Object oriented distributed computing infrastructures and programming language interoperability

Hans – Arno Jacobsen
International Computer Science Institute
1947 Center Street, Suit 600
Berkeley, CA 94703
jacobsen@icsi.berkeley.edu

May 16, 1997

Abstract

Achieving transparency and interoperability across distributed, heterogeneous computing environments is a major goal of several distributed computing frameworks, developed over the past five to seven years. This goes hand in hand with the question of how to provide an architectural framework that integrates different programming languages and paradigms for building distributed systems.

This work presents an overview of common object oriented distributed computing infrastructures which address these problems. We motivate key concepts of the individual approaches and pin point their differences.

Our main focus is on how the individual approaches achieve programming language interoperability.

We present a detailed study that shows how to integrate the Sather object oriented programming language into one of the more common infrastructures. This manifests itself in the second part of our work which presents the IDL–to–Sather language mapping, developed.

Contents

1 Introduction

Interoperability of software and hardware systems, software component composition in a building–block–like manner, as well as, compositional libraries have long been the objective of research in computer science.

It is thus not surprising that with the manifestation of object orientation several new approaches have emerged based on these concepts. Clearly, the advantages OO technology offers, namely encapsulation and code–reuse, is evident for this context.

As a matter of fact, compositional libraries have at all been made widely possible by object oriented approaches. The question of whether these libraries have to be replicated for different programming languages or environments remains still largely a problem.

Adding the distributed nature of many large applications, the vast heterogeneity of computer hardware and operating systems, the problem becomes even harder.

Several architectural frameworks have been proposed and implemented over the past five to seven years which try to solve the above outlined problems in a uniform manner.

In this paper we will take a comparative look at the following approaches:

- The Common Object Request Broker Architecture (CORBA), specified by the Object Management Group (OMG) [Gro95a] in 1989 and constantly refined up to the present day.
- The Inter Language Unification (ILU) [BJ97] designed at Xerox Parc.
- OLE/COM, Microsoft's proprietary solution to distributed component computing (recently renamed to ActiveX).
- A programming language–based approache

We will look at how these approaches are leveraging the boundaries of heterogeneous platforms, different operating systems and programming languages in order to solve the problem of system interoperability and integration.

Our key interests is with programming language interoperability. We therefore spend considerable time defining an 'interface' for one of the frameworks presented.

Section 2 presents our survey. In section 4 we define a language mapping for IDL to Sather.

2 Various approaches

This section describes the individual distributed computing infrastructures. We spend by far the most time on CORBA. This is, because it is well establish and cleanly defined. We also use it as a reference to compare the other approaches against.

ILU is very close in overall conception and design to CORBA so that we omit details and concentrate on outlining differences to CORBA.

It turned out to be very difficult to get a clear picture of OLE/COM. It intends to address the same issues as the above two approaches but does so in a very peculiar way. We give a top–level description and try to relate it to the other two approaches by comparing their key features in a separate section.

Finally, we describe an alternate way to obtain programming language interoperability. The approach is based on programming language support, as offered by the Sather object oriented language.

2.1 The Object Management Architecture and CORBA

The Common Object Request Broker Architecture specification, CORBA 2.0, [Gro95a] is an emerging standard for heterogeneous distributed object computing proposed by the Object Management Group

(OMG). CORBA is part of a larger architectural framework – the Object Management Architecture (OMA) – for distributed object computing at large. The OMA specification [Gro95a] was initially defined by the OMG in the fall of 1989 and revised in 1992. Figure 1 shows its main elements:

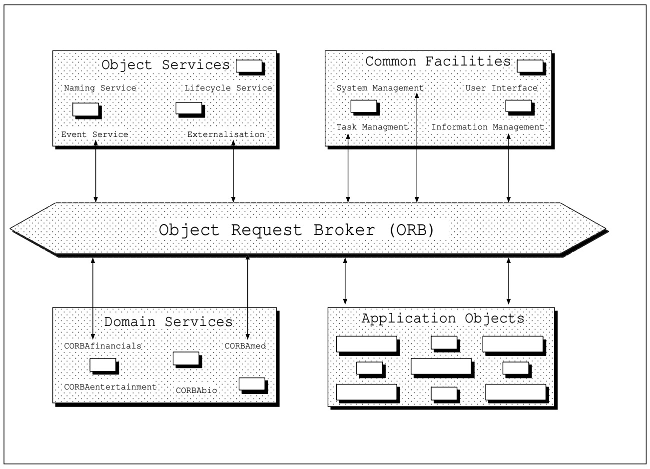

Figure 1: The Object Management Architecture [OMG 89].

1. the *Object Request Broker* (ORB), also referred to as object bus, makes up the core part of the architecture
2. the *Common Object Services*, accessible via the ORB, providing basic services for the entire architecture
3. the *Common Facilities* divided into domain specific, *Vertical Facilities* and application transcending *Horizontal Facilities*
4. the *domain applications* are the actual domain specific user defined components

The *Object Request Broker* serves as communication infrastructure. It defines interfaces and operations letting objects transparently request and receive operations from other objects located locally or remotely on the object bus.

The functionality provided by the ORB is supported by the *CORBA Common Services*, most of which have already be standardized. These services include naming service, transaction service, persistence service and trader service among others [Gro95b].

The *Common Facilities* are intended as application–level services. They may include services such as GUI components and editing facilities. This part of the architecture is still in the Request–For–Information–phase of the standardization process.

The *domain applications* are user defined objects. These make up the applications running on top of the framework using services, facilities and the request broker to communicate with other applications and services.

CORBA provides a uniform object–oriented communication infrastructure. It aims at facilitating and enabling transparent and location–independent invocation of services distributed across heterogeneous computing platforms, regardless of operating system and implementation language.

This component interoperability is achieved through the use of the Interface Definition Language (IDL). All components of the architecture announce their functionality by defining an IDL interface. IDL is a non–algorithmic, declarative specification language with object oriented features, such as encapsulation, limited polymorphism and multiple inheritance. The IDL interface is mapped to a programming language by a stub–compiler generating programming language stubs for interfacing the application to the ORB on client, as well as, server side. These programming language stubs handle argument and return value marshaling and demarshalling and also the remote method invocation.

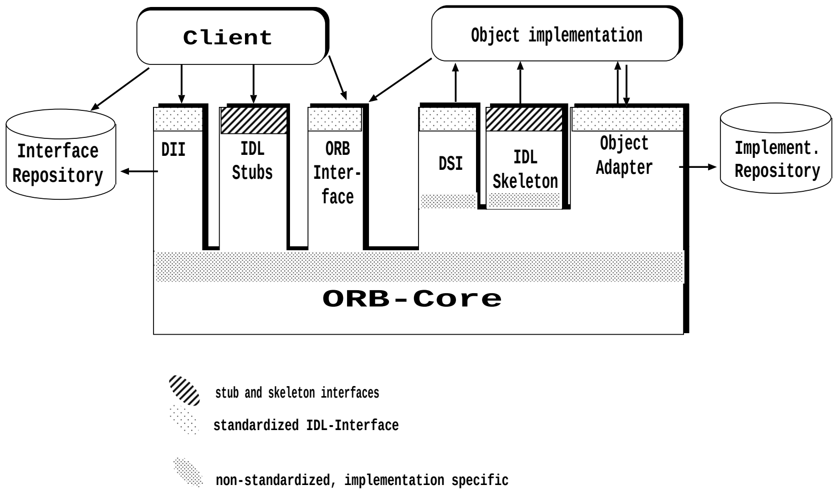

Figure 2: The Common Object Request Broker Architecture [OMG 95].

Mappings of IDL to C, C++, SmallTalk and Ada have been defined [Gro95a]. Mappings for other languages are in preparation. In this work we will develop an IDL–to–Sather mapping, presented in great detail in section 4.

It is important to understand that the language mapping is the actual interface to CORBA a programmer will see. Even though the mapping will usually be generated by a support tool for a given interface, the programmer will need to thoroughly understand it, since it defines how certain types are represented in the language used.

We now turn to the Object Request Broker, which implements the actual communication functionality of the entire architecture.

Figure 2 gives a more detailed picture of the request broker architecture. For the sake of illustration one side has been denoted as client and the other as object implementation–side. These are not pre–defined roles. An object may be client to one object and server to another one at the same time.

In the following we will briefly go over the individual components of the architecture.

The *interface repository* is essentially a database with appropriate interfaces defined in IDL. It contains type information for all the objects located on the object bus. All abjects have to register their interfaces with it. The repository may be used during operation of the system to obtain information of objects at run–time. This is for example useful to dynamically invoke methods on objects whose interfaces were not available at compile time.

The *implementation repository* is another database which contains information about the status of objects. It tracks whether an object is currently running, where it is running and the like.

The *Basic Object Adapter* (BOA) handles incoming client requests and passes them on to the corresponding object. It uses information stored in the implementation repository to locate and activate objects.

The ORB, itself, provides several in IDL specified interfaces to access its functionality. It provides a simple naming service (a separate naming service is defined as CORBA service) which helps bootstraping an application.

The *static stub and skeleton interfaces* provide the connection of an application to the ORB. One of such interface is defined for each application object on the bus. Stubs and skeletons handle argument marshalling and unmarshalling.

The *dynamic invocation* and a *dynamic skeleton interface* are alternate mechanisms to pass and receive requests to and from applications. These are mainly used in situations where prior to compilation of the application no IDL interface for a particular object had been available. This situation may, for example, occur in interpreted or drop–and–drag based programming environments where objects are added and compiled by users at different times. Recompiling the entire system would be awkward and inefficient each time a new object is added. Requests are therefore constructed at run–time. Note, however, that in order to construct the request the IDL interface of the target object needs to be available in a machine readable form. The *interface repository*, as described above, takes over this task.

2.2 Inter–Language–Unification

ILU's primary goal is to provide a framework for inter–connecting different programming languages. Furthermore, it provides mechanisms for building distributed systems out of individual software components written in different programming languages. It is a specification as well as an implementation, freely available from Xerox Parc.

ILU is described as an multi-language object interface system by its designers [BJ97]. Its main features are:

- support for building multi–lingual object oriented libraries with language–independent interfaces
- support for implementing distributed systems
- support for documenting interfaces between modules of non–distributed programs.
- hide implementation distinctions between

– different languages
– different address spaces
– different operating system

These are much the same features as CORBA aims to provide. The mechanism applied by ILU to achieve programming language interoperability is essentially the same as in CORBA. An interface defined by an universal definition language is mapped into stubs, handling argument conversions, and method invokaction for the particular programming language used.

We therefore, omit details and focus on its key differences to CORBA.

ILU interfaces are specified in ILU's Interface Specification Language (ISL). ISL is not the same as OMG's IDL. The key difference is that ISL does not support the CORBA type `any`.[1] It is therefore not possible to map IDL interfaces, using this type, into ISL interfaces. This of course, renders a full interoperability between both systems impossible. ILU, does, however, support a subset of OMG IDL, as a second interface definition language. To interoperate with CORBA, components need to use this subset to specify their interfaces.

ILU does not foresee a mechanisms for invoking operations dynamically, such as the dynamic invocation and the dynamic skeleton interface in CORBA. Moreover, ILU does neither provide an interface nor an implementation repository.

Currently ILU supports the following programming languages Common Lisp, C++, ANSI C, Modula-3, and Python. it runs on most common operating systems and platforms, including Windows.

2.3 OLE/COM and ActiveX

This section gives a brief overview of OLE/COM, Microsoft's solution to language interoperability and possible support for distribution. It is difficult to find a well rounded description and architectural overview of OLE/COM. We give, therefore, a very top–level description of the key concepts. We dtew our knowledge from [Bro96].

OLE was initially geared towards providing support for compound documents. Its initial name, Object Linking and Embedding, still reminds of that. By 1991 an object model for OLE, named COM (Component Object Model) was added. COM is a specification that provides a language–independent binary standard for object implementations.[2]

This standard is based on the concept that an object implements a set of functions, subsets of which are collected in interfaces. An object may implement one or all of its interfaces. An implementation of an interface is an array of pointers to functions, referred to as vtables. This terminology is inspired by the related concept from C++. COM Interfaces are essentially abstract base classes in C++.

Clients access COM objects through these interfaces, and not through a pointer to the object itself. Figure 3 depicts the basic idea. This structure is central to the binary standard of COM. It avoids the need for an additional interface definition language, as required by the above two approaches. This is, however, little gain, compared to the restriction imposed by it. OLE/COM does not support multiple inheritance. This is a strong weakness in our eyes, since inheritance is a key concept for designing object oriented systems.

[1] Due to limitation of space have we not discussed this type in the section on IDL. The OMG IDL type `any` may be substituted for any of the OMG IDL types, i.e. for basic types, as well as, for user defined types. It supports an interface that lets a client query its `TypeCode` denoting the actual run time type.

[2] In the meantime has the OLE/COM specification been given to an 'Open Group', called Active Group for further refinement. The next generation standard is collectively (OLE & COM) referred to as ActiveX. For this report we will still be using OLE/COM, since we were not able to find an appropriate standard reference.

Instead, OLE/COM provides aggregation for achieving implementation reuse. In the aggregation model, the object 'including' aspects of another object directly exposes the interface of the latter. The included object must, however, know that it is now acting as an included object, since method calls to some of its interfaces must be be served by the including object. (This scheme is as messy as it sounds.)

CORBA on the other hand, does not require or rule out implementation inheritance. It leaves this decision up to the implementor/vendor.

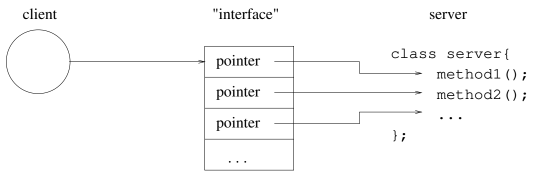

Figure 3: Client–Server relationship in OLE/COM.

Objects are implemented by servers. Three different kinds of implementations are possible, two of which are currently supported. These are in–process, local or remote. A server is implemented in–process if it resides in the same address space as the client. A local implementation refers to the crossing of address spaces boundaries upon method invocation and requires RPC support. OLE/COM bases these invocations on DCE's RPC specification. A true remote invocation, crossing machine and network boundaries is not yet available, but announced as DOLE (distributed ...).

OLE/COM, as it is currently available, does not offer support for building distributed systems. This is a considerable weakness, as compared to the 'competing' approaches.

Language interoperability is achieved through the binary standard, strongly based on C++. The OLE/COM description [Bro96] allows interoperability with any language that allows for double pointer indirection. This seems to be a severe restriction, since most commercially accepted languages, besides C/C++, do not.[3]

So far OLE/COM is only available for Microsoft Window and NT. Extensions to other operating systems have been announced, such as for Mac and IBM mainframes. (this is very puzzling).

2.4 Programming language based approach

In this section we briefly describe a programming language approach to interoperability. The emphasize is on interoperability as compared to the above approaches and not at all on heterogeneous and distributed programming environment support. Clearly, the individual design goals are vastly different.

The language we chose, Sather [SO96],is particularly interesting in this respect, since it provides, as part of its language definition, a way to interoperate with other languages. This makes it distinct from the above described schemes, which impose interoperability from the outside, so to speak.

Sather provides the language construct '`external`' which is applied to classes. An external class which interfaces to ANSI C is designated with the language identifier 'C', similarly for FORTRAN.

External C classes may contain routine signatures without bodies. Such signatures must contain only built–in or user defined C types. The routine must be implemented externally in ANSI C. These

[3]This might not be true for VisualBasic, we don't know.

routines may be called from within regular Sather code, thus providing an interface to ANSI C code. Routines with bodies, whose signatures contain only C types, may be called from C, thus providing an interface from C to Sather.

The advantage of this scheme is its simplicity and clean integration into the language. Its major disadvantage is that adding another language to interoperated with, is a small undertaking in terms of compiler modification.

2.5 Alternatives, related approaches

Other approaches for achieving interoperability exist which we did not have the time to study in greater detail. The above selection of systems was mainly due to their object oriented nature and apparent popularity.

The Telecommunications Information Networking Architecture Consortium (TINA-C) provides its own solution, including a proper object definition language, ODL–95 [TC93]. Their primary focus is on interoperation between different networks.

The Distributed Computing Environment (DCE), proposed by the OSF [OSF96] is an earlier approach to achieving interoperability in distributed environments. The focus, there, however, is more on interoperability on the platform level and not so much on the programming languages level. Extensions have been deployed to introduce object oriented features into DCE, resulting in OODCE [Cha96].

Many 'special purpose solutions' for integrating diverse applications based on the object oriented paradigm have been proposed by independent companies over the last few years, for example, OpenDoc by CI–Labs/Apple, OpenStep by NeXT and Taligent by Taligent. Their focus, however, is on compound documents, i.e. multimedia documents.

3 Comparing the different systems

This section summarizes and compares the key issues of the different approaches discussed above. Table 1 presents the summary.

It is hard to draw prospective conclusions regarding the future of the individual approaches (CORBA or OLE/COM). Clearly, CORBA runs on more platforms, integrates more languages and supports distributed computing as one of its key concepts. OLE/COM on the other hand is shipped with every Windows platform and is therefore likely to outnumber CORBA supporting platforms.

It seems therefore safe to say that both technologies will most likely co–exist for some time to come. This seems to be also the opinion in the respective camps. Orbix, a very popular CORBA implementation, for example, provides access to OLE/COM components.

4 The OMG IDL–to–Sather–language–mapping

This section comprises the second part of our work. Here we demonstrate how to integrate a new language into the CORBA framework. Several steps need to be taken:

1. define a language mapping from IDL to the language of your choice
2. implement a stub compiler, i.e. a compiler that maps IDL interfaces into language stubs of this language

issue	OLE/COM	CORBA	ILU
		specification/standard issues	
Standard	propritary	open standard body	open/research
implementation	spec./impl.	specification	spec./impl.
		language issues	
lang. supported	C/C++	C/C++, SmallTalk,Ada	C/C++, Lisp, Modula–3 Python
interop. approach	binary standard, i.e. memory layout	stub/skeleton compilers	stub/skeleton compilers
paradigm	object–based; no inheritance; no clear encapsulation; several interfaces per object	object–oriented multiple inheritance; one interface one type	object–oriented multiple inheritance interfaces corr. to namespaces; types corr. to obj/classes
error handling	limited support by single bit in value returned	supports exception structures	supports exception structures
Dyn. invokation	OLE Automation type repository	DII/DSI interface repository	not supported not available
Garbage Collected	no, maually by `RefCount` op.	depends on lang.	depends on lang.
		platforms	
OS supported	Microsoft Window, NT	all common ones	all common ones
Distributed support	not yet, DOLE in make	yes, one of key concepts	yes, one of key concepts

Table 1: Summary of key issues of different distributed computing infrastructures.

```
<type declarations>;
<costant declarations>;
<exception declarations>;

module <identifier> {
        <type declarations>;
        <costant declarations>;
        <exception declarations>;

        interface <identifier>  [:<inheritance(s)>] {
                <type declarations>;
                <costant declarations>;
<attribute declarations>;
                <exception declarations>;

                [<op_type>] <identifier> (<parameters>)
                   [raises exception ][context];
                   ...
                [<op_type>] <identifier> (<parameters>)
                   [raises exception ][context];
        }// ends interface
        module <identifier> {...}
}// ends module

interface <identifier>  [:<inheritance(s)>]
    { ... }
```

Figure 4: Overview of IDL

We have completed step one, step two is not absolutely neccessary since the language stubs may be manually written. This is of course awkward, error prune and not recommended.

Section 4.2 presents the IDL–to–Sather language mapping. The mapping follows the guidelines put forth by the OMG for language mappings [Gro95a] in general. We will motivate the IDL construct considered, but will not explain them at large. The nect section will give a brief overview of IDL. For more details refer to [Gro95a].

4.1 The Interface Definition Language

OMG IDL is a descriptive, non–algorithmic interface definition language with object oriented features and support for distributed computing. It is very C++–like. It is used to describe interfaces of CORBA objects. Figure 4 gives a brief abstract overview of IDL.

For a more detailed description refer to **??**.

//IDL maps to – Sather	
module	Sather classes scoped by module name
interface	abstract classe
multiple inheritance	multiple inheritance
operations, attributes	methods, attribute accessor routines
basic types e.g. ushort	Sather classes providing the required behavior class CORBA_UNSIGNED_SHORT
struct, enum, union	Sather classes
sequence<T>, array, string	Sather classes : CORBA_SEQU{L,T} CORBA_ARRAY{T}, CORBA_STRING
typedef	Sather classes
constants	Sather class constants
exceptions	exceptions
any	Sather abstract class $CORBA_OBJECT

//IDL & –Sather	
module	no equivalent
interface	abstract class
operation	method
in parameters	default
attribute	attr
template type	parametrized class
typedef	no equivalent construct
type	class

4.2 Overview of the IDL–to–Sather Mapping

The following table summarizes the IDL–to–Sather mapping, more formally specified, in the next section.

4.3 Terminology

The following table summarizes the difference in terminologing used by the OMG and the Sather community. The difference in terminology is small but distinct.

In the following we use the term *Sather interface class* to denote an abstract classes representing a particular IDL interface.

In the mapping we will use the following terms to refer to sets of types. An OMG IDL *structured type* is one of `struc, enum, or union`. An OMG IDL *exception type* refers to an OMG IDL exception definition. An OMG IDL *template type* is one of `sequence<T>, string or array<T>`. If not further specified, strings and sequences may either refer to bounded or unbounded template types, arrays are always of fixed size [Gro95a]. An OMG IDL *basic type* is one of `short, long, double, ushort, ulong, boolean, or octet`.

We refer to identifiers in an OMG IDL definition as of being in *global scope* if the identifier appears outside any module and interface. Respectively for identifiers used inside modules and interfaces, as of being in *module–scope* and *interface–scope*.

The IDL–to–Sather mapping is described by an intuitive formalism. Our goal is to precisely capture the mapping but not to overwhelm the reader with formality. The OMG [Gro95a] language mappings are all based on a more 'prosaic' description of the mapping enhanced with examples to illustrate the principal ideas. We think a more formal approach to illustrate the mapping is more helpful in generalizing the mapping and for reasoning about it.

The formalism we have adopted describes the IDL construct in an intuitive manner based on its structure. All terminals, such as '`interface`' are present as such without extra markings.

Non–terminals are enclosed in brackets '< ... >'. They usually describe more complex entities such as an entire struct declaration, e.g. '<structured declaration>'. Whenever the more precise break–down of the non–terminals is needed it is given. In the Sather mapping we make reference to these declarations, by explicitly using their name or type.

The Sather representation of IDL is based on the same formalism. The use of lower case for non–terminals in IDL and Sather mapping description should indicate that any case of the identifier is allowed (i.e. upper, lower). In Sather we capitalize these identifiers to indicate that in Sather the capitalized form need to be used.

4.4 Design goals

The OMG IDL–to–Sather language mapping provides the ability to access and implement CORBA objects by programs written in the Sather object oriented programming language. The mapping is based on the definition of the "Common Object Request Broker Architecture and Specification, Revision 2.0" guidelines for language mappings. The OMG IDL–to–Sather language mapping uses Sather's support for object oriented programming to preserve the object model described by the CORBA Architecture and Specification.

The mapping was designed such that it would be easy to use, portable and efficient. Sather's strong object oriented character greatly contributed to achieve these goals.

4.5 The Mapping of IDL to Sather

The language mapping is defined by presenting the OMG IDL construct in a formal manner. The representation in Sather is given based on this formal expression. The mapping is briefly discussed and possible alternatives are presented. A concrete example of the mapping is given, at times, to better illustrate the key issues.

4.6 Syntactic considerations

This section specifies the mapping for IDL identifiers, literals, and constant expressions.

4.6.1 Mapping for Identifiers

IDL identifiers follow rules similar to those of Sather regarding case sensitivity and the use of underscores. Accordingly, the only rule for the mapping of identifiers is that an identifier that collides with a Sather reserved word must have be suffixed by "_".

4.6.2 Mapping of literal expressions

OMG IDL literals are mapped to lexically equivalent Sather literals. The following sections describe the lexical mapping of OMG IDL literals to Sather literals. This information may be used to provide semantic interpretation of the literals found in IDL constant expressions in order to calculate the value of an IDL constant or as the basis for translating those literals into equivalent Sather literals.

Integer Literals OMG IDL supports decimal, octal, and hexadecimal integer literals. They are mapped as follows:

- A decimal literal consists of a sequence of digits that does not begin with '0'. Decimal literals are lexically equivalent to Sather integer literal expressions requiring no specific translation.
- An octal literal consists of a leading '0' followed by a sequence of octal digits (0 .. 7). Octal literals are lexically equivalent to Sather octal literal expressions. (i.e., OMG IDL '016' becomes the Sather octal literal expression '0o16').
- A hexadecimal literal consists of "0x" or "0X" followed by a sequence of hexadecimal digits (0 .. 9, [a|A] .. [f|F]). Hexadecimal literals are lexically equivalent to Sather hexadecimal literal expressions. (i.e., IDL '0x14' becomes the Sather hexadecimal literal expression '0x14').

Floating–Point Literals An OMG IDL floating–point literal consists of an integer part, a decimal point, a fraction part, an 'e' or 'E', and an optionally signed integer exponent. The integer and fraction parts consist of sequences of decimal digits. Either the integer part or the fraction part, but not both, may be missing. Either the decimal point and the fractional part or the 'e' (or 'E') and the exponent, but not both, may be missing.

A lexically equivalent Sather floating point literal expression may be formed by appending, to the integer part (or "0" if the integer part is missing), with a decimal point, the fraction part (or "0" if the fraction part is missing), with an "E", and the exponent (or "0" if the exponent is missing).Optionally, the ending "E0" may be left off if the OMG IDL did not have an exponent.

Character Literals OMG IDL character literals are single characters or escape sequences enclosed by single quotes. The first form is lexically equivalent to an Sather character literal expression. The below table (not included) supplies lexical equivalent expression for the escape sequences defined.

String Literals An OMG IDL string literal is a sequence of IDL characters surrounded by double quotes. Adjacent string literals are concatenated. Within a string, the double quote character must be preceded by a '\'. A string literal may not contain the "nul" character.

Lexically equivalent Sather string literal expressions may be formed by mapping the individual characters of the IDL string according to the mapping defined for character literals. Adjacent OMG IDL string literals are mapped equivalently to Sather string literal expressions.

Particularly long string literals may be broken up according to this concatination mechanism, since Sather does not allow string literal expressions to extend pass a source line of text (e.g. `"string1" "string2"`).

Mapping of Constant Expressions In OMG IDL, constant expressions are used to define the values of constants in constant declarations. A subset of these expressions, that evaluates to positive integer values, may be found as the maximum length of a bounded sequence, as the maximum length of a bounded string, or as the fixed array size in constant declarations.

The exact representation of OMG IDL constants in Sather is described below.

Mapping of Operators The following table (not included) provides the correspondence between IDL operators in a valid constant expression and semantically equivalent Sather operators. This information may be used to provide semantic interpretation of the operators found in IDL constant expressions or as the basis for translating expression containing these operators into equivalent Sather expressions.

4.7 Scoped Names

OMG IDL identifier can be in any one of three scope levels, as explained above (global scope, module scope, interface scope). Identifiers in the same scope have to be distinct. Identifiers only differing in case collide.

The Sather programmer must use the correctly scopped name for an OMG IDL construct. Occurrences of '::' in an IDL definition are mapped to '_' in Sather according to the scoping level of the IDL type referred to.

Scoped names in OMG IDL are mapped to Sather scopes as follows:

- All OMG IDL types (structured, exceptions) defined at global scope are mapped to a proper class scoped by the type name according to the rules defined below. Constants declared at global scope are declared in a separate class – `class IDL_GLOBAL_CONST is ... end;`.
- All OMG IDL types (structured, exceptions) defined at module scope are mapped to proper Sather classes, according to the mapping defined below, scoped by the module name, followed by an underscore, followed by the name of the type. Constants declared at module scope are declared in a separate Sather class named after the module and suffixed by '`_CONST`'.
- OMG IDL interface types are mapped to Sather abstract classes as described below.
- All OMG IDL types (structured, exception) defined at interface scope are mapped to Sather classes prefixed by the scoped interface name, followed by an underscore, followed by the type identifier.

Since Sather requires class names to be capitalized all IDL type identifiers mapped to Sather class identifiers must be capitalized. Note, this will not cause name clashes of OMG IDL identifiers since OMG IDL identifiers are case insensitive. Furthermore, there is only one namespace for OMG IDL identifires so that using the same name for a constant and an interface produces compilation errors [Gro95a].

All other OMG IDL identifiers are mapped verbatim to Sather identifiers, according to the rules laid out below. The formal mapping is as follows:
Mapping for global scope:

```
//IDL

    <structured type declaration>
    <exception type declaration>
    <constant declaration>


-- Sather

    class <STRUCTURED TYPE IDENTIFIER> is ... end;
    class <EXCEPTION TYPE IDENTIFIER> is ... end;
    class IDL_GLOBAL_CONST is const ... end;
```

Mapping for module scope

```
//IDL  module

     <module identifier> {
          <structured type declaration>
          <exception type declaration>
          <constant declaration>
          <interface type declaration>
     }


-- Sather

     class <MODULE IDENTIFIER>_<STRUCTURED TYPE IDENTIFIER> is ... end;
     class <MODULE IDENTIFIER>_<EXCEPTION TYPE IDENTIFIER> is ... end;
     class <MODULE IDENTIFIER>_CONST is const ... end;

     abstract class <MODULE IDENTIFIER>_<INTERFACE IDENTIFIER> is ... end;
```

Mapping for interface scope

```
//IDL

     interface <interface identifier> {
          <structured type declaration>
          <exception type declaration>
          <constant declaration>
          <attribute declaration>
          <operation declaration>
     };


-- Sather

    class <STRUCTURED TYPE IDENTIFIER> is ... end;
    class <EXCEPTION TYPE IDENTIFIER> is ... end;
    partial class <INTERFACE IDENTIFIER>_CONST is
          <constant identifier> :<CONSTANT TYPE>:=<constant value>
    end;

    abstract class $<INTERFACE IDENTIFIER> is
          <attribute> : <ATTRIBUTE TYPE>
          <attribute>(a : <ATTRIBUTE TYPE>)
          <operation declaration>
    end;
```

A partial class is used to declare the constants of the OMG IDL interface. We could also use a regular class. It is, however, not possible to simply define an accessor routine signature as part of the abstract class interface, since we also need to provide a mapping for the value of the constant.

This mapping gives the object implementor the maximal degree of freedom. See section **??** for more detailed discussion of object implementation alternatives. The partial class can simply be included in the object implementing class.

We now illustrate the mapping on a simple derived example, some additions have been made to better capture the above concepts.

```
//IDL

    struct COMPANY {...};
    const string OS = "NT";
    module database {
          const long lang = 1;
          enum flags {...};
          struct datum {...};
          interface dbm {
               const long owner = 3;
               attribute string dbName;
               enum SecurityLevel {...};           ...
          };
    }
```

is mapped to

```
-- Sather

    class COMPANY is ... end;
    partial class IDL_GLOBAL_CONST is const OS:CORBA_STRING:= "NT"; end;
    partial class DATABASE is const lang:CORBA_LONG:=1; end;
    class DATABASE_FLAGS is ... end;
    class DATABASE_DATUM is ... end;
    partial class DATABASE_DBM is const owner :CORBA_LONG:=3; end;
    class DATABASE_DBM_SECURITYLEVEL is ... end;
    abstract class $DATABASE_DBM is
          dbName : CORBA_STRING;
          dbName(d : CORBA_STRING);
    end;
```

Unlike C++ or Java, Sather does not provide a namespace or package construct which would be a 'natural' mapping for representing OMG IDL modules. The scoping rules described above have to be used to 'mimic' the notion of a module. Modules are used in IDL to introduce namespace [Gro95a], no other functionality is associated with them.

4.8 Ambiguities in Mapping

The following ambiguities may arise for the defined mapping:

```
//IDL
     struct bar_foo {...}
-- Sather
     class BAR_FOO is ...
```

```
// IDL
     interface bar {struct foo{...}}
-- Sather
     class BAR_FOO is ...

//IDL                                 -- Sather keyword
interface PROTECT {                   -- keyword 'protect'
     void protect ()
};
```

Other cases can be constructed. Similar problems arise for constructed examples for the OMG IDL C and C++ mapping. OMG recommends a careful use or avoidance of the underscore character in IDL interface definitions. Note in Sather it is not allowed for an identifier to start with an underscore which would make it possible to partially disambiguate the above cases (IDL identifiers cannot start with the underscore character). The use of an unique prefix/suffix is ruled out to keep the mapping of identifiers as close to the IDL definition as possible but is the only completely correct solution; alternatives subject to the same criticism are the introduction of a mangeling scheme. A similar problem arises with the use of Sather keywords in IDL definitions. In the C++ mapping C++ keywords are prefixed by the underscore character.

To 'solve' the problem we attribute the detection of possible ambiguities to the IDL to Sather compiler. The implementor can then react by placing '#pragma' instructions into the IDL source file to force names for certain identifiers.

Sather keywords are mapped verbatim but suffixed by an underscore. Note the treatment of keywords is case sensitive in Sather, so that keywords differing in case (i.e. not actual keywords) may be used as identifiers in the IDL source file (the mapping does not suffix the identifier with an underscore).

4.9 Sather suport for CORBA objects – the Sather `CORBA_<XXX>` classes

The OMG IDL language mappings defer ORB functionality to a CORBA IDL module. The module is specified in IDL. Its implementation must be provided by the specific CORBA product used. In the language mappings these interfaces are mapped to Sather `CORBA_<XXX>'` classes which provide an interface to the vendor's implementation. Adaptors based on Sather external C classes have to be provided to interface to the product, often implemented in C or C++. These adaptor classes will necessarily be vendor dependant.

4.10 Mapping for interfaces

An OMG IDL interface defines a type. It is represented in Sather by an abstract class of the same name scoped by the module name containing the interface definition, if there is one.

Each operation defined within the IDL interface is mapped to a Sather method in the abstract class of the same name and signature. For details refer to the section below on mapping operations to Sather methods.

Each attribute is mapped to two accessor routines, a writer and a reader routine for the attribute, for details refer to the section on mapping attributes.

Types defined inside the IDL interface are represented by their corresponding mapping 'outside' the class. They are scoped by the interface's scoped name, followed by an underscore followed by the name of the type.

Note, IDL does not support nested interfaces but does support the definition of an exceptions, a structure, a union, and an enumerated type within the interface definition. Sather does not allow for nested classes. The notion of being defined inside an IDL interface has to be mimiced by name scoping separated by underscore.

Each IDL interface, respectively, each corresponding Sather interface class must be implemented by a concrete Sather class. This implementation is purposely left unspecified to allow for different implementation alternatives. See the section on object implementation for more details.

Each IDL Sather class, i.e. each abstract Sather class defining the Sather view of an IDL interface, inherits from a common super type $CORBA_OBJECT that defines methods common to all CORBA objects. A corresponding implementation class – `class CORBA_OBJECT` – is provided, which implements the common behavior. It must be included or reimplemented by the object implementations.

Formally OMG IDL interfaces are mapped to Sather as follows :

```
//IDL

     interface <interface identifier> {

          <structured type declaration>
          <exception type declaration>
          <constant declaration>
          <attribute declaration>
          <operation declaration>
     }

-- Sather

    class <STRUCTURED TYPE IDENTIFIER> is ... end;
    class <EXCEPTION TYPE IDENTIFIER> is ... end;
    partial class <INTERFACE IDENTIFIER> is
         <constant identifier> :<CONSTANT TYPE>:=<constant value>
    end;

    abstract class $<INTERFACE IDENTIFIER> < $CORBA_OBJECT is
        <attribute identifier> : <ATTRIBUTE TYPE>
        <attribute identifier>(a : <ATTRIBUTE TYPE>) <operation declaration>
    end;
```

An example has already been given in the previous section on scoping.

4.11 Mapping for multiple inheritance

OMG IDL supports multiple inheritance which can be directly expressed by Sather's multiple inheritance features. Note, IDL does not allow the overwriting of methods from base classes in derived classes. Formally the mapping is as follows:

```
 //IDL
```

```
    interface <interface identifier1> {...}
    interface <interface identfier2> {...}

    interface <interface identifier3> :
         <interface identifier1>,
         <interface identifier2> {...}
    }

-- Sather

    abstract class $<INTERFACE IDENTIFIER1> < $CORBA_OBJECT is ... end;
    abstract class $<INTERFACE IDENTIFIER2> < $CORBA_OBJECT is ... end;
    abstract class $<INTERFACE IDENTIFIER3> < $<INTERFACE IDENTIFIER1>,
                                              $<INTERFACE IDENTIFIER2> is
         ...
    end;
```

4.12 Mapping for operations

Operations from IDL interface definitions are mapped directly to Sather methods in the corresponding abstract class. Argument modes are mapped to the equivalent modes in Sather.

IDL	Sather
`in`	'default'
`out`	`out`
`inout`	`inout`

An IDL operation

```
interface <interface identifier> {
     <return type> <operation identifier>(<mode><parameter type>
     <identifier>,...,      <mode> <parameter type> <identifier>)
}
```

is mapped to a method signature in an abstract class representing the IDL interface containing the operation.

```
abstract class <IDENTIFIER> is
     <operation identifier>(<mode> <identifier> : <parameter type>, ...,
          <mode> <identifier> <parameter type>) : <return type>
          is...end;
end;
```

Parameter modes are mapped according to the above table.

In previous sections we have already given many examples for the mapping of operations which we wont repeat here.

OMG IDL provides an operation attribute 'oneway' used to indicate what operation semantic the communication service must provide. It specifies a best–effort semantic for the operation invocation. This means the operation will be invoked at most once.

The attribute indicates to the IDL compiler what code needs to be generated for stubs and skeletons, it is not reflected in the Sather mapping.

4.13 Mapping for attributes

An IDL attribute is mapped to two accessor functions defined in an abstract class, one returning the attribute and the other one setting the attribute. OMG IDL readonly attributes are mapped to a single accessor function. Formally this is expressed as follows :

```
// IDL

     interface <interface identifier> {
          attribute <attribute type> <attr identifier 1>
          readonly attribute  <attribute type> <attr identifier 2>
     }


-- Sather

     abstract class $<INTERFACE IDENTIFIER> < $CORBA_OBJECT is
          <attr identifier 1> : <attribute type>
          <attr identifier 1>(a : <attribute type>)
          <attr identifier 2> : <attribute type>
     end;
```

```
    -- in module scope database
    interface dbm {
         readonly attribute long last_optimized;           ...
    }

    abstract class $DATABASE_DBM < $CORBA_OBJECT is
        last_optimized : CORBA_LONG;
    end;
```

Note, only one accessor routine is provided, which reads the attribute, since the attribute is defined readonly in the IDL interface.

4.14 Mapping for constants

OMG IDL allows constants to be defined at three levels:

- at global scope
- at module scope
- at interface scope

OMG IDL constants are mapped directly to Sather constants. For each of the three scoping levels at which constants may be defined a partial class is provided that declares the constants. The partial class is named after the interface, or the module it appears in, i.e after the respective scoped name. Constants at global scope are put into a class named – `partial class IDL_GLOBAL_CONST` – . All these constant declaring classes are suffixed by '`_CONST`' to allow for the existence of an implementation classes to carry the name of the IDL interface which would otherwise be taken by the partial class defining the constants.

Note partial classes are not absolutely necessary regular classes are equally suitable for declaring constants. Formally the mapping is as follows:

```
//IDL

// in any of the three scopes, carrying the name <identifer>
const <type>  <constant identifier> = <value>;



-- Sather

    partial class <IDENTIFIER>_CONST is
         const <constant identifier> :
         CORBA_<type> := <value>;
    end;
```

An OMG IDL constant can only be of basic type.
An example has already been given in the section on scoping.

OMG IDL allows for basic operations ($+$, $-$,...) to be specified in the constant declarations. These operations are evaluated statically by the IDL–to–Sather compiler according to the rules specified for these operations in [Gro95a].

4.15 Mapping for basic data types

All OMG IDL basic data types are mapped to Sather classes implementing the specified behavior. To present a coherent framework all OMG IDL basic types are named `CORBA_<basic type>`. These types can be converted internally to other 'similar' Sather types, i.e. `CORBA_FLOAT` can be converted to Sather type `FLT`.

The `CORBA_<basic types>` have been provided since not all of the specified OMG IDL basic types are also required Sather types, i.e. required for any Sather implementation. Furthermore the C–types defined for the external C classes in Sather are platform dependent and can therefore not serve as `CORBA_<basic types>`.
The following table summarizes the mapping for basic types.

For OMG types any, string, and enum see discussion below.

4.16 Structured Data Types

OMG IDL offers three structured data types: enumerated types, structure types and discriminated unions. Their mapping to Sather are described below.

4.16.1 Mapping for OMG IDL structs

An IDL struct maps to a Sather class containing one attribute for each field in the structure. There are two constructors for each 'struct class'. One taking no arguments, initializing the struct to default values and the other one taking the fields of the structure as arguments, and initializing the structure accordingly. Formally this amounts to:

//IDL	–Sather
IEEE single precision: `float`	`CORBA_FLOAT`
IEEE double precision: `double`	`CORBA_DOUBLE`
range: -2^{31} ... 2^{31-1}: `long`	`CORBA_LONG`
range: 0...2^{32-1} : `unsigned long`	`CORBA_UNSIGNED_LONG`
range: -2^{15} ... 2^{15-1} : `short`	`CORBA_SHORT`
range: 0 ... 2^{16-1} : `unsigned short`	`CORBA_UNSIGNED_SHORT`
Iso 8859.1 and ASCII (8 bit) : `char`	`CORBA_CHAR`
`boolean`	`CORBA_BOOL`
8–bit quantity, does not undergo	`CORBA_OCTET`
chnages when transmitted `any`	`$CORBA_OBJECT`

```
//IDL

    struct <struct identifier>{
         <basic type> <identifier>
         <structured type> <identifier>
         <template type> <identifier>
    }


-- Sather

     class <STRUCT IDENTIFIER> is
          attr <identifier> : <basic type>;
          attr <identifier> : <structured type identifier>
          attr <identifier> : <template type identifier>

          create : SAME is return new end;
          create( 'arguments for all attributes' ) : SAME is
               o ::= new;
               o. 'initializations for all arguments';
               return o;
          end;
     end;
```

In the database example the type of a key and the associated information is defined by a struct :

```
//IDL

     struct datum {
          string field;
          long dbsize;
     };
```

This translates to Sather as follows :

```
class DATUM is
     attr field : CORBA_STRING;
     attr dbsize : CORBA_LONG;
     create : SAME is return new; end;
     create(f : CORBA_STRING, d : CORBA_LONG) : SAME is
          o ::= new;
          o.field := f;
          o.dbsize := d;
     end;
end;
```

4.16.2 Mapping for discriminated unions

OMG IDL provides a union type. A selector determines which kind of value is actually stored in the union. The union construct is motivated as a space saving type [Gro95a].

```
//IDL

     union <identifier> switch <tag type>
          case <value of type 'tag type'> : <type 1> <identifier 1>;
          case <value of type 'tag type'> : <type 2> <identifier 2>;
          default     : <type 3> <identifier 3>;
     };
```

The tag type must be one of integer, char, boolean, or enum.

```
- Sather

     class <IDENTIFIER> is
          private attr v : $CORBA_OBJECT;
          create(o : $CORBA_OBEJECT) : SAME is
               obj ::= new;
               typecase o
               when <type 1> then obj.v := o;
               when <type 2> then obj.v := o;
               when <type 3> then obj.v := o;
               else raise #CORBA_UNKNOWN_UNION_TYPE;
               end;
               return obj;
          end;

          -- accessor
          <identifier 1> : $CORBA_OBJECT is return v; end;
          <identifier 2> : $CORBA_OBJECT is return v; end;
          <identifier 3> : $CORBA_OBJECT is return v; end;
     end;
```

Sather provides a typecase construct which can be used to select the current type of the union. The selector provided in the IDL definition is not needed in the Sather mapping. It is not used elsewhere in IDL source, except in the C and C++ mappings to select and remember the value stored in the union at run time. There it is part of the object state representing the union. The following example shows a use of the union in Sather.

```
-- Sather
     u:<IDENT>:=#(value);

     -- an exception is raised if 'value' is of incorrect type
     l ::= u.<ident 1>;

     -- typecase variable must be a local

     typecase l
     when <type 1> then #OUT+"It is type 1.\n";
     when <type 2> then #OUT+"It is type 2.\n";
     when <type 3> then #OUT+"It is type 3.\n";
     else #OUT+"Unknown type.\n";   -- will not happen !
     end;
```

The C and C++ mappings provide an explicit mapping for the IDL union construct. The explicit mapping implements a selector function which must be used to select the correct value at run time. The selector attribute is set whenever the union is initialized with a value. Accessing a union type with an accessor function not appropriate for the union type yields undefined behavior.

In the proposed IDL–Sather mapping any accessor can be used to access the union's value. The result variable must, however, be of type `$CORBA_OBJECT`. With the use of the typecase statement the current type of the object can be further narrowed down.

4.17 Mapping for OMG IDL enumeration types

An IDL enumeration type maps to a Sather class that contains the enumeration elements as 'enumerated' constants in the order given in the IDL definition. Sather assures that the order constraint imposed by the IDL definition is reflected in its implementation. Formally:

```
//IDL
     enum <identifier> {<ident_1>, ..., <ident_n>};

-- Sather
     class <IDENTIFIER> is
        const <ident_1>, ..., <ident_n>;
     end;
```

Example :

```
//IDL
     enum flags {INSERT, REPLACE};

-- Sather
     partial class FLAGS is      const INSERT, REPLACE;      end;
```

4.18 Template data types

OMG IDL defines template data types as sequences of type, T. Sequences can be of fixed size, i.e. they are bound, or of unspecified size, unbound sequences. Sequences are defined in OMG IDL as arrays of given type with the following two characteristics:

- they have a maximum size
- they have a variable length (less or equal to max. size)

The implementation has to assure that the length of a sequence does never exceed the maximum size of a sequence. Furthermore it must be possible to query the length of a sequence, as well. as its maximum capacity.

IDL contains strings as separate types due the often available language support and their more efficient implementation [Gro95a] Strings also come in bound as well as unbound form.

Moreover, IDL defines arrays, possibly multidimensional. Arrays have to be of fixed, at compile time specified, size.

4.18.1 Mapping for Sequences

OMG IDL defines a sequence type as a one dimensional array with two characteristics: a maximum size, determined at compile time, and a length, determined at run time.

OMG IDL distinguishes, however, between bounded and unbound sequences. For bound sequences the maximum size of the sequence is specified in the IDL definition. The length of a bound sequence has to be set in a language dependant manner prior to passing it as an argument to an operation. After receiving a sequence as the result of an operation invocation the length of the returned sequence has been set. It must be accessible in a language dependent manner.

Unbound sequences leave the maximum size unspecified. Maximum size, length and a buffer to hold the data items of the sequence have to be set in a language dependent manner prior to passing a sequence as argument.

When receiving an unbound sequence as return result these quantities have been set.

In Sather, OMG IDL sequences are represented as `CORBA_BND_SEQ{T,L}` and `CORBA_UNBND_SEQ{T}` with the OMG specified behavior ('T' is the type of the sequence elements and 'L' is a type specifying the maximum length.

```
// IDL
     bound sequence
     sequence< <type>, <length> > <identifier>;

-- Sather
     class BOUND_<IDENTIFIER> is
          const length:INT:=<length>;
     end;

     <identifier> : CORBA_BND_SEQ{<type>, BOUND_<IDENTIFIER>};

// IDL
     unbounded sequence      sequence< <type> > <identifer>;

-- Sather
     <identifier> : CORBA_UNBND_SEQ{<type>};
```

The implementation of CORBA_BND_SEQ{T,L} makes use of the constant 'length' to allocate enough space for the buffer containing the elements of the sequence.

OMG IDL allows sequences of type sequence. The same can easily be achieved by the Sather mapping, simply by substituting the right parameter by the sequence type.

4.18.2 Mapping for strings

OMG IDL singles out strings due to often available language support for these types. In essence a string is a sequence of characters (//IDL : sequence<char>). IDL strings come in bounded and unbounded form as explained above.

In Sather two classes provide the demanded string functionality. CORBA_BND_STRING{L} and CORBA_UNBND_STRING. Both classes are analogous to the above classes. A specific CORBA_BOUND_<IDENTIFIER> is constructed for bounded strings.

4.18.3 Mapping for arrays

OMG IDL supports one, as well as, multidimensional arrays. All dimensions are of fixed length, i.e. their size are specified at compile time. IDL arrays map to Sather CORBA arrays (i.e. CORBA_ARRAY{T}).

```
//IDL
     <type> <identifier>[<length>];

-- Sather
     class BOUND_<IDENTIFIER> is
          const length:INT:=<length>;
     end;

     <identifier> : CORBA_ARRAY{<type>, BOUND_<IDENTIFIER>};
```

The implementation of 'CORBA_ARRAY' makes use of the constant 'length' in 'BOUND_<IDENTIFIER> to set the length of the array.

In multidimensional arrays the class names needs to be additionally suffixed by the number of dimension. We count from left to right starting with one.

```
//IDL
     <type> <identifier>[<length1>][<length2>];

-- Sather
     class BOUND_<IDENTIFIER>_1 is
          const length:INT:=<length1>;
     end;

     class BOUND_<IDENTIFIER>_2 is
          const length:INT:=<length2>;
     end;

     <identifier> : CORBA_ARRAY{CORBA_ARRAY{<type>, BOUND_<IDENTIFIER>_2},
                BOUND_<IDENTIFIER_1>}
```

4.19 Mapping for type Declarations (typedef)

OMG IDL defines a typedef construct for naming data types. Unlike C and C++, Sather does not provide an equivalent language construct.

Each type declaration in IDL defines therefore a proper type in Sather with the name given by the IDL definition, possibly scoped by the enclosing interface and module names.

In the mapping the following cases need to be distinguished.

```
//IDL
     typedef <basic type> <identifier>;
-- Sather
     class <IDENTIFIER> is include basic type> end;

//IDL
     typedef <constructed type> <identifier>;
-- Sather
     class <IDENTIFIER> is <constructed type> end;
```

'`<constructed type>`' is defined according to the mappings in this report for structs, enums, and unions.

```
//IDL
     typedef <user defined type> <identifier>;

-- Sather
     abstract class $<IDENTIFIER> < $<user defined type> is ... end;
```

Inheritance is used to derive an abstract class of the declared name.

```
//IDL
     typedef <template type> <identifier>;
-- Sather
     class <IDENTIFIER> is include <template type>; end;

//IDL
     typedef <declared type> <identifer>;
```

The type '<declared type>' can be of any one of the above defined types and is mapped accordingly.

In the database example the datum could for example be declared by a typedef construct:

```
//IDL
     typedef struct entry {
          string field;
          long dbsize;
     } datum;

     typedef enum insert_fg {INSERT, REPLACE} flags;

     interface dbm {
           ...
           datum fetch(datum key);
           long store(datum key, datum content, flags flag);
```

```
        ...
    };
```

```
-- Sather

    class DATUM is
          attr field : CORBA_STRING;
          attr dbsize : CORBA_LONG;
          create : SAME is .... end;
          create(...) : SAME is .... end;
    end;

    class FLAGS is const INSERT, REPLACE; end;

    abstract class $DBM is
         fetch(key : DATUM, content : DATUM, flag : FLAGS);
         store(key : DATUM, content : DATUM, flag : FLAGS):CORBA_LONG;
    end;
```

The type declarations containing OMG IDL arrays, strings, and template types are treated similarly. A new class of the declared name is provided which includes the bound or unbound collection type and ensures that it is parametrized as demanded in the declaration. For example :

```
//IDL
     interface dbm;
     typedef dbm datawarehouse[100];
     interface dbm { ... };

-- Sather

     class BOUND_DATAWAREHOUSE is
          const length:INT:=<length>; end;

     class DATAWAREHOUSE is
          include CORBA_ARRAY{$DBM, BOUND_DATAWAREHOUSE} create->old_create;
          create : SAME is return old_create; end;
     end;
```

The mapping for the typedef construct is somewhat awkward since there is not one coherent way to map IDL's typedef to Sather. Moreover, we assume that each typedef declares a separate type. Consider for example:

```
\\IDL
     typedef struct A {...}  structA;
     typedef struct A {...}  structA_again;
```

In Sather the two types 'STRUCTA' and 'STRUCTA_AGAIN' are distinct:

```
-- Sather
      s1 : STRUCTA;
      s2 : STRUCTA_AGAIN;
```

```
... --illegal : s1 := s2;
```

There is unfortunately no solution based on the current Sather 1.1 language specification which would simplify the mapping for typedef. The introduction of type equality in Sather would ease the mapping for IDL typedefs to Sather and would solve the above problem of 'typedeclaring' one type to multiple names in IDL. With type equality the mapping would map each typedef to an abstract class and an implementation as above. The abstract classes would represent the declared type and equality of multiply defined types would follow trivially due to the reflexivity of the type equality operation. The coherent use of abstract classes for the mapping would largely simplify the mapping, i.e. one mapping scheme would be enough.

4.20 Mapping for exceptions

OMG IDL operations may raise exceptions to indicate erroneous conditions. The Sather language offers full support for exceptions. It is therefore straight forward to map OMG's IDL exception mechanism to Sather. In IDL exceptions are defined similarly to structs. Each exception type may contain several fields of different type. In the Sather mapping an IDL exception is represented by a concrete class providing for each IDL exception field a proper attribute. The class also provides a constructor for initializing the exception attributes with values. Like any other type defined in any of the three scoping levels, Sather 'exception classes' have to be correctly scoped, as outlined in the section on scoping.

Formally IDL exceptions are mapped to Sather exceptions as follows:

```
// IDL

     exception <exception identifier>{
          <type> <identifier>;
     end;


-- Sather

     class <EXCEPTION IDENTIFIER> is
          attr <identifier> : <type>;
          create('attrs to be initialized') : SAME is ... end;
     end;
```

Note, IDL allows one to specify that a particular operation may raise an exception. It is not possibly to express this in the Sather language. The OMG IDL specification requires, however, that on the server site an exception thrown, which is not part of the optional raises clause of an operation, must be detected. Each operation, specifying a raises clause must therefore be called through a wrapper function which test whether one of the specified exceptions has been raised. Note, only user defined exceptions can be specified in the exception clause of an operation.

An example can be found in a later section on the server site mapping.

4.21 Mapping for type any

The OMG IDL type 'any' can represent any OMG IDL type. It is represented in Sather by the abstract class `$CORBA_OBJECT`. All IDL Sather classes must subtype from this class. All `CORBA_<basic`

type> classes defined in the CORBA library must also subtype from `$CORBA_OBJECT` since the any type may also represent an IDL basic type.

4.22 Server–side mapping

The server–side mapping refers to portability constraints for object implementations. It is the part of the OMG CORBA specification least worked out [Gro95a]. Many decisions are left to the CORBA implementor. As a consequence smooth portability of object implementations can not be expected. The OMG promises a more elaborate standard in future revisions [Gro95a].

To define an implementation in Sather, one must provide a concrete class which implements the behavior specified in the IDL interface. I.e. an implementation for the Sather interface class. Several different design patterns can be used to relate the Sather interface class with its implementation. The OMG illustrates two design patterns for its C++ language binding: implementation by inheritance and implementation by delegation. Other approaches are possible.

The server–side mapping requires that user defined exceptions thrown by operations but not listed in a function header's raises clause be detected. In C++ this is easily possibly by qualifying the method signature with 'throw(exception–type)'. Sather, however, does not provide an equivalent construct. To enforce this constraint must we explicitly test whether the condition is satisfied. A wrapper around the actual call does the job. Example:

```
\\IDL

     interface dbm{
          ...
          exception DISK_FULL (...);
          exception READ_ERROR (...);           ...
          long store(datum key, datum content, flags flag)
               raises(DISK_FULL, READ_ERROR);
          ...
     }

-- Sather

     class READ_ERROR is ... end;
     class DISK_FULL is ... end;
     abstract class $DBM < $CORBA_OBJECT is           ...
          private store_ex_less(datum key, datum content, flags flag);
          store(datum key, datum content, flags flag);
          ...
     end;
```

The method '`store(...)`' is a wrapper that makes a call to '`store_ex_less(...)`' encapsulated in a protect statement testing whether one of the listed exceptions has been thrown.

In the concrete class implementing the Sather interface class the method '`store`' has to be implemented in the following way :

```
     ...
     store(datum key, datum content, flags flag) is
```

```
protect
      store_ex_less(datum key, datum content, flags flag);
when DISK_FULL then
when READ_ERROR then
when $EXCEPTION
else raise
      #UNKNOWN_EXCEPTION;
end;
```

'`$EXCEPTION`' is an abstract class provided by the Sather CORBA library. It defines the common behavior for all CORBA system exceptions. Note we must also test whether a system exception has been thrown, since CORBA system exceptions may not be listed in raises clauses.

Moreover, the server–side–mapping requires that within each method implementing an IDL interface operation the following two operations exist. The operation 'self' returns a reference to the target object. The operation 'boa' returns a reference to the basic object adaptor. '`Self`' is per default available in Sather. 'boa' is realized as part of the common code included in every class.

We complete this section by briefly discussing the two design patterns proposed in [Gro95a] for object implementations. They are implementation by inheritance, i.e. a concrete class is derived from the Sather interface class, and implementation by delegation, i.e. the Sather interface class is passed as parameter to a parametrized class and implemented therein.

These are merely suggestions, other approaches are certainly possible. Neither one is standardized.

We use the following example to illustrate the implementation alternatives:

```
\\IDL
     interface dbm {
          ...
           dbm open(string file, flags flag, long mode);
           void close();
          ...
     };
```

The following Sather interface class provides the Sather view of the IDL interface:

```
-- Sather

     abstract class $DBM < $CORBA_OBJECT is
           ...
          open(file:CORBA_STRING, flag: $FLAGS, mode : CORBA_LONGB):$DBM;
          close;
          ...
     end;
```

Implementation by inheritance On the server–side a skeleton class is provided which derives from '`$DBM`' adding additional server specific methods. The implementor must then provide a concrete class which implements these methods.

The IDL compiler generates the following skeleton class:

```
-- Sather
     abstract class $DBM_SK < $DBM is
          -- server side implementation details
```

```
        ...
    end;
```

The programmer needs to provide a concrete class which implements the demanded behavior. A 'skeleton' of this class could be generated by the compiler.

```
-- Sather
    class DBM < $DBM_SK is
        -- include common behavior
        include BOA;
        include SERVER_SIDE;
        ...
        open(file:CORBA_STRING,flag:$FLAGS,mode:CORBA_LONGB):$DBM is
            ...
        end;

         close is ... end;
        ...
    end;
```

Implementation by delegation Implementation by inheritance may not always be an ideal solution. For one it imposes an inheritance hierarchy on the program with the resulting overhead of dispatching for all method invocations. Furthermore, it might not always be possible to derive an implementation by inheritance. If for example legacy code is used to implement an IDL interface inheritance from a base class will only be possible through reimplementation. This latter argument is, however, somewhat derived for a language like Sather, since an appropriate adaptor class is quickly designed.

Implementation by delegation provides wrappers that delegate upcalls to the actual implementation. The class implementing these adaptors are referred to as tie classes in OMG terminology. [...].

5 Conclusion

We have presented and compared several distributed object oriented communication infrastructures. The comparison focused on how the individual approaches achieve programming language inteoperability. This is an important aspect, since it will ease component software development and integration. Furthermore, we have emphasized on CORBA, since it has a clear specification and is widely accepted. We think that it provides a well established reference point to compare other approaches against.

In the second part of our work we showed how to integrate a new language into the CORBA framework. We have defined a complete IDL–to–Sather language mapping. In contrast to other languages we conclude that Sather is ideally suited for integration into CORBA. The rather lengthy but relatively straight forward mapping of IDL to Sather shows that integrating a new language into CORBA is straight forward.

References

[BJ97] M. Spreitzer B. Janssen. Ilu reference manual. Technical report, Xerox Parc, 1997.

[Bro96] K. Brockschmidt. *Inside OLE*. Microsoft Press, 1996.

[Cha96] D. Chappell. Dce and objects. Technical report, (unknow), 1996.

[Gro95a] Object Management Group. The common object request broker architecture and specification. revision 2.0. Technical report, OMG, 1995.

[Gro95b] Object Management Group. The corba common object service specification. Technical report, OMG, 1995.

[OSF96] OSF. Distributed computing environment. Technical report, Open Software Foundation, 1996.

[SO96] D. P. Stoutamire and Stephen M. Omohundro. The sather 1.1. specification. Technical report, ICSI, 1996.

[TC93] TINA-C. Dpe phase 0.1 specification. Technical report, Telecommunication Information Networking Architecture Consortium, Dezember 1993.