The SBW-CORBA Gateway

Michael Hucka and Andrew Finney

{mhucka,afinney}@cds.caltech.edu
Systems Biology Workbench Development Group
ERATO Kitano Systems Biology Project
Control and Dynamical Systems, MC 107-81
California Institute of Technology, Pasadena, CA 91125, USA

http://www.cds.caltech.edu/erato

Principal Investigators: John Doyle and Hiroaki Kitano

May 31, 2002

Contents

1 Introduction

We describe the SBW-CORBA gateway. This gateway enables SBW modules to communicate with CORBA objects, and vice versa, subject to some limitations on interoperability stemming from differences in data types and framework design. We do not discuss *how* communications between SBW-based and CORBA-based applications take place; the details of inter-framework communications are left to another document.

2 Organization of the SBW-CORBA Gateway

We have construct a gateway in such a way that, from the standpoint of each framework, the *other* framework appears to be a single module providing a number of services/interfaces. In reality, these services/interfaces will be the modules provided by the other framework.

2.1 OpenORB CORBA ORB

The gateway is implemented using the OpenORB CORBA ORB (OpenORB, 2002). The version, `1.3.0`, from CVS was used rather than the latest download. In the future we hope to use a designated download. The OpenORB software is free open source software with a BSD-like license. To ensure complete bidirectional mapping of objects the following OpenORB components should be downloaded and built:

- OpenORB
- tools
- ManagementBoard
- PersistentStateService
- TransactionService
- InterfaceRespository

Follow the instructions in the `README` files in each module on how to build each module. Each module contains a PDF file describing how to configure the module. The following example script shows how initialize OpenORB:

```
#!/bin/sh
CLASSPATH="c:/OpenOrb/OpenORB/dist/openorb-1.3.0.jar"
CLASSPATH="${CLASSPATH};c:/OpenOrb/PersistantStateService/dist/openorb_pss-1.3.0.jar"
CLASSPATH="${CLASSPATH};c:/OpenORB/InterfaceRepository/dist/openorb_ir-1.3.0.jar"
CLASSPATH="${CLASSPATH};c:/OpenORB/TransactionService/dist/openorb_ots-1.3.0.jar"
CLASSPATH="${CLASSPATH};c:/OpenORB/OpenORB/dist/openorb_tools-1.3.0.jar"
CLASSPATH="${CLASSPATH};c:/OpenOrb/OpenORB/dist/avalon-framework.jar"
CLASSPATH="${CLASSPATH};c:/OpenOrb/OpenORB/dist/logkit.jar"
CLASSPATH="${CLASSPATH};c:/OpenOrb/OpenORB/dist/junit.jar"
CLASSPATH="${CLASSPATH};c:/OpenOrb/OpenORB/dist/xerces.jar"
CLASSPATH="${CLASSPATH};c:/OpenORB/InterfaceRepository/dist/openorb_ir-1.3.0.jar"
CLASSPATH="${CLASSPATH};c:/OpenORB/TransactionService/dist/openorb_ots-1.3.0.jar"
CLASSPATH="${CLASSPATH};c:/OpenORB/PersistentStateService/dist/openorb_pss-1.3.0.jar"
CLASSPATH="${CLASSPATH};c:/OpenORB/ManagementBoard/dist/openorb_board-1.3.0.jar"
CLASSPATH="${CLASSPATH};c:/OpenORB/ManagementBoard/dist/jlfgr-1.0.jar"
CLASSPATH="${CLASSPATH};c:/OpenORB/ManagementBoard/dist/jhall-1.1.1.jar"
export CLASSPATH
java org.openorb.util.MapNamingContext -ORBPort=2001 &
sleep 10s
```

```
java org.openorb.ots.Server -naming &
sleep 10s
java org.openorb.ir.Server &
```

2.2 CORBA from the Standpoint of SBW

From the standpoint of SBW, the SBW-CORBA gateway appears to be a single SBW module exporting a set of services. This SBW module is named "edu.caltech.CORBAGateway" and has an SBW management type of "unique", meaning there will be only one instance running in a given SBW session. All CORBA interfaces *usable in SBW* will show up as SBW services to the "CORBA" module. In other words, from the standpoint of other SBW modules, there will be one module named "CORBA" with a number of services on it, each service having a number of methods. Behind this façade, each service in reality will be a proxy for a CORBA object residing elsewhere.

The CORBA objects that are exposed to SBW are found by searching the CORBA Name Server. Only objects with interfaces which can be mapped to a SBW form are exposed to SBW. This process is described in more detail in section 5.

2.3 SBW from the Standpoint of CORBA

From the standpoint of CORBA, the SBW-CORBA gateway provides a number of objects, each standing in for a service defined by an SBW module. The objects are identified by a hierarchy of naming contexts underneath a particular naming context called "SBW" which is bound to the root naming context of the CORBA name server. The organization of this hierarchy is discussed below. CORBA modules at large are able to find the "SBW" naming context, then walk through the hierarchy to search for specific services or modules. We provide a tool that generates CORBA IDL from SBW module definitions, using the approach described in Section 4.1. A programmer is able to use the IDL definitions to generate client code stubs for a particular programming language using whatever IDL conversion tools are provided by the ORB being used by the programmer. This client-side interface will communicate using the normal CORBA mechanisms with an instance of the object residing in the SBW-CORBA gateway. Unbeknown to the client, the object in the gateway will actually be a proxy object communicating with the real SBW module.

3 The Organization of CORBA Naming Contexts

In order for CORBA objects to find the services made available by SBW modules through the SBW-CORBA gateway, the gateway will record the SBW modules and services in a hierarchy of naming contexts in a CORBA naming service. The interfaces of the leaf objects in this hierarchy are shown in Figure 2 on page 7. The functionality of these interfaces is described below in this section.

The gateway can be invoked, to create this hierarchy, using the following command line:

```
java -jar <SBW_HOME>/modules/CORBAGateway.jar -sbwmodule
```

where `SBW_HOME` is the directory in which SBW is installed.

As mentioned above, SBW hierarchy of naming contexts will be rooted at a naming context called "SBW". Underneath it, there will be three subcontexts: **Services**, **Modules** and **ModuleInstances**.

Services: This contains multiple levels of subcontexts corresponding to the service categories exported by SBW modules. The organization is illustrated in Figure 1 on page 6 (top). The leaves of the category tree will be references to factory objects corresponding to each service. These factory objects can be used by clients to create new instances of each service and access CORBA proxy objects for those instances. The IDL for the service factory object (`ServiceFactory`) is defined in Figure 2 on page 7.

The `getService()` method in `ServiceFactory` returns an object having an interface called `Service`; this interface is the parent interface class for all services defined by SBW modules. Each specific interface defined by a module will be a subclass of `Service`. A client that needs to instantiate a particular service can use `getService()` on the `ServiceFactory` object, then narrow the resulting proxy object to the necessary specific service interface. Section 4.1 describes how to create the IDL for a proxy object.

Modules: The entries in this naming context are a set of subcontexts, one for each module registered with SBW. (This is in contrast to the **ModuleInstances** subcontext, below, where the entries are module *instances* running in SBW.) Each module's naming context will contain (a) a reference to the factory object for each service that it implements, (b) a reference to a factory object for the standard service called "SYSTEM" implemented by every SBW module, and (c) a reference to an object of type `ModuleDescription` that returns descriptive information about the module.

ModuleInstances: The entries in this naming context are a set of subcontexts, one for each module instance running in SBW. Under the entry for each module will be an entry for each running instance of the module; each instance will be numbered "1", "2", etc. Within the context of each instance, there will be sets of objects corresponding to the interfaces of the services defined by this module. The organization of the **ModuleInstances** hierarchy is illustrated in Figure 1 (bottom).

3.1 `ServiceFactory`

This interface allows the caller to create new CORBA proxy objects that provide access to a given service on a module instance as well as methods to access information about the service.

Service getService() raises (SBWException)

Returns a proxy object for the given service. The returned Service can be narrowed to the interface for the service that is generated according to the mechanism described in section 4.

string getHelp()

Returns the help string for this service

string getDisplayName()

Returns the display name for this service

CosNaming::NamingContext getCategory()

Returns the naming context for the category in which this service occurs.

CosNaming::NamingContext getModule()

Returns the naming context for the module in which this service occurs.

string getName()

Returns the unique service identification string for this service.

- Services
  - *categoryA*
    - *subcategoryB*
      - *subcategoryC*
        - ServiceFactory*S1_Module1*
        - ServiceFactory*S2_Module1*
  - *categoryD*
    - *subcategoryE*
      - ServiceFactory*S3_Module2*
  - ServiceFactory*S4_Module3*
  - ServiceFactory*S5_Module4*
- Modules
  - *Module1*
    - ModuleDescription
    - ServiceFactory*S1*
    - ServiceFactory*S2*
    - SystemServiceFactory
  - *Module2*
    - ModuleDescription
    - ServiceFactory*S3*
    - SystemServiceFactory
  - *Module3*
    - ModuleDescription
    - ServiceFactory*S4*
    - SystemServiceFactory
- ModuleInstances
  - *Module1*
    - *1*
      - *Service1*
      - *Service2*
      - SYSTEM
    - *2*
      - *Service1*
      - *Service2*
      - SYSTEM
  - *Module2*
    - *1*
      - *Service3*
      - SYSTEM

Figure 1: *(Top) The organization of the* `Services` *naming context. Each category establishes a subcontext, where the leaves are actual services. The objects actually recorded for the services (i.e., the leaves of the category tree) will be references to factory objects that can be used to create new instances of the given service. Uncategorized services will appear at the top-level under* **`Services`**. *(Middle) The organization of the* `Modules` *naming context. Each entry is a subcontext for a particular module defined in SBW. The entries under each module are references to the factory service objects that can be used to generate instances of the service. (Bottom) The organization of the* `ModuleInstances` *naming context. Each entry is a subcontext for a particular module defined in SBW. Underneath each module, each instance currently running in SBW is listed; the instances are numbered "1", "2", etc. Then, the entries under each module instance are references to the service objects.*

```
#include <CosNaming.idl>

module edu {
  module caltech {
    module sbw {
      module CORBA {

        enum ManagementType
        {
          Unique,
          SelfManaged
        };

        interface Service;

        interface ServiceFactory
        {
          Service getService() raises (SBWException);
          string getHelp();
          string getDisplayName();
          CosNaming::NamingContext getCategory();
          CosNaming::NamingContext getModule();
          string getName();
          string getCategoryString();
          string getModuleName();
        };

        interface ModuleDescription
        {
          string getName();
          string getHelp();
          string getDisplayName();
          string getCommandLine();
          ManagementType getManagementType();
          CosNaming::NamingContext getInstances() raises (SBWException);
        };

        interface ServiceDescription
        {
          string getMethodHelp(in string sbwMethodSignatureOrName)
                raises (SBWException);

          CosNaming::NamingContext getModuleInstance();
          ServiceFactory getServiceFactory() raises (SBWException);
        };

        interface Service
        {
          ServiceDescription getDescription();
        };

        interface SystemService : Service
        {
          void shutdown() raises (SBWException);
        };

      };
    };
  };
};
```

Figure 2: *The IDL definitions of various interfaces used in the naming contexts of Figure 1 on the preceding page and Section 3. The definition of `SBWException` is shown in Figure 3.*

string getCategoryString()

Returns the category string for this service.

string getModuleName()

Returns the unique module identification string for the module containing this service.

3.2 `ModuleDescription`

This interface gives access to information about a given module.

string getName()

Returns the unique module identifier of this module

string getHelp()

Returns the help string for this module

string getDisplayName()

Returns the display name for this module

string getCommandLine()

Returns the command line that is used to create an instance of this module

ManagementType getManagementType()

Returns the management type of this module. This can be either:

- `Unique` one module instance services all requests: all calls requesting the same service on such a module, using the method `ServiceFactory::getService`, all return the same object.
- `SelfManaged` multiple calls to `ServiceFactory::getService` for such a module returns different objects each implemented by a different module instance.

CosNaming::NamingContext getInstances() raises (SBWException)

returns the naming context containing all the instances of this module.

3.3 `Service`

This interface acts as a base interface for the CORBA Service proxy objects created by the gateway. This interface consists of one method:

ServiceDescription getDescription()

Returns an object which describes the service.

3.4 `ServiceDescription`

This interface gives access to information about a given service.

string getMethodHelp(in string sbwMethodSignatureOrName) raises (SBWException)

Returns the help for a given method on the service. The method is specified by giving either the method name, the method name and arguments or the method's complete signature.

CosNaming::NamingContext getModuleInstance()

Returns the naming context for the module instance containing this service.

ServiceFactory getServiceFactory() raises (SBWException)

Returns the service factory which constructed this service.

3.5 `SystemService`

All the objects in bound to the module instance naming contexts as "SYSTEM" have this interface. This interface consists of one method:

void shutdown() raises (SBWException)

This method disconnects the module instance from the SBW broker. Typically this results in the module instance terminating.

4 Mapping SBW method signatures to CORBA IDL

In order for CORBA and SBW to communicate, there must be a mapping between the CORBA interface definitions and SBW module/service/method descriptions. For bi-directional communications, two directions must be established: one from SBW module/service/method descriptions to CORBA, and the reverse. In this section, we describe how SBW definitions are turned into CORBA IDL definitions. This is the direction needed to make SBW services available to CORBA-based clients. The mapping we describe here attempts to balance simplicity and ease of use against strict type checking. In Section 5, we consider the reverse mapping, from CORBA to SBW.

4.1 Creating IDL files from SBW

You can use the SBW `Browser` module to create CORBA IDL files corresponding to the interfaces exposed by a CORBA service proxy object for a SBW service. Invoke the following command in a Unix shell or Windows command prompt:

```
java -jar <SBW_HOME>/modules/browser.jar
      -g IDL -m <module> -s <service> -o <idl file>
```

where `SBW_HOME` is the directory in which SBW is installed, `module` is the unique identification string of the SBW module, `service` is the unique identification string of the service and `idl file` is the name of the IDL file to be created.

4.2 Introduction to SBW to CORBA Mapping

Table 1 on the next page summarizes the equivalences between constructs in SBW and CORBA. Establishing a mapping between *most* constructs in SBW and their equivalents in CORBA IDL

SBW	IDL	Comments	Section
module	`module`	CORBA `module` definitions can be nested to imitate the namespace conventions used in SBW	4.3
service	`interface`	Only a subset of IDL interface definition elements are permitted: specifically, attribute definitions are not allowed	4.4
exception	`exception`	Strictly speaking, SBW does not have an exception data type, but where possible, the SBW language bindings provide an exception mechanism	4.5
`int`	`long`	These are direct mappings	4.6
`double`	`double`		
`byte`	`octet`		
`boolean`	`boolean`		
`string`	`string`		
defined list	`struct`	The structures must also be named	4.7
undefined list	`SBWAnyList`	A special type must be used to limit the datatypes permitted in the list	4.8
array	`sequence`	The sequences must be also `typedef`'ed	4.9

Table 1: *Summary of mapping between constructs in SBW and CORBA IDL.*

is straightforward for many elements of SBW, but arrays and lists cause some difficulties. We therefore begin with the easy cases and then treat lists and arrays last.

4.3 Modules

Modules in CORBA IDL introduce a namespace for definitions of interfaces and other items. SBW modules map directly to IDL modules. In SBW, we have taken to naming modules using Java-style dotted notation, as in, for example, "edu.caltech.trigj". The equivalent can be achieved in CORBA IDL by simply nesting module definitions. The following is an example:

```
module edu {
  module caltech {
    module trigj {
      // ... interface and type definitions here ...
    };
  };
};
```

Most IDL-to-Java generation tools will translate the module name in this IDL definition into a package named `edu.caltech.trigj`, exactly as desired.

4.4 Services

An SBW service maps naturally to a CORBA IDL `interface`. Suppose that our module "edu.caltech.trigj" defines a service named "Trig" having two methods with the following SBW method signatures: `double sin(double)` and `double cos(double)`. The following is a translation of this into CORBA IDL:

```
module edu {
  module caltech {
    module trigj {

      interface Trig {
        double sin(in double x) raises(edu::caltech::sbw::SBWException);
```

```
        double cos(in double x) raises(edu::caltech::sbw::SBWException);
      };
    };
  };
};
```

Each method (known as an *operation* in CORBA IDL parlance) needs to be defined as raising an `SBWException` because this is how SBW informs a client of problems during service method invocations. (We leave the definition of `SBWException` to the next section.) Note also the presence of the `in` attributes on the parameter declarations. IDL requires that parameters be declared as being `in`, `out`, or `inout`. SBW only has the concept of `in` parameters, therefore SBW method definitions always translate into IDL operations with `in` parameters.

An additional issue is that SBW allows method parameters to be specified in terms of types only, without parameter names, but IDL requires parameters to be named. This implies that the SBW-to-IDL mapping may sometimes have to generate names. Since humans will want to read the IDL produced, we are faced with the question of how to generate automatically some readable and pleasant-looking names for the parameters. We propose the following convention: for every unnamed parameter in a method argument list, use a name of the form `arg`*`X`*, where *`X`* stands for the ordinal number of the parameter in the list of parameters. For example, if the SBW signature is "`double foo(boolean, int, string)`", the IDL becomes

```
interface servicename {
  double foo(in boolean arg1, in int arg2, in string arg3);
};
```

IDL allows data types to be defined within an `interface` definition. SBW does not have an equivalent capability; however, because sequences and structures in IDL have to be named (using `typedef`'s), the IDL mapping for an SBW service definition may contain type definitions. This is discussed further in Sections 4.7 and 4.9.

4.5 Exceptions

As mentioned above, SBW does not actually define an exception data type, but it does communicate exceptional conditions to clients. In object-oriented languages such as Java, SBW uses the built-in exception definition mechanisms; in languages such as C, callers must check the return value from method calls and explicitly call specific methods (e.g., `SBWExceptionGetCode()`, `SBWExceptionGetMessage()`, etc.) to get information about exceptions that occur.

Since IDL provides exceptions, we make use of them. Figure 3 on the following page gives the definition of an IDL that defines `SBWException`. The definition is placed inside the namespace `edu::caltech::sbw`, matching what is used in the SBW language bindings. IDL files can include the file that defines this exception using the IDL directive `#include <SBW.idl>`.

The definition of `SBWException` uses an enumeration to define the possible types of SBW exceptions that can be generated. In SBW, the various exceptions are hierarchically organized, with `SBWException` being the parent class of subclasses such as `SBWCommunicationException`. However, CORBA IDL does not provide for inheritance of exception types. The simple approach taken here to communicating the specific type of an exception is to use a variable (`code`) that expresses which specific type of exception has occurred. The error message and detailed error message associated with the exception are recorded in additional variables `message` and `detailedMessage`, respectively.

4.6 Basic Types: `int`, `double`, `byte`, `boolean`, and `string`

The five most basic data types in SBW, `int`, `double`, `byte`, `boolean`, and `string`, map directly to specific primitive types in CORBA IDL. The mapping is shown in Table 1 on the page before.

```
module edu {
  module caltech {
    module sbw {
      module CORBA {

        enum SBWExceptionCode {
          SBWApplicationException, SBWRawException, SBWCommunicationException,
          SBWModuleStartException, SBWTypeMismatchException,
          SBWIncompatibleMethodSignatureException, SBWModuleIdSyntaxException,
          SBWIncorrectCategorySyntaxException, SBWServiceNotFoundException,
          SBWMethodTypeNotBlockTypeException, SBWMethodAmbiguousException,
          SBWUnsupportedObjectTypeException, SBWMethodNotFoundException,
          SBWSignatureSyntaxError, SBWModuleDefinitionException,
          SBWModuleNotFoundException
        };

        exception SBWException {
          SBWExceptionCode code;
          string message;
          string detailedMessage;
        };

      };
    };
  };
};
```

Figure 3: *The portion of the file `SBW.idl` defining `SBWException`.*

4.7 Defined Lists

The SBW signature type `list` maps most naturally to an IDL structure, `struct`. This type allows the definition of an ordered, heterogeneous collection of elements.

Unfortunately, the IDL `struct` type cannot be used directly in a method's parameter declaration or a return type declaration—a `struct` cannot be used anonymously. Each structure definition instead introduces a type name which can then be used in parameter or return type declarations. This is unlike method signature strings in SBW, where the types do not need to be named. The implication is that when mapping SBW signatures to IDL, some situations require automatically generating new identifiers. For example, the SBW method signature "`{string s, boolean b} foo()`" cannot be defined in IDL simply and directly as "`struct {string s; boolean b;} foo()`". Instead, the `struct` must be given a name, for example as follows:

```
struct fooReturnType {
  string s;
  boolean b;
};
fooReturnType foo();
```

A further issue is that SBW allows a list to have unnamed members, which is not allowed in IDL. This introduces another situation in which identifiers need to be generated automatically. To take a specific example, representing an SBW signature string such as "`void foo({int, int} x)`" requires that the type of `x` must be defined as a named `struct` type and the two integer elements of the structure must be given names.

Since structure and structure member names are optional in SBW method signatures, it means that the identifiers for `struct` definitions sometimes need to be machine-generated when mapping SBW to IDL. The problem is made somewhat easier by the fact that the types only need to be defined in the context of method signatures. We propose the following conventions. First, the SBW-to-IDL mapping will place `struct` definitions within the `interface` block containing the methods for which the argument types are being defined. Second, the SBW-to-IDL mapping will follow these rules:

- *If the list is the return type of a method*: give the equivalent IDL `struct` a name of the form ____`ReturnType`, where the blank stands for the name of the method. For example, if some service named `servicename` has a method with an SBW signature of "`{double x, int y} foo()`", the IDL becomes

  ```
  interface servicename {
    struct fooReturnType {
      double x;
      long y;
    };
    fooReturnType foo();
  };
  ```

- *For each unnamed element of a list*: generate a name of the form `memberX`, where `X` stands for the ordinal position of the element in the list. For example, the SBW method signature "`{double, int} foo()`" turns into the following IDL:

  ```
  interface servicename {
    struct fooReturnType {
      double member1;
      long member2;
    };
    fooReturnType foo();
  };
  ```

  Where the types of the objects inside the list are complex and need to defined separately the member type name has the form *`structure name_member name`*`Type`. *structure name* is the name of the structure type being defined and *member name* is the name of the list member.

4.8 Undefined Lists

The list expression "{}" in an SBW method signature signifies a list whose contents and length are unspecified. CORBA IDL *does* provide a way to express this concept, using a `sequence` of `any`, and this is a viable approach to representing the SBW "{}" construct. However, the problem with this approach is that it does not allow for compile-time checking of the types of items inserted into the list. It would be too easy for CORBA-based modules to insert data types that are not supported by SBW-based modules. The resulting error would not be detected until run-time, potentially leading to difficult-to-debug situations.

To work around this issue, we define a particular IDL type, `SBWAnyList`, to stand for SBW's "{}" expression. The definition is shown in Figures 4 on the following page and 5 on page 15. It is an unbounded sequence of elements of type `SBWType`, each of which is a union of the possible SBW types that can appear in a list. This allows an SBW method signature string such as "`{} foo(double x)`" to be translated into IDL as "`SBWAnyList foo(double x)`".

4.9 Arrays of Non-List Types

The array type in CORBA IDL requires that the length of an array be prespecified. This conflicts with the array type in SBW, whose length is not specified. Thus, surprisingly, SBW arrays cannot be mapped to the natural array type in IDL. Instead, is it necessary to use the `sequence`, an unbounded array of elements all of the same type. Table 2 on the following page gives mappings between various examples of arrays of data types in SBW and their equivalents in terms of IDL's `sequence` type.

Unfortunately, the sequences shown in Table 2 on the next page cannot be used anonymously for

SBW Signature	Equivalent CORBA IDL
`int[]`	`sequence<long>`
`double[]`	`sequence<double>`
`byte[]`	`sequence<octet>`
`boolean[]`	`sequence<boolean>`
`string[]`	`sequence<string>`
`{}[]`	`sequence<SBWAnyList>`
`int[][]`	`sequence<sequence<long> >`
`double[][]`	`sequence<sequence<double> >`
`byte[][]`	`sequence<sequence<octet> >`
`boolean[][]`	`sequence<sequence<boolean> >`
`string[][]`	`sequence<sequence<string> >`
`{}[][]`	`sequence<sequence<SBWAnyList> >`

Table 2: *SBW arrays expressed as arrays in IDL.*

the return type of a method or the type of a parameter. CORBA IDL requires that sequences be given identifying names using `typedef`. This is unlike method signature strings in SBW, where the types do not need to be defined first.

The need to define the types means that instead of producing IDL such as "`void foo(in`

```
module edu {
  module caltech {
    module sbw {
      module CORBA {

        enum TypeEnum {
            StringType, IntType, DoubleType, BooleanType, ByteType, ListType,
            String1DArrayType, Int1DArrayType, Double1DArrayType,
            Boolean1DArrayType, Byte1DArrayType, List1DArrayType,
            String2DArrayType, Int2DArrayType, Double2DArrayType,
            Boolean2DArrayType, Byte2DArrayType, List2DArrayType
        };

        typedef sequence<string> StringSeq;
        typedef sequence<StringSeq> StringSeqSeq;
        typedef sequence<double> DoubleSeq;
        typedef sequence<DoubleSeq> DoubleSeqSeq;
        typedef sequence<long> LongSeq;
        typedef sequence<LongSeq> LongSeqSeq;
        typedef sequence<boolean> BooleanSeq;
        typedef sequence<BooleanSeq> BooleanSeqSeq;
        typedef sequence<octet> OctetSeq;
        typedef sequence<OctetSeq> OctetSeqSeq;
      };
    };
  };
};
```

Figure 4: *The definition of types for the type `SBWAnyList` in file `SBW.idl`. See Figure 5 for the definition of `SBWAnyList`*

.

sequence<long> arg)", it is necessary to define the type separately, for example as follows:

```
typedef sequence<long> argType;
void foo(in argType arg);
```

Since these types are not given names in SBW signatures, the identifiers for the typedef's must

```
module edu {
  module caltech {
    module sbw {
      module CORBA {

        union SBWType switch( TypeEnum ) {
         case StringType:
             string stringValue;
         case IntType:
             long intValue;
         case DoubleType:
             double doubleValue;
         case BooleanType:
             boolean booleanValue;
         case ByteType:
             octet byteValue;
         case ListType:
             sequence<SBWType> listValue;

         case String1DArrayType:
             StringSeq string1DArrayValue;
         case Int1DArrayType:
             LongSeq int1DArrayValue;
         case Double1DArrayType:
             DoubleSeq double1DArrayValue;
         case Boolean1DArrayType:
             BooleanSeq boolean1DArrayValue;
         case Byte1DArrayType:
             OctetSeq byte1DArrayValue;
         case List1DArrayType:
             sequence<sequence<SBWType> > list1DArrayValue;

         case String2DArrayType:
             StringSeqSeq string2DArrayValue;
         case Int2DArrayType:
             LongSeqSeq int2DArrayValue;
         case Double2DArrayType:
             DoubleSeqSeq double2DArrayValue;
         case Boolean2DArrayType:
             BooleanSeqSeq boolean2DArrayValue;
         case Byte2DArrayType:
             OctetSeqSeq byte2DArrayValue;
         case List2DArrayType:
             sequence<sequence<sequence<SBWType> > > list2DArrayValue;
        };

        typedef sequence<SBWType> SBWAnyList;

      };
    };
  };
};
```

Figure 5: *The definition of* `SBWAnyList` *in file* `SBW.idl`*. See Figure 4 for the definition of types not defined here.*

.

be machine-generated when translating SBW definitions to IDL. We propose the following two rules for generating the names:

- *If the type is the return type of a method*: use a type name of the form ____`ReturnType`, where the blank stands for the name of the method. For example, if the SBW signature is "`int[] foo()`", the IDL becomes

```
interface servicename {
  typedef sequence<long> fooReturnType;
  fooReturnType foo();
};
```

- *If the type is the type of a parameter*: use a type name of the form ____`Arg`*X*`Type`, where the long blank stands for the name of the method and *X* stands for the ordinal number of the parameter in the list of parameters. For the name of the parameter in the IDL definition, use `arg`*X*. For example, if the SBW signature is "`double foo(boolean, int[], string)`", the IDL becomes

```
interface servicename {
  typedef sequence<long> fooArg2Type;
  double foo(in boolean arg1, in fooArg2Type arg2, in string arg3);
};
```

  The need to include the name of the method in the type definition stems from the fact that there may be multiple methods defined in an interface, each with their own `typedef`'ed argument types. If the type names were only distinguished, say, on the basis of the name of the argument, name collisions could result.

There is one last case to consider. Table 2 on page 14 does not show the case of arrays of *defined lists*. This case is a simple combination of defined lists (Section 4.7) and the IDL `sequence` type, with the addition that the sequences must be first named using a `typedef`. To handle this mapping, we propose the following name-generation rules:

- *If the array type is the return type of a method*: (a) for the array elements, use a name of the form ____`ReturnArrayElement`, where the blank stands for the name of the method; and then (b) for the sequence of elements, use a type name of the form ____`ReturnType`, where the blank stands for the name of the method. For example, if some service named `servicename` has a method with an SBW signature of "`{double x, int y}[] foo()`", the IDL becomes

```
interface servicename {
  struct fooReturnArrayElement {
    double x;
    long y;
  };
  typedef sequence<fooReturnArrayElement> fooReturnType;
  fooReturnType foo();
};
```

- *If the array type is the type of a parameter*: (a) for the array element type, use a name of the form ____`Arg`*X*`ArrayElement`, where the long blank stands for the name of the method and *X* stands for the ordinal number of the parameter in the list of parameters; and (b) for the name of the whole type, use a type name of the form ____`Arg`*X*`Type`. For example, if the SBW signature is "`double foo(boolean, {int, boolean}[], string)`", the IDL becomes

```
interface servicename {
  struct fooArg2ArrayElement {
    long element1Type;
```

```
    boolean element2Type;
  };
  typedef sequence<fooArg2ArrayElement> fooArg2Type;
  double foo(in boolean arg1, in fooArg2Type arg2, in string arg3);
};
```

5 Exposing CORBA to SBW

The gateway exposes CORBA objects to as SBW services. These services appear as part of the `edu.caltech.CORBAGateway` module. The exposed CORBA objects are found during a search of the CORBA name server. A CORBA object must comply with the following conditions to be exposed as SBW service:

- the object must be bound to a naming context which itself is on a path of bindings from the root node of the CORBA name server
- the object's interface definition must be registered with the CORBA interface repository
- the object's interface must comply with the IDL restrictions described in section 5.1.

The gateway uses the path of bindings from the root of the name server to exposed objects to derive a unique name for the exposed SBW services. These paths are derived as part of the depth first search of the name server used to locate CORBA objects for exposure. This search ensures that the path to each object found is unique. Section 5.2 describes how these paths are translated in to service identification strings.

In addition to services which correspond to CORBA objects the gateway also has a service called `CORBAGateway` which has a single method `void refresh()` which adds new objects that appear in the CORBA name server to the set of services exposed by the gateway.

5.1 Limiting IDL to SBW-Compliant Constructs

CORBA IDL provides more data types and constructs than SBW's module/service/method definition formalism. Therefore, arbitrary CORBA IDL definitions cannot in general be translated into SBW definitions. The IDL definitions of objects intended for exposure to SBW must only use the constructs listed in Table 1 on page 10. This is because SBW has fewer constructs than CORBA IDL, therefore CORBA clients cannot use more than this subset in their interactions with SBW-based applications. We propose the following scheme to map CORBA types to SBW method signatures:

- CORBA services exposed through the SBW-CORBA gateway must use the same subset of IDL used in Section 4 for the SBW-to-IDL mapping. The specific naming schemes for `typedef`'ed names, etc., used in Section 4 do *not* have to be used; it is only the following subset of IDL constructs:

`boolean`	`octet`	`string`
`double`	`raises`	`struct`
`interface`	`SBWAnyList`	`typedef`
`long`	`SBWexception`	
`module`	`sequence`	

- Other CORBA constructs, such as other data types (e.g., `enum`, `fixed`, `short`, `unsigned`, `float`, `char`, `any`, etc.), qualifiers (e.g., `const`), parameter types besides `in` (i.e., `out` and `inout`), attributes within interfaces, and `pragma`'s, cannot be used.

5.2 Rewriting CORBA Names

The path of bindings that uniquely identify an object relative to the root naming context of the CORBA context can potentially consist of several name context bindings. In addition the names in CORBA name services may include characters that are not permitted in SBW service names. Thus, a path of bindings needs to be translated when a CORBA definition is exposed through SBW. We propose the following set of rewriting rules for translating a path of bindings to SBW service names:

- Any character not allowed in an SBW service name (i.e., any character other than alphanumerics and the underscore, '_') is translated to a series of characters of the form `_n`*`XXX`*`_` where *`XXX`* is the byte code of the character.
- The start of a CORBA *component identifier* is prefixed with `_i`.
- The start of a CORBA *component kind* is prefixed with `_k`.
- The start of a subsequent CORBA *naming context* is prefixed with `_l` ("l" for "level").
- Any underscore characters are translated into two underscores ("__") in a row.

References

OpenORB (2002). The Community OpenORB version 1.3.0. Available via the World Wide Web at `http://openorb.sourceforge.net/`.