyrights protecting the proprietary code. Secondly, the solution is highly specialized to one product and requires the same investment of work for porting it to other ORBs. The single advantage is its performance which is directly proportional to the available mappings for other languages.

1.4.8 Comparison and evaluation

This section compares and evaluates the individual approaches according to the following criteria: n–to–n–language interoperability, portability of approach, independence, full language support, efficiency, and implementation effort.

With n–to–n–language interoperability we describe the potential of the approach to interoperate any language with any other language. Portability refers to the possibility to exchange ORBs, given an implementation of the approach. With independence we capture the feature of whether or not the approach requires support from a mapping already implemented for a given ORB (i.e., uses its stubs, etc.). Full language support describes the possibility to use the full set of features a languages offers, or whether some features must be "turned off" (e.g., garbage collection cannot always be supportively used in client and server code due to the interfacing to C++ (see discussion above).) Implementation effort describe whether or not the approach requires one time, per–ORB, per–application, or a combination of the former effort, to be implemented. Table 1.1 summarizes the evaluation according to these criteria. From the table it becomes clear that none of the approaches is consistently strong in all points. Choices of which one to adapt for a particular ORB–language combination has to be strongly usage driven, with the trade–offs identified in the table.

[6]Generated stub code is commonly in source language format. While reverse engineering it is difficult, it is not "impossible".

approach / feature	pseudo–stub	completely integrated	lang. adaptors	IIOP based	DII/ DSI	C–binding	manuelly
n–to–n	yes	no	no	no	yes	no	yes
portable	yes	no	yes	yes	yes	no	no
independent	no	yes	NA	no	yes	no	yes
full lang.	no	yes	yes	yes	yes	yes	yes
efficient	yes	yes	yes	no	no	yes	yes
impl. effort	one time	one time + per–ORB	per–app.	none	one time	one time	one time + per–ORB

Table 1.1 Evaluation of language interoperability solutions.

1.5 TOWARDS AN OPEN INTEROPERABILITY PLATFORM

In this section we propose an addition to the CORBA standard which solves the user extensibility problem. Furthermore, it provides for an extensible framework that gives rise to other extensions, as discussed in Section 1.3.

All of the above discussed solutions for realizing language interoperability with CORBA have shown considerable drawbacks, either in terms of expected performance, or in terms of portability and extensibility of the approach. Unless language interoperability is supported by the vendor, it is not possible to more effectively implement it with state–of–practice CORBA implementations, due to lack in the standard to mandate openness at the ORB–level.

To achieve user extensible the CORBA standard should be augmented with an open portability interface, presented as pseudo object. The interface must reveal the ORB's bare communication operations. It should be simple, just revealing enough functionality for a third party to extend and use the ORB. The intend is to bypassing the SII/SSI and not use the less efficient DII/DSI.

It suffices to provide operations for sending and receiving of messages in a predefined data format, i.e., reading and writing to the network. Packaging, marshalling, and unmarshalling are entirely in the responsibility of the client of this interface. Such an interface is inherent, in one way or another, in all ORB implementations, such that no modifications to existing systems are necessary. In Figure 1.1 we denoted it as "proprietary stub–to–ORB interface". Figure 1.3 depicts the CORBA architecture conceptually, and shows which interfaces must be additionally standardized to introduce extensibility and openness.

Opening interfaces in the CORBA standard that would enable third party extensions are not popular among ORB vendors, who fear to lose market shares by providing a *too* open architecture. A move towards a similar extension of CORBA from within the

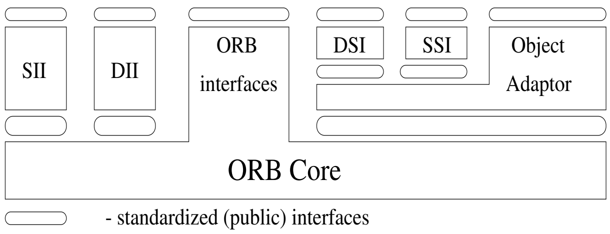

Figure 1.5 Corba architecture with additional open interfaces to obtain a more extensible design.

telecom domain, asking for means to plug different communication protocols into the ORB[7], has failed to pass through the OMG standardization process.

The proposed addition would not only allow to better interoperate programming languages with CORBA, it would also allow to decouple the stub–compiler and all the, inside the stubs hidden, functions (i.e., stub and skeleton themselves, marshalling and unmarshalling, data–structures, and packaging and unpackaging, et cetera) from the core broker. This can only be of benefit, since these functions could then be implemented separately with different foci, e.g., high–performance, marshalling with respect to different protocols and data formats, et cetera.

Moreover, the extensibility gained through an open ORB architecture would enable independent implementations of the following features, currently hardly supported, even by ORB vendors themselves: CORBA application management support (i.e., through monitoring hooks), debugging support for distributed applications, performance measurement support, customized marshalling and application specific stubs and skeletons, (e.g., object cache on client side) access control, and universal language interoperability, as stressed throughout this work.

1.6 CONCLUSION

We have pointed out that user extensible programming language interoperability could play a crucial role in the further spread and acceptance of CORBA technology for parties interested in interfacing proprietary languages to CORBA middleware. At least conceptually, the integration of a new OMG IDL language mapping in the CORBA standard does not pose a problem, as manifested by the large number of IDL mapping specifications available to date. We concluded by arguing for stronger CORBA standard support to enhance programming language interoperability in future CORBA specifications. This support could be in form of an additional ORB pseudo object interface which gives the user direct access to the ORB's low level communication facilities and provides for access points for additional ORB extensions.

[7]The RFP on pluggable protocols propagated by the OMG telecom domain task force.

References

References

[1] ARCHITECTURE PROJECT MANAGEMENT. *The Advanced Network System Architecture (ANSA)*. Castle Hill, Cambridge, England, 1989.

[2] BOND, A., ARNOLD, D., AND CHILVERS, M. Designing and building an ODP environment. Tech. rep., CRC for Distributed systems Technology, 1996.

[3] BOX, D. Introducing Distributed COM and the new OLE features in Windows NT4.0. *Microsoft Systems Journal* (1996).

[4] ENGLANDER, R. *Developing Java Beans*. O'Reilly, 1997.

[5] GOKHALE, A., AND SCHMIDT, D. C. The performance of the CORBA dynamic invocation interface and dynamic skeleton interface over high-speed ATM networks. In *IEEE GLOBECOM '96* (Nov 1996).

[6] HURWITZ, B., AND NAYERI, F. IDL to Modula–3 language mapping. Tech. rep., GTE Laboratories, 1994.

[7] JACOBSEN, H.-A. Specification of the OMG-IDL to Sather mapping. Tech. rep., ICSI, 1996.

[8] JACOBSEN, H.-A. User extensible programming language interoperability with corba. In *CORBA Management Workshop* (Dublin, Ireland, September 1997), OMG.

[9] JANSSEN, B., AND SPREITZER, M. ILU Reference Manual. Tech. rep., Xerox Parc, 1997.

[10] OLIVETTI, AND ORACLE RESEARCH LABORATORY. omniORB 2.5 user documentation. www.orl.co.uk/omniORB/omniORB.html.

[11] OMG. The Common Object Request Broker Architecture and Specification. Revision 2.0. Tech. rep., Object Management Group, 1998.

[12] OOC. Omnibroker. http://www.ooc.com/ob.html.

[13] OPEN SOFTWARE FOUNDATION. OSF Distributed Computing Environment Rationale. Cambridge, MA, 1990.

[14] Römer, K., and Puder, A. MICO. http://www.vsb.cs.uni-frankfurt.de/ mico/.

[15] Siegel, J. *CORBA Fundamental and Programming*. John Wiley & Sons, Inc., 1996.

[16] TINA–C. DPE Phase 0.1 Specification. Tech. rep., Telecommunication Information Networking Architecture Consortium, 1993.