Specification of the OMG IDL to Sather mapping

Technical Report, May 1997, ICSI (TR–97–XX–XX)
DRAFT

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

Abstract

The Common Object Request Broker Architecture (CORBA) aims at providing transparency and interoperability across distributed, heterogeneous computing environments. Furthermore, it offers an object oriented architectural framework for integrating different programming paradigms and languages.

CORBA achieves this by having each component of the architecture specify its interface in an universal, non–algorithmic, declarative interface definition language (IDL).

Interoperability is gained by mapping these component interfaces to the client programming language. Remote method invocations, object activation and object location are handled transparently by the CORBA middleware.

After briefly introducing the Sather object oriented programming language, the mapping of OMG IDL to Sather is formally laid out and discussed. The mapping is motivated by examples throughout the specification, illustrating its use.

It will become clear that Sather constitutes the ideal language for the mapping. This is partly because all IDL constructs are naturally expressible with the idioms and features of Sather and partly due to Sather's object oriented features which permit a clean integration of the language into the CORBA object model.

Contents

1 Introduction

This work proposes a language mapping of the Sather object oriented programming language [?], [?] to the Object Management Group's Interface Definition Language (IDL) [?].

The key features of the Sather object oriented programming language are summarized in the following section. The remainder of the report follows the outline and guidelines put forth by the Object Management Group (OMG) for language mappings. Section 1.2 summarized the language mapping. Section 1.3 outlines the difference in terminology used by the OMG and the Sather community. In this report we will use either terminology, however, each one in its proper context; i.e. OMG terms for referring to OMG IDL and Sather terminology within the Sather context. Section **??** presents the design rational of the proposed language mapping. Sections **??**, **??** and **??** presents the mapping of IDL to Sather. In section **??** we briefly show how the mapping deviates for alternative Sather dialects. The mapping has been designed such that the deviation would be small.

1.1 The Sather Language

Sather is an object oriented language designed to be simple, efficient, safe, flexible and non–proprietary. One way of placing it in the "space of languages" is to say that it aims to be as efficient as C, C++, or Fortran, as elegant as and safer than Sather, and support higher–order functions and iteration abstraction as well as Common Lisp, CLU or Scheme [?].

Sather has garbage collection, statically–checked strong (contravariant) typing, multiple inheritance, separate implementation and type inheritance, parameterized classes, dynamic dispatch, iteration abstraction, higher–order routines and iterators, exception handling, assertions, preconditions, postconditions, and class invariants. Sather code can be compiled into C code and can efficiently link with object files of other languages. pSather, the parallel and distributed extension, presents a shared memory abstraction to the programmer while allowing explicit placement of data and threads.

Sather has a very unrestrictive license aimed at encouraging contribution to the public library without precluding the use of Sather for proprietary projects.

More information about the Sather language can be found on the web pages `www.icsi.berkeley.edu/ Sather` and `www.ipd.uni--karlsruhe/"Sather`.

As will be seen in the language mapping part of this document, Sather is the ideal language for implementing OMG's IDL. Interfaces in IDL have a direct correspondence with Sather abstract classes, multiple inheritance is expressible in Sather without restriction, operation signatures carry directly from IDL to Sather. Especially the argument modes in Sather serve helpful in expressing IDL argument modes. Sather is garbage collected which renders most of the explicit reference counting for the C++ mapping absent. After presenting the detailed mapping we will take up this point and give a more profound discussion.

//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`

Several dialects of the Sather language are currently implemented. Namely [?] and [?]. Their essential differences are described in [?]. The presented language mapping tries to be general enough to embrace these dialects. The mapping is syntactically oriented along the lines of Sather presented in [?]. Whenever something should not be expressible with one or the other language an alternate mapping is described in section ??. We chose this bifurcation since currently no convergence of the different dialects seem to be in sight.

1.2 Overview of IDL st Sather Mapping

The following table summarizes the IDL to Sather mapping, more formally specified, in the following sections. It should serve as an overview for reference. It is organized according to the subsections in the succeeding section.

1.3 Terminology

The following table summarizes the difference in terminology 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 [?]. 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

//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

identifiers used inside modules and interfaces, as of being in *module–scope* and *interface–scope*.

We use an 'intuitive' formalism to illustrate the IDL to Sather mapping. Our goal is to precisely capture the mapping but not to overwhelm the reader with formality. The OMG [?] 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 capture the mapping is better suited for its presentation.

The formalism we have adopted describes the IDL construct in a programming–construct–like manner. All terminals, such as for example, 'interface' are presented as such without extra markings. Non–terminals are enclosed in brackets '< ... >'. They usually describe more complex entities such as entire struct declarations (e.g. '<structured declaration>'). The more precise break–down of the non–terminal is presented in its proper section in the mapping.

In the Sather mapping we take reference to parts of the description such as their name or type. The Sather representation is described in the same manner.

The use of lower case for non–terminals should indicate that any case of the identifier is allowed. In Sather we capitalize these terms to indicate that they need to be capitalized.used.

2 Design goals

The OMG IDL to Sather mapping allows to integrate Sather objects in

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

2.0.1 Design Decisions

Conformity The mapping proposed here makes every effort not to limit the implementation freedoms of ORB developers. For each IDL and CORBA construct, this mapping explains the syntax and semantics of using the construct from Sather. A client or server program is said to conform to this mapping if it uses the constructs as described here.

A CORBA implementation is said to conform to this mapping if it correctly executes any conforming client or server program. A conforming client or server program is therefore portable across all conforming implementations.

2.1 RFP–mandated Requirements

2.1.1 Mapping for Entire IDL Language

This proposal maps the entire OMG IDL language as defined in CORBA 2.0 [OMG 95].

2.1.2 Consistent Style

The style of this mapping is consistent with the language idioms of the Sather object oriented programming language.

2.1.3 Justification of IDL, CORBA, Object Model Extensions

No extensions to IDL, CORBA, or the object model are proposed. Sather can fully implement the IDL interfaces and interoperate with CORBA 2.0 compliant ORBs.

2.1.4 No Implementation Descriptions

The mapping has been carefully designed by avoiding details in the mapping that would constrain implementations while still allowing clients to be fully source compatible with any conforming implementation.

2.2 Evaluation Criteria

2.2.1 Reliability

The proposed language mapping uses the features of the Sather object oriented programming language to naturally represent the IDL definitions. The mapping does not impact the reliability of CORBA implementations or clients and servers build on the ORB.

2.2.2 Portability

Currently two implementations of Sather are available, as described in the section on the Sather language above. The mapping embraces both dialects.

3 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. An concrete example of the mapping is given to better illustrate the key issues. The mapping specification is oriented along the guide–lines put forth by the OMG [OMG 95] for language mappings in general. We tried to be somewhat more formal in the below presentation.

3.1 Syntactic considerations

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

3.1.1 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 IDL identifier that collides with an Sather reserved word must have a prefix of "IDL_" before the identifier.

See appendix A for a list of reserved words.

3.1.2 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').

Description	IDL Escape Sequence	ISO 646 Octal	Sather Lexical Mapping
newline	\n	012	\n
horizontal	tab \t	011	\t
vertical	tab \v	013	\v
backspace	\b	010	\b
carriage return	\r	015	\r
form feed	\f	014	\f
alert	\a	007	\a
backslash	\\	134	\\
question mark	\?	077	?
single quote	'	047	'
double quote	"	042	"
octal number	\ooo	ooo	0o
hex number	\xhh		0x

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), a "." (decimal point), the fraction part (or "0" if the fraction part is missing), 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 supplies lexical equivalent expression for the escape sequences defined.

Note that an ASCII code equivalent form is always allowed, but not recommended in concatenated sequences.

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 concatenation mechanism, since Sather does not allow string literal expressions to extend pass a source line of text.

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 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.

3.2 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 scoped 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 identifiers so that using the same name for a constant and an interface produces compilation errors [OMG 95].

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 xxx 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 our example, some additions have already 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;
```

Not all uses of '::' in OMG IDL are resolved to the use of '_' in Sather. The programmer has to pay close attention to the scoping level referred to in IDL. E.g. [...]

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 rule taken up above has to be used to 'mimic' the notion of a module. Modules are used in IDL to introduce namespace [OMG 95], no other functionality is associated with them.

3.2.1 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 mangling 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).

3.2.2 Sather support for CORBA objects

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 dependent.

3.3 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 abstract 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;
```

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

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

3.4 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;
```

3.5 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.

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.

3.6 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 as 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 the database example we can add a readonly attribute denoting the date the database has last been optimized, i.e. indicies, data clusters introduced by system administrator.

```
-- 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.

3.7 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 <identifier>      const <type>
     <constant identifier> = <value>;


-- Sather

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

//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`

```
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 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 [OMG 95].

3.8 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 type, 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.

3.9 Structured Data Types

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

3.9.1 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

    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;
```

3.9.2 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 [OMG 95].

```
//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 code 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, C++, as well as, the proposed Java mapping [Orbix 96] 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.

3.10 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;
```

3.11 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 [OMG 95]. 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.

3.12 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 dependent 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_SEQT,L` and `CORBA_UNBND_SEQT` 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> > <identifier>;

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

The implementation of `CORBA_BND_SEQT,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.

3.13 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_STRINGL` and `CORBA_UNBND_STRING`. Both classes are analogous to the above classes. A specific `CORBA_BOUND_<IDENTIFIER>` is constructed for bounded strings.

3.14 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_ARRAYT).

```
//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>}
```

3.15 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> <identifier>;
```

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.

3.16 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.

3.17 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.

3.18 Files output by IDL – Sather compiler

Sather does not put any restrictions on input file and output file organization and names. Multiple classes may be defined per file. The IDL–Sather compiler can therefore use a strategy best suited for the ORB/CORBA implementation it is tailored for.

Sather provides a '.module' file concept to ease specification of compiler options and to tie the compilation of different files/classes together. The IDL compiler should generate such a file to tie the compilation of the different output files together with the client and server code implementations.

4 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 [OMG

95]. Many decisions are left to the implementor. As a consequence smooth portability of object implementations can not be expected. The OMG promises a more elaborate standard in future revisions [OMG 95].

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 [OMG 95] for object implementations. They are implementation by inheritance. I.e. a concrete class is derived from the Sather interface class. And implementation by delegation.

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 Mapping for Pseudo–Objects

This section is still somewhat shaky!!

The CORBA standard defines so called pseudo interfaces. These are interfaces defined in IDL which do, however, not strictly adhere to the defined language mappings. An instance of a pseudo interface is termed pseudo object. Several restrictions apply to these pseudo objects :

- a pseudo object does not inherit from CORBA_OBJECT
- a pseudo object is not registered with the ORB
- a pseudo object cannot be specified as parameter to an operation invocation on an ordinary object
- a pseudo object is not accessible using the dynamic invocation interface
- a pseud object has no definition in the interface repository
- no proxies are created for pseudo objects

The implementation of these interfaces is provided by the specific CORBA implementations. Their behavior is specified by a pseudo IDL interface.

The IDL to Sather mapping provides an abstract class for each pseudo interface. The abstract class is implemented by a vendor specific adaptor handling calls and callbacks via Sather external C classes.
For example:

```
// IDL Environment interface in CORBA namespace

     pseudo interface Environment  {
          attribute exception exception;
          void clear();
     };
```

---

maps in Sather to

```
-- Sather

     abstract class $CORBA_ENVIRONMENT is
          exception:EXCEPTION;
          exception(e:EXCEPTION);
          clear;
     end;

     class CORBA_ENVIRONMENT is
          attr exception : EXCEPTION;
          create : SAME is ... end;
          clear is ... end;
     end;

     external C class CORBA_ENVIRONMENT_ADAPTER
          -- interfaces in a vendor specific way to the library functions
          -- provided by the vendor's CORBA implementation.
          ...
     end;
```

Note the adapter does not incur a performance loss, since its methods may be inlined. Many ORB's are, however, implemented in C++, to allow correct linking with the Sather code we have to provide 'extern C' interfaces for the classes implementing the pseudo interfaces in the vendor code. This extra indirection is not inlineable but we do not expect a great performance loss.

To keep with OMG specification tradition we present all pseudo IDL interfaces and the corresponding Sather classes in the appendix. Adaptors and external C interfaces are straight forward but depend to some extend on the vendor's implementation.

5.1 Mapping Rules

5.2 Pseudo–Object Semantics

5.3 Environment

```
-- Sather

     abstract class $ENVIRONMENT is
          exception(e : EXCEPTION);
          exception : EXCEPTION;
          clear;
     end;
```

5.4 NamedValue

```
-- Sather
     abstract class $NAMEDVALUE is
           name : IDENTIFIER;
           value : $CORBA_OBJECT;
           flags : FLAGS;
     end;
```

5.5 NVList

```
-- Sather
     abstract class $NVLIST is
          count : CORBA_UNSIGNED_LONG;
          add(flags : FLAGS) : NAMEDVALUE;
          add_item(item_name : IDENTIFIER, flags : FLAGS) : NAMEDVALUE;
          add_value(item_name : IDENTIFIER,
                    val : $CORBA_OBJECT,
                    flags : FLAGS) : NAMEDVALUE;
          item(in unsigned long index) raises(Bounds) : NAMEDVALUE;
          remove(index : CORBA_UNSIGNED_LONG) : STATUS;
     end;
```

5.6 Request

```
-- Sather
     abstract class $EXCEPTIONLIST is
          count : CORBA_UNSIGNED_LONG;
          add(exc : TYPECODE);
          item(index : CORBA_UNSIGNED_LONG) : TYPECODE;
          remove(index : CORBA_UNSIGNED_LONG) : STATUS;
     end;

     abstract class $CONTEXT_LIST is
          count : CORBA_UNSIGNED_LONG;
          add(ctxt : CORBA_STRING);
```

```
        item(index : CORBA_UNSIGNED_LONG) : CORBA_STRING;
        remove(index : CORBA_UNSIGNED_LONG) : STATUS;
    end;

    abstract class $REQUEST is
        target : CORBA_OBJECT;
        operation : IDENTIFIER;
        arguments;: NVLIST;
        result : NAMEDVALUE;
        env : ENVIRONMENT;
        context : CONTEXTLIST;
        ctx : CONTEXT;
        ctx(ctx : CONTEXT);
        invoke : STATUS;
        send_oneway : STATUS;
        send_deferred : STATUS;
        get_response : STATUS;
        poll_response : CORBA_BOOL;
    end;
```

5.7 Context

```
-- Sather
    abstract class CONTEXT is
        context_name : IDENTIFIER;
        parent : CONTEXT;
        create_child(child_ctx_name : IDENTIFIER, out child_ctx : CONTEXT) : STATUS;
        set_one_value(propname : IDENTIFIER, propvalue : $CORBA_OBJECT) : STATUS;
        set_values(in NVList values) : STATUS;
        delete_values(in Identifier propname) : STATUS;
        get_values(start_scope : IDENTIFIER,
                   op_flags : FLAGS,
                   pattern : IDENTIFIER,
                   out values : NVLIST): STATUS;
    end;
```

5.8 Principal

```
-- Sather
    abstract class $PRINCIPAL is
```

```
        -- 'interface principal' has an empty body
    end;
```

5.9 TypeCode

```
-- Sather
    class TYPECODE_BOUNDS is  create : SAME is return new; end;
    class TYPECODE_BAD_KIND is  create : SAME is return new; end;
    abstract class $TYPECODE is

        // for all TypeCode kinds
        equal(tc : TYPECODE) : CORBA_BOOL;
        kind : TCKIND;

        // for tk_objref, tk_struct, tk_union, tk_enum, tk_alias, and tk_except
        RepositoryId id : REPOSITORYID;
        name : IDENTIFIER;

        // for tk_struct, tk_union, tk_enum, and tk_except
        member_count : CORBA_UNSIGNED_LONG;
        member_name(index : CORBA_UNSIGNED_LONG) : IDENTIFIER;

        // for tk_struct, tk_union, and tk_except
        member_type(index : CORBA_UNSIGNED_LONG) : TYPECODE;

        // for tk_union
        member_label(index : CORBA_UNSIGNED_LONG) : $CORBA_OBJECT;
        discriminator_type() : TYPECODE;
        long default_index() : CORBA_LONG;

        // for tk_string, tk_sequence, and tk_array
        unsigned long length

        // for tk_sequence, tk_array, and tk_alias
        TypeCode content_type() raises(BadKind);

        // deprecated interface
        long param_count : CORBA_UNSIGNED_LONG;
        parameter(index : CORBA_LONG) : $CORBA_OBJECT;
    end;
```

5.10 BOA

```
-- Sather
     abstract class $BOA is

          create(id : REFERENCEDATA,
                 intf : InterfaceDef,
                 impl :IMPLEMENTATIONDEF) : $CORBA_OBJECT;

          dispose(obj : $CORBA_OBJECT);
          REFERENCEDATA get_id(obj : $CORBA_OBJECT);
          change_implementation(obj : $CORBA_OBJECT, impl : IMPLEMENTATIONDEF);
          get_principal(obj : $CORBA_OBJECT, ev : ENVIRONMENT) : PRINCIPAL;
          impl_is_ready(impl IMPLEMENTAITIONDEF);
          deactivate_impl(impl IMPLEMENTATIONDEF);
          obj_is_ready(obj : $CORBA_OBJECT, impl : IMPLEMENTATIONDEF);
          deactivate_obj(obj : $CORBA_OBJECT);
     end;
```

5.11 ORB

```
-- Sather

     class ORB_REQUEST_SEQ is
          -- in 'interface ORB' ... typedef sequence<Request> RequestSeq;
          include CORBA_SEQUENCE{REQUEST};
     end;

     abstract class $ORB is
          object_to_string(obj : $CORBA_OBJECT) : CORBA_STRING;
          string_to_object(str : CORBA_STRING) : $CORBA_OBJECT;

          create_list(count : CORBA_LONG, out new_list : NEW_LIST) : STATUS;
          create_operation_list(oper : OPERATIONDEF, out new_list : NVLIST):STATUS;
          create_named_value(out nmval : NAMEDVALUE) : STATUS;
          create_exception_list(out exclist : EXCEPTIONLIST) : STATUS;
          create_context_list(out ctxtlist : CONTEXTLIST) : STATUS;

          get_default_context(out ctx : CONTEXT) : STATUS;
          create_environment(out new_env : ENVIRONMENT) : STATUS;
```

```
        send_multiple_requests_oneway(req : REQUESTSEQ) : STATUS;
        send_multiple_requests_deferred(req : REQUESTSEQ) : STATU;S

        poll_next_response : CORBA_BOOL;
        get_next_response(out req : REQUEST);
    end;
```

---

6 Acknowledgements

7 References

G. Goos. Sather Specification.*Technical Report 8/95*, 1995, University of Karlsruhe (TH).

Object Management Group. The Common Object Request Broker: Architecture and Specification. *Revision 2.0, July 1995.*

J., Siegel, CORBA Fundamental and Programming, *John Wiley & Sons, INC. 1996.*

D. P. Stoutamire and Stephen M. Omohundro, The Sather 1.1 Specification, *Technical Report 1995, International Computer Science Institute, Berkeley.*

D. P. Stoutamire and Wolf, Sather Differences *Technical Report 1996, International Computer Science Institute, Berkeley.*

S. Vinoski, CORBA: Integrating Diverse Applications Within Distributed heterogeneous Environments, *IEEE Communications Magazine, Vol 14, No. 2, February 1997.*

A Sather reserved words

`will be completed later`