2nd IFIP WG 6.1 International Conference on Distributed Applications and Interoperable Systems (DAIS'99), Helsinki, Finland, June 1999

CAN CORBA SAVE A FRINGE LANGUAGE FROM BECOMING OBSOLETE?

S. Eisenbach, E. Lupu, K. Meidl, H. Rizkallah*

Department of Computing, Imperial College
180 Queen's Gate, London SW7 2BZ

E-mail: se@doc.ic.ac.uk

Abstract: Not only does CORBA offer the advantage of distribution transparency for building applications, but it may provide esoteric programming languages with greater capabilities through its interoperability standard. Implementing interoperability with the CORBA/IIOP gives rise to several problems of compatibility between the CORBA computational model and those of the languages or sub-systems for which an IIOP bridge implementation is built. This paper describes how an actor-based language, Rosette, has been extended to provide support for distributed environments by extending the language with support for the CORBA/IIOP. The prototype IIOP interface has been implemented as a half-bridge. The Rosette types, object model and concurrency are very different from those generally available in conventional languages and CORBA. We discuss the issues relating to type compatibility, run-time type identification, and multi-threading of concurrent invocations.

Keywords: Interoperability, Actor Model, CORBA, Half-bridge, IIOP, Rosette

1 INTRODUCTION

While languages like C++ or Java are more widely used, other languages come into their own for specific problems. C++ and Java are not ideal for scripting or prototype development of distributed applications, since they do not allow the programmer to interact dynamically with a concurrent system. Programmers may wish to evolve their design dynamically through trial-and-error experiments. While languages like Smalltalk [4] and Self [6] have been successful in providing a development platform for specific applications and supporting different concurrency models, they have

* now with Andersen Consulting, Arundel St., London

been all too often abandoned for interoperability reasons in heterogeneous environments. *The Common Object Request Broker Architecture (CORBA)* [5] allows not only distribution transparency in a distributed processing environment, but also interoperability between systems implemented in different programming languages. In this paper, we present an implementation of a system that allows programs written in an actor-based language, Rosette [3], to interact with the CORBA world through the use of the *Internet Inter-ORB Protocol (IIOP)*. The difficulty with this task lies in the fundamental differences between the concurrency and computational model of the Rosette language which is based on the *Actor Model* [1], and the more static, strongly-typed model assumed in IDL and object-oriented languages (such as C++ or Java) which are closer to the CORBA worldview.

The actor model is well suited for distributed applications with its reliance on asynchronous message passing as the basic means for communication. The Actor Model and its Rosette implementation offer a high degree of concurrency by considering that each entity within the system is an actor executing in parallel and independently from all the other actors. Thus, concurrent execution is very fine grained. Furthermore, after processing a message the actor may replace its behaviour. This permits additional flexibility in cases where the actor has to react to changes in its environment and must adapt its behaviour accordingly. The actor model is ideal for building multi-agent systems. But this computational model differs substantially from the object based paradigm adopted by CORBA and ensuring interoperability between CORBA and Rosette requires implementing a half-bridge able to translate between object invocations and the asynchronous message based communication adopted by Rosette. Furthermore, there are substantial differences between the Rosette type system and the CORBA IDL types. Therefore, the bridge must provide the functionality for performing conversions between the two type systems without compromising data integrity (e.g., truncation).

This paper describes a proof of concept prototype, Internet Inter-ORB extension for Rosette, which has been implemented as an ORB half-bridge. The main challenge in the creation of a Rosette/IIOP interface is the mapping of the CORBA object model onto the actor model and in particular type checking, data structures and system management. More general issues that have also been dealt with are how CORBA sees Rosette objects, how Rosette objects implement CORBA interfaces, how Rosette performs a remote request and how it handles an incoming remote request from a CORBA-compliant ORB. CORBA works on the basis of strong type checking while Rosette implements a system whereby method signatures are not types. Instead, Rosette offers a suite of predicates that return the type of an object. What the CORBA specification takes for granted requires explicit control in Rosette. Another problem involves CORBA data types, which have specified formats, and fields. Rosette uses a tuple data structure to represent and structure messages between actors. This makes it necessary to ensure that the correct tuple format is being received, a notion that goes under the category of type checking in the CORBA specification. The Rosette environment is dynamic, and its state changes over time. Within the actor model, a dedicated agent handles co-ordination. Messages sent between agents are intercepted by the co-ordination agent and forwarded according to the current behaviour of a configuration agent. The interception is completely transparent to the sender who does not know which agent

or agents have actually received it. Such a scheme permits dynamic changes to be made to the configuration and co-ordination of the distributed system.

Section 2 of this paper, provides background information on the Actor Model, and Rosette. This is followed by an in-depth description of the Rosette IIOP System in Section 3, which covers each of the system's components, the handling of invocations, marshalling and de-marshalling activities. Section 4 presents some of the related work on language interoperability and is followed by the conclusions.

2 THE ACTOR MODEL AND ITS ROSETTE IMPLEMENTATION

2.1 The Actor Model

The Actor model [1] is a simple, yet powerful means for defining agent-based systems. In this model, everything in a system is an actor. This is similar to the uniform approach of Smalltalk [4] where everything is an object, however with two important differences: (1) each active actor is completely independent of all other actors in the system; (2) all the actions taken by an actor upon receipt of a message are concurrent, i.e., there is no implicit serial ordering of the actions in a method.

An actor is an active entity that has one kind of event, communication, and one kind of activity, answering messages. It is an object that resides at an address, and is characterised by an identity and a current behaviour which determines how it will respond to the next message it receives. Once created, an actor's identity does not change, even though the way in which it behaves over time may. The identity corresponds to the address of a mailbox which buffers incoming messages until they can be processed. Thus, the basic form of interaction between actors is asynchronous buffered peer-to-peer communication, which reduces problems that could arise due to blocking. The message subsystem supports weak fairness: it guarantees delivery of all communications but makes no guarantees on the preservation of message order or delivery time (other than it is finite). Therefore, processing will occur according to the local order of messages, and no assumptions can be made about this.

Actors may be partitioned into primitive and non-primitive classes. Primitive actors correspond to atomic types in other languages, such as numbers and characters. Non-primitive actors have an identity represented by a reference and a current behaviour. The current behaviour is defined by local encapsulated state, data or knowledge base, and a script which defines how the actor will respond to the next message it receives. The local state data is analogous to instance variables, and the script to methods. Since the local state data is comprised of a set of actors with which communication takes place, they are called *acquaintances*. When an actor decides to accept a message, it responds to it with an answer or a side effect. An actor's behaviour is displayed through three fundamental capabilities:

1. **Communication:** An actor can send messages to itself and its acquaintances, and delegate subtasks to its acquaintances.
2. **Creation:** New actors can be created in order to delegate subtasks to them.
3. **Modification:** The actor *must* create a replacement behaviour, which governs its responses to the next message.

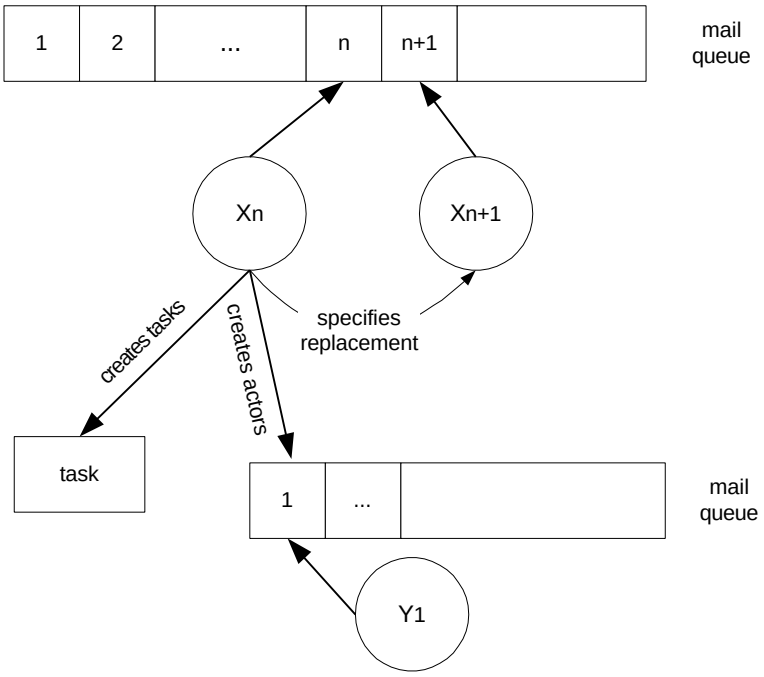

Figure 1 An abstract representation of a transition

Behaviour replacement (Figure 1) is the main feature of the actor model that distinguishes it from other object models, including those that support concurrency. Often the new behaviour corresponds to simple changes in the encapsulated state, for example, updating a table in a storage actor. However, the behaviour change may be more complex, such as generating a new script for the next message, sending communications to specific target actors or creating new actors. Figure 1 gives an abstract representation of what occurs when an actor processes a message from its mail queue [1]. When the actor processes the n^{th} communication, it determines the replacement behaviour that will process the $n+1^{th}$ communication. Changes made to the behaviour only come into effect on receipt of the next message. This ensures that changes will not affect the remaining processing of the current message, or any earlier threads that may still be executing. An actor is locked from the time when it begins processing a message until the new behaviour has been specified. During this period, it cannot process further messages, thus ensuring that threads do not interfere with one another.

2.2 Rosette

Rosette is a concurrent, object-oriented and dynamic language, designed and implemented as part of the *Carnot* research project by *Microelectronics and Computer Technology Corporation (MCC)* [3]. Rosette is based on an object-oriented extension of the actor model. It is an extensible, interpreted language with

its own virtual-machine architecture (similar to that of Java) [10] which permits it to be platform-independent, and in which concurrent, object-oriented programs may be written and run in a scaleable manner. Rosette is uniform in a similar way to the Smalltalk object model [4], and the syntax is based on a Lisp-like functional notation [9]. Rosette is a dynamic language, ideal for the rapid development and support of distributed computing applications, especially middleware. As an interpreted language, it supports late binding and automatic storage management. It is also highly extensible.

In Rosette, an actor is considered to be a collection of slots that may be viewed as key-value pairs [3]. When an actor is created, it inherits the slots, both data and methods, from a prototype on which it is based. The slot values, which are unique to the actor, are stored as part of it. Although Rosette does not enforce type restrictions, it does have a type system which is based on the inheritance hierarchy. On top of that it has constructed sum, product, subtype and complement types. It therefore provides the programmer with the ability to specify and restrict types dynamically.

The Rosette interpreter follows the actor model, dividing subtasks among new actors. Task delegation represents independent logical processes that may be mapped onto physical resources to achieve parallel execution. The execution model provides for distribution of the computational load across system resources without accounting for it in the design. All actions taken by an actor upon receipt of a message are concurrent. In particular, statements in a method body are executed concurrently unless otherwise specified. Therefore, computation is viewed as concurrent and extremely fine-grained. This style of programming creates multiple threads of execution that are scheduled by the Rosette core.

3 THE ROSETTE IIOP SUB-SYSTEM

In order to support the Internet Inter-Orb Protocol (IIOP) a half-bridge has been implemented in Rosette. This IIOP sub-system needs to support invocations which comply with the IIOP specification as given in [5]. Other CORBA Services [7] have also been implemented within the IIOP sub-system, however these are outside the scope of this paper. The sub-system is composed of six main actors (Figure 2) that perform marshalling of outgoing and de-marshalling of incoming messages according to the *Common Data Representation (CDR)* [7], and permit asynchronous communication that is transparent to the user.

3.1 IIOP Bridge

The **IIOPManager**, represents the ORB half-bridge and co-ordinates all the activities in the IIOP sub-system. It interfaces with foreign ORBs and Rosette. Incoming messages are in the form of CDR-encoded byte streams. They are intercepted and forwarded to the IIOPManager by an **IIOPListener** actor (Section 3.2) which requests de-marshalling from CDR format to Rosette types from the **CDRAgent** (Section 3.4). All messages, in and out, must pass through the IIOPManager, thus ensuring that it handles all IIOP communication between Rosette and external ORBs. The IIOPManager does not support context-passing in messages, but offers the following services. It locates the object on which an invocation must

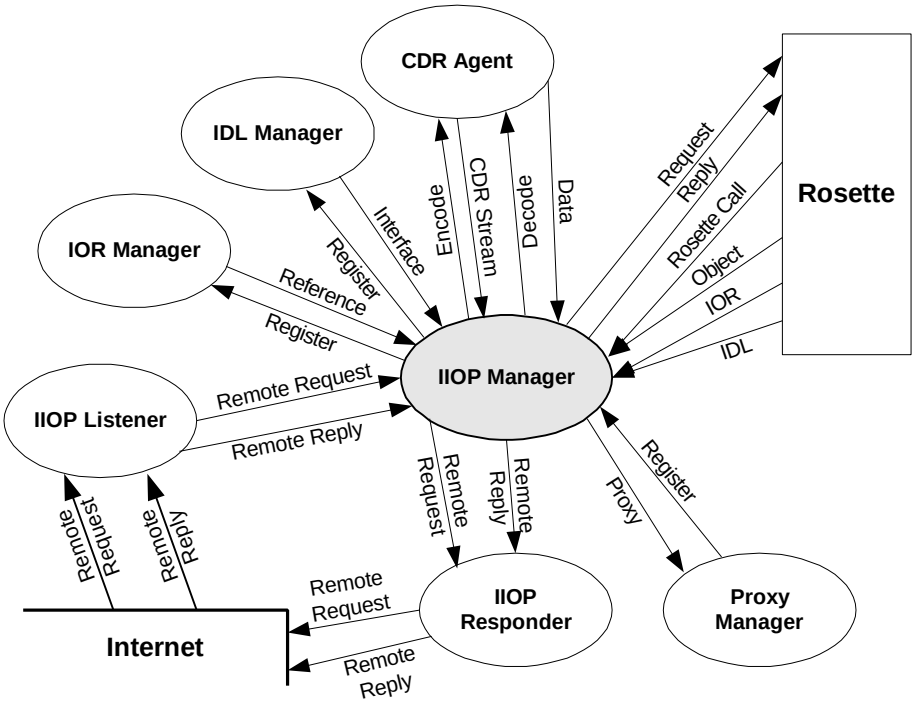

Figure 2 The IIOP sub-system

be performed, and invokes an operation on that object in response to an incoming remote request. It handles all communication: constructs the reply to an invocation and returns this to foreign ORBs, constructs requests to foreign ORBs, provides exception handling for badly constructed IIOP messages, and forwards exceptions raised by Rosette in response to requests from other ORBs. It also provides a remote request/reply mechanism, whereby IORs replace unmappable objects in outgoing messages, and generates one proxy object for each IOR encountered in a message. It matches outgoing remote requests with incoming replies.

3.1.1 Handling Incoming Remote Requests

On detecting an incoming message, an **IIOPListener** (Section 3.2) waits until the entire message has been received (using length information which it extracts from the message header) before passing it on to the IIOPManager. If the message is a valid IIOP message the IIOPManager requests decoding from the **CDRAgent** (Section 3.4) then routes the message onto a different handler according to the message type. The incoming message can either be: (1) a *reply* to a request issued by a Rosette object and made though the IIOP sub-system, (2) a *request* message from a foreign ORB to a Rosette object or (3) an unrecognised message type.

In the case of a reply to an earlier remote request made through the IIOP System (message type **IIOPReply**), the CDRAgent provides the IIOPManager with the request identifier and reply type, which are then used to retrieve the context in which the original request was made. The reply type can either be the results of an operation, a call to another actor, a system or user exception, or a location forward. In the event of an exception, the user that issued the request is informed accordingly. A location-forward reply type means that the object does not reside in the ORB to which the request was sent. In this case, the reply body will contain the new IOR where the desired object can be found and the IIOP System redirects the request to the new location. Thus redirection remains transparent to the request initiator. If the reply contains the results of a request, the IIOPManager extracts the proxy and context type from the message. Using the proxy, an IOR for the requesting object can be obtained from an **IORManager** (Section 3.3). The context yields the method name. Given the object and method, the signature for the method needs to be determined, i.e., the types of its parameters and result. This poses a problem in Rosette since the language does not identify types explicitly in its definitions. The **IDLManager** provides a type repository service, which is needed to ensure strong typing for IIOP compliant invocations. This actor registers objects with their signatures in the form of a Rosette tuple, with one member for each of the result and parameter types. Method signatures are grouped within a hash-table. The IDLManager maintains a global hash-table for all objects indexed by object type, which points to a second table of methods and their signatures. This provides an efficient mechanism for adding IDL signatures, and for querying them without fear of clashes that occur due to synonymous method names. Once the signature of the method is determined, it is passed onto the CDRAgent who demarshalls the reply and returns the result to the IIOPManager via a Rosette context return.

In the case of a request from a foreign ORB (message type **IIOPRequest**), the CDRAgent provides the IIOPManager with the request identifier, length, method name and the object IOR, all of which are contained in the message header. The IORManager can then determine the required object using the IOR. If the requested object is missing, then the CDRAgent marshalls a reply that indicates a system exception. If the object is found, the IIOPManager obtains the method's signature from the IDLManager and dynamically creates an expression that will make a Rosette call on the required object. The expression is evaluated and the result stored. In the meantime, the IIOPManager creates a reply message header. If the result of the operation is successful, the reply header will indicate that no exception has occurred, and the CDRAgent will encode the result and out parameters. Otherwise, the header indicates a Rosette user exception. The marshalled reply is then handed to an **IIOPResponder** (Section 3.2) who delivers it to the requesting ORB.

For any other type of incoming message the IIOPManager simply indicates that the IIOP System does not support that message type.

3.1.2 Handling Outgoing Remote Requests.

Since remote requests should be invoked transparently, the syntax and semantics of calls made by Rosette users on a remote object, located in another ORB, must be identical to any other Rosette request. Such invocations are performed on a proxy

object which will intercept the message, and forward it to the IIOPManager in the form of a Rosette context. This context contains the method to be invoked, and its parameters. Proxies are introduced into the system dynamically, but have an associated IOR in the **ProxyManager** which provides a proxy-IOR registration service. The proxy-IOR correspondence is stored in hash tables permitting efficient bi-directional querying. However, there are two problems associated with requests made on proxies: (1) the proxy must be able to accept operations that have not been defined on it, so that it can handle a request even if it does not directly support that operation and (2) the IIOPManager should not busy-wait whilst waiting for the reply of a remote request to a foreign ORB.

Customising Rosette solves the first problem. The IIOPProxy is a Rosette actor with a special constructor that has been devised to give it the capability of accepting operations that are not defined on it. Each actor is initialised with a reference to the IIOPManager, and when an operation is invoked on a proxy, it is delegated to a remote-request handler in the IIOPManager.

Reflective methods are employed to tackle the second problem. The proxy contains a reflective method, *delegated*, which triggers request information on the IIOPManager. When the IIOPManager constructs the remote request, it stores the request identifier together with the context passed by the IIOPProxy. This information will help the manager to match replies received with requests, thus freeing it to continue with other activities.

As mentioned, each proxy has an associated IOR in the ProxyManager, which is used by the IIOPManger to determine the host and port to which the request is to be sent. The IIOPManager also has the context of the call, and can therefore extract the name of the operation and its arguments. Using this information, the method's signature is obtained from the IDLManager, the message is marshalled by the CDRAgent, and forwarded by the IIOPManager. Once a reply is returned, it is demarshalled, the identifier matched against the request identifier, and the results returned via a Rosette context return on the proxy. Thus the invocation will receive a reply as if it had been computed locally from within the Rosette system. Although remote IIOP requests take longer than Rosette invocations, this does not affect the Rosette communication sub-layer, which makes no guarantees on delays.

3.1.3 Exception Handling

There are two types of exceptions, those generated by CORBA and those generated by user code. A system exception is raised when a request is made on an object not registered with the IIOP System, or when marshalling information is invalid. A user exception may be raised by Rosette in response to an invocation of a Rosette operation. Exceptions are marshalled into the reply header and sent to the requesting ORB. Currently, exceptions are only detected and reported to the user, but processing still continues. Implementation of exception handling has been left up to the user.

3.2 Transport Layer

The transport layer is a TCP/IP implementation for Rosette, and is responsible for the delivery of IIOP messages from a given port to the IIOPManager, and vice-versa. Services handled by this layer include: (1) returning invocation results to the IIOPManager, (2) returning marshalled replies to foreign ORBs, (3) relaying requests/replies to IIOPManager and (4) ensuring asynchronous communication that is transparent to the user. Within the IIOP System, a dedicated actor is used to listen for messages (IIOPListener), and another for communication (IIOPResponder).

The **IIOPListener** is actually a predefined Rosette actor (called a TCPListener) receiving messages from a given port. When detecting a message that corresponds to a new communication, the IIOPListener creates an **IIOPResponder**, which will process the incoming stream. One responder is set-up for each communication channel. Its function is simply to collect all data that constitutes a request/reply, whether incoming or outgoing. For an incoming message the IIOPResponder is initialised with the address of the IIOPManager and a reference to the message handler in the IIOPManager which will carry out all further processing of the message. When the entire message has been received (the message length is specified in the header), the responder wraps the received data in an **IIOPPacket** actor which holds a *ByteVec* – an indexable Rosette structure that represents an octet sequence – and sends the information to the IIOPManager. For outgoing messages, it is the IIOPManager that spawns an IIOPResponder, initialising it with the stream that contains the data, and the host and port to which it must be sent.

3.3 Interoperability Object Reference (IOR) Management

The IIOP System supports remote invocations to/from objects located in external ORBs. To this end, it must translate between Rosette object addresses and interoperable object references in both incoming and outgoing messages. IOR management provides these services, namely it: registers all Rosette objects that will participate in the IIOP, ensures the uniqueness of the IORs, matches incoming remote requests with the correct Rosette objects and replaces object addresses with their respective IORs in outgoing messages.

Within the IIOP System, the **IOR** actor is a simple wrapper for the IOR data structure containing a field for every field of an IIOP profile and providing the access functions to them. An IOR can arrive as a parameter of a remote request, or as result of a reply. In the latter case, the IOR is recreated from the demarshalled information. An IOR can also be introduced into the IIOP System explicitly. CORBA specifies an IOR format which is a hexadecimal representation of a CDR-encoded IOR. A more popular form is the URL-style IOR which is more visible and provides an opportunity for checking. In addition to these, the IIOP System presented here also supports IORs in the form of a Rosette tuple and provides translation services from one format to another.

All Rosette objects on which remote invocations can be performed from external ORBs must register with the IORManager via the IIOPManager. The **IORManager** actor provides storage and querying facilities on references to Rosette objects, and ensures the uniqueness of IORs. Object registration can be done using explicit

references, or using references that are created by the IORManager. The associations between Rosette objects and IORs are stored in double indexed hash-tables which permit efficient two-way queries. The IORManager also offers an object de-registration service, which makes those objects inaccessible to foreign ORBs.

3.4 Marshalling and Demarshalling

The Common Data Representation (CDR) is the CORBA-specified format for data marshalling. Within the IIOP System, marshalling and demarshalling activities are handled by the **CDRAgent** which caters for both encoding and decoding of data. These operations raise two issues that must be dealt with: byte-alignment and byte-ordering. In order to understand how these problems arise, it is necessary to note the various data types supported by CDR, and to understand how octet sequences are represented in Rosette. CDR supports two main categories of data types: *primitives* (including byte, character, boolean, long, unsigned long, short, unsigned short, double and float) and *constructed* types (including arrays, sequences, structs and structures such as the IOR). In Rosette, the internal representation of an octet sequence is a *ByteVec*. This is useful since the basic unit corresponds to bytes, and the ByteVec structure can be indexed. Byte-alignment problems arise during the *encoding* of primitives since the index of the first byte in a primitive has to be a multiple of the its length. For example, a Long is 4 bytes long and so must be encoded starting at index 0, 4, or 8 etc. Byte-ordering issues arise during *decoding* activities. The CDRAgent must be able to decode numbers represented in a different byte ordering. This involves reversing the ByteVec that holds a given number.

Encoding functions within the CDRAgent – one function per data type handled – deal with the problem of byte-alignment. Each data item is routed to its respective function using the type information provided to the CDRAgent by the IIOPManager. Byte-ordering is not a problem during encoding since numbers are represented in the ordering of their local system. It is, however, necessary to specify the correct ordering in the IIOP message that is being set-up by the CDRAgent. Thus, the agent needs to know the byte-order that needs to be used which it obtains from a system global flag.

Decoding proceeds in a similar fashion to encoding. Once the CDRAgent is loaded with a buffer, type information is used to read each data item and to forward it to the correct decoding module which handles type-verification and takes into account padding and byte-ordering issues. As the data is read from the buffer, the index is moved forward to the next data item. Table 1 shows the mapping of Rosette data types to CORBA types.

Rosette abstracts details such as number representation, thus making it difficult to find the byte representation of longs, shorts, floats and doubles. The representations could be derived using mathematical manipulation, but it is simpler to handle this at the Rosette implementation level (which is in C++). Therefore, the language must be extended with a means of converting numbers into ByteVec structures, and of converting ByteVec structures into their numerical representations. The Rosette language extension has been implemented as operations in the form of coercion functions. For example, to convert a Long to a ByteVec, a Fixnum is first passed to a C++ function that converts it to a Long. Next, a ByteVec structure that has length 4

Table 1 CORBA Rosette type mapping

Rosette	CORBA	Rosette	CORBA
Fixnum	long unsigned long short unsigned short octet enum	Tuple	struct sequence union array
Float	float double	Character	char
Bool	boolean	Symbol	string
String	string		

(the length of a Long) is created. Finally, the bytes forming the Long are copied into the ByteVec structure. The ByteVec, that now holds the Long's representation, can then be introduced back into the Rosette system. Conversion from a ByteVec to a Long involves passing the ByteVec to a C++ function that stores its contents in a memory location. The contents of the ByteVec are then cast into a Long. Finally, the Long is converted into a Rosette Fixnum. When encoding, the byte-ordering will be that of the system that Rosette is implemented on. When decoding, the C++ functions expect the ByteVec to contain numbers in the local byte-ordering, which can be reversed at the Rosette level whenever necessary.

4 RELATED WORK

Smalltalk IIOP Implementation. Smalltalk and Rosette both have an object model that is uniform but different to C++. The Smalltalk/CORBA mapping [12] deals with issues such as Smalltalk data types and memory management, object reference representation and naming conventions.

Whenever possible, IDL types are mapped directly to existing and portable Smalltalk classes, e.g., the CORBA sequence is converted to a Smalltalk *OrderedCollection*. One of the design goals was to make every Smalltalk object used in the mapping a pure Smalltalk object. All data types are stored completely within Smalltalk memory, so no explicit memory management is required. Objects that are not used are garbage collected. Object references are designed as Smalltalk objects that represent a CORBA object. The Smalltalk object must then respond to all messages defined by a CORBA object's interface. This is similar to the IIOPProxy actor defined in the Rosette implementation. Many of the same design decisions were necessary for the Smalltalk mapping as for the Rosette mapping of CORBA. However, the data conversion functions reflect the language specifics in each case. The Smalltalk implementation also needs to translate identifiers according to pre-established naming conventions, e.g., *is_prime_number* to *isPrimeNumber*, since SmallTalk does not allow identifiers containing underscores.

ANSA-OSI Adapter. As with Rosette, OSI Network Management has generally been ignored by distributed programming environments such as ANSAware, CORBA and DCE. This has prompted the need to provide OSI Network Management tools with an interface to ANSAware management interfaces and vice-versa, provide ANSAware applications with access to OSI Network Management objects [11]. The adapter must deal with issues such as interface specification, object references and naming conventions, interaction mechanisms and memory management.

Furthermore, mapping of the different interaction mechanisms adopted by each model is complex: OSI adopts message-based communications where messages can be sent to multiple objects and produce multiple results, whereas in ANSA interactions are either operation invocation (with a single response) or announcement operations (with no responses). Therefore, translating between the two communication styles involves two distinct phases: conversion of CMIP [11] messages into ANSA-CMIS operation invocations (or terminations), and then a mapping of ANSA-CMIS operations to corresponding ANSAware operations as expected by ANSAware interfaces that support them. Operation invocation is implemented similarly to our system despite a fundamental difference between ANSA and CORBA: ANSAware does not provide a dynamic invocation interface (only static interfaces). In order to dynamically invoke an operation, a specification database is used to store signatures of operations and information on how to convert them into an ANSA operation. This is similar to the type repository implemented by the IDLManager, but is much more complex since it stores a larger variety of information (such as GDMO/ASN.1 and IDL specifications). This requires more complex mechanisms for adding and removing specifications than in the IIOP System. Furthermore, for scalability reasons the ANSA/OSI interface specification database must be distributed which requires further load-balancing, availability and replication controls.

Additional problems are caused by the different strategies used by OSI and ANSA for naming interfaces. The exclusion of certain characters in the ASN.1 character set means that lexical translations and naming conventions have to be used to convert GDMO and ASN.1 identifiers into IDL, which is not the case with Rosette.

Generation of object identifiers is also more involved. Each statically defined GDMO element, such as a class template or attribute template, needs an identifier. Since all classes inherit from a top class, one identifier is defined for this top class and then a mechanism generates identifiers for each GDMO component resulting from the IDL interfaces translation. This is a much more involved process than the IOR generation mechanism implemented in the Rosette IIOP System.

The design issues involved in implementing the adapter are similar to those associated with the IIOP System, with differences arising at model-specific and model-implementation levels. The overall architecture deviates from ours in that the adapter is implemented as a collection of different tools which may each be used independently, whereas the Rosette IIOP System is a single system that encompasses different actors.

5 CONCLUSIONS

Systems development using an esoteric programming language will be restrictive without a mechanism to communicate with alien code. Using CORBA interoperability can eliminate the need for building multiple bridges to different systems. However, the architectural choices made in CORBA make building bridges complex and difficult. Furthermore, full automation is sometimes not achievable, in which case naming conventions and programmer's discipline are required.

CORBA/Rosette interoperability has been achieved by designing and implementing a bridge between the two systems using IIOP as the communication protocol with external ORBs. The differences between the interaction paradigms, procedure call in CORBA and asynchronous message passing in Rosette, require the IIOP sub-system to maintain considerable information in the form of contexts. CORBA stresses the use of statically defined interfaces and strong typing while interpreted languages such as Rosette often rely on weak type compatibility and dynamic behavioural changes. The IIOP sub-system must therefore maintain IDL specifications for Rosette objects that can be accessed from external ORBs in order to perform type checking. However, since the behaviour of a Rosette object may change over time, every Rosette object must provide a special interface that supports the invocation of arbitrary operations. Marshalling and de-marshalling of parameters are handled by a dedicated CDRAgent actor, which performs the conversions between CDR and the Rosette types. Because Rosette has a weak type system (e.g., tuples, fixnums), the CDRAgent must also perform bounds checks when converting data. Although the current implementation does not cater for this, bound checking can be carried out before decoding commences and an exception announcing a "subscript" error would be raised if a check fails.

Since the naming of actors in the Rosette system significantly differs from CORBA object references, an IOR actor is necessary in order to register all Rosette objects that participate in IIOP interactions. Similarly to the functioning of an ORB the bridge offers the ability to create a proxy for every external object on which an invocation is performed. A ProxyManager maintains then all the IORs associated with the proxy actors. However, a reference to the proxy must be maintained in the IIOP Manager which performs the requests and receives the reply. This is necessary because the IIOP Manager must be able to delete the proxy actor once the result of the invocation has been received. Although the bridge architecture relies on a central component, the IIOP Manager, the underlying computational model of Rosette ensures that all activities are performed concurrently in multiple threads.

Preliminary performance and interoperability tests with other non-Rosette ORBs have proven that the framework implemented is usable but still requires additional effort. Further work was undertaken in a commercial environment to extend and optimise the implementation. In particular, protocol optimisations such as request cancelling and forwarding had to be implemented. Additional, support for context passing in requests was also provided, as well as a better exception handling mechanism. Many parts of the CORBA specification, such as those on implementation and interface repositories, and IDL interpretation [5], were omitted here in order to focus on the IIOP service implementation. Overall improvement to system design could be achieved by incorporating more inheritance into the

managing actors' structure, e.g., by deriving all managers from an abstract manager and overriding registration and de-registration methods as required.

This work reports an initial prototype implementation, which was extended commercially by *Trans Enterprise Computer Communications Ltd. (TECC).* TECC have implemented and tested a full Rosette ORB, called *TECCware*, which includes IDL compiler, full IIOP support and interface repository [8]. The approach of building the ORB into the language allows the simple construction of CORBA servers, which offer all the benefits of both Rosette and CORBA. Their software has been used successfully in large applications in the financial domain.

6 ACKNOWLEDGEMENTS

We gratefully acknowledge the advice and help provided by Frank Taylor and Matthias Radestock of TECC Ltd and the other members of the Distributed Software Engineering Group at the Imperial College Department of Computing, as well as the financial support from the EPSRC under grant ref: GR/K73282.

References

[1] AGHA, G. A. ACTORS: A Model of Concurrent Computation in Distributed Systems, MIT Press, Cambridge, Mass., 1986.

[2] HEWITT, C. Viewing Control Structures as Patterns of Passing Messages, Artificial Intelligence, 1977.

[3] Microelectronics and Computer Consortium, 1991, Rosette Reference Manual, available from: http://www.mcc.com/projects/carnot/rosette.

[4] GOLDBERG, A., ROBSON, D. Smalltalk-80: The Language and its Implementation, Addison Wesley 1983.

[5] Object Management Group (OMG), The Common Object Request Broker: Architecture and Specification, Sections 19-20, July 1995, available from: www.omg.org

[6] UNGAR, D., SMITH, R. Self: The Power of Simplicity. Proceedings OOPSLA '87, October 1987.

[7] Object Management Group (OMG). CORBAservices: Common Object Services Specification, March 1995, available from: www.omg.org

[8] TAYLOR, F., RADESTOCK, M. TECCware product definition. Technical report, Trans Enterprise Computer Communications Ltd. 1998, available from: www.tecc.co.uk

[9] STEELE JR., G. L. Common Lisp the Language, 2nd Edition, available from: www.cs.cmu.edu/Groups/AI/html/cltl/clm/clm.html.

[10] LINDHOLM, T. and YELLIN F. The Java Virtual Machine Specification, Addison-Wesley: The Java Series, Sept. 1996.

[11] GENILLOUD, G. and POLIZZI, M. Managing ANSA Objects with OSI Network Management Tools, Proceedings of the Second International Workshop on Services in Distributed and Networked Environments, June 5 – 6, 1995.

[12] DNS Technologies Synergistic Software, *Smalltalk Broker*, available from: http://www.dnstech.com/stbfaq.htm.