12

Using SDL to develop CORBA object implementations

Björkander, M.
Telelogic AB
Box 4128, S-203 12 Malmö, Sweden
Phone: + 46 40 17 47 31
E-mail: morgan.bjorkander@telelogic.se

Abstract

This paper describes a methodology for using SDL in conjunction with CORBA; more specifically the CORBA-oriented approach is explored, where SDL is used to implement a predefined IDL description. This approach also includes the definition of mapping rules between IDL and SDL.

An in-depth description of how the implementation of the CORBA-oriented approach is carried out is also provided, focusing particularly on how an SDL design tool is made to interwork with an existing ORB implementation.

Keywords

CORBA, IDL, SDL, OOA, OOD

© IFIP 1997. Published by Chapman & Hall

1 INTRODUCTION

With the advent of the Common Object Request Broker Architecture (CORBA) to manage large, heterogeneous object systems, the need to be able to specify, verify, and test such systems is becoming increasingly important. It is not sufficient to be able to specify object behaviour using for example C++, because the behaviour of such systems are not easily verifiable. Instead, by using a formal description language such as the Specification and Description Language, SDL (ITU-T Recommendation Z.100, 1995), to define the object behaviour all the commonly used techniques for verification and validation become available.

The work presented in this paper is based on results obtained when implementing CORBA-support in the SDL Design Tool (SDT) developed by Telelogic, and is considered both from a methodology and an implementation point of view.

In Section 2, a CORBA-oriented approach of using SDL together with CORBA is presented. This approach is also incorporated in the SDL-based Object Modelling Technique (SOMT), which provides a framework for how object-oriented analysis and SDL-based design might be used in a coherent way. The necessary mapping rules between the Interface Definition Language (IDL) and SDL are also summarised. This section is concluded with some examples of how object models might be mapped to IDL.

Section 3 goes more into detail regarding how an SDL application designed according to the principles outlined in Section 2 can be implemented together with an existing ORB implementation.

The last section describes the conclusions obtained from this work, and also some directions for future work.

2 DEFINING OBJECT BEHAVIOUR USING SDL

2.1 Using CORBA and SDL

As is described in (Olsen, 1995), there are basically two ways to use SDL with CORBA, depending on whether SDL is viewed as a specification or design language:

- The CORBA-oriented approach, where SDL is supported as implementation language for the definition of behaviour and treated in the same way as the already supported implementation language C++.
- The SDL-oriented approach, where a CORBA platform is used as execution system for SDL processes.

When to use the different approaches

The above approaches may seem very similar at first, but the methodologies used are quite different.

The SDL-oriented approach is primarily used when SDL is viewed as a specification language. Once the specification (of the entire distributed system) has been performed in SDL, it should be partitioned into arbitrarily small subsystems that are then (optionally) implemented on different machines and communicate with each other using CORBA. The smallest unit of partitioning should be at the process level. In this case, IDL descriptions are automatically generated from the SDL system; it might, however, be necessary to limit the available SDL concepts when using this approach.

The CORBA-oriented approach, on the other hand, is primarily used when SDL is viewed as an implementation language, and a given IDL description should be designed using SDL. That IDL description is used to generate a stub (skeleton) in SDL, to which behaviour is then added.

This paper focuses on the CORBA-oriented approach, whereas a more thorough description of the SDL-oriented approach can be found in (Olsen, 1995).

The CORBA-oriented approach

The most important parts of the development process when using the CORBA-oriented approach are shown in Figure 1.

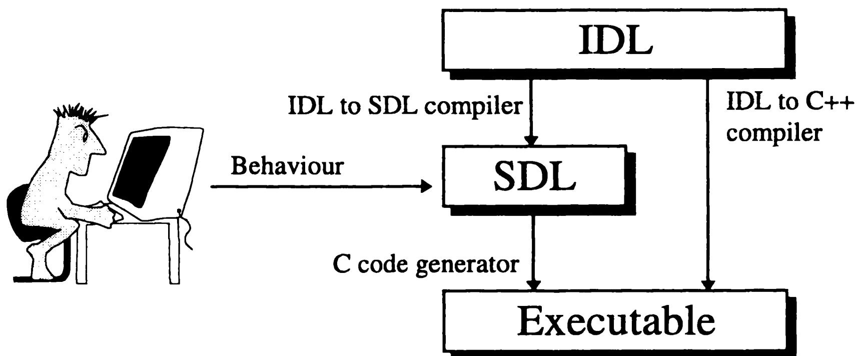

Figure 1 The activities of the CORBA-oriented approach.

The activities that can be clearly distinguished when using this approach are:

1. Convert an IDL description to an SDL stub.
2. Define the behaviour of the SDL system using the generated SDL stub as the starting point.
3. Generate the C/C++ code that is used to implement the SDL application.
4. Generate the C++ code that is needed by the ORB.

In reality, several additional activities occur, but these will be further outlined in Section 3.1. The only activities that should concern the developer are the first two; the user must be aware that the SDL system is a CORBA server (i.e., an object implementation that can also act as a client by requesting services from other object implementations).

By using the mapping rules described in Section 2.3 it is possible create a tool that automatically converts an IDL description to an SDL stub. The latter two activities described above are also performed automatically, and it is necessary to adhere to the IDL to C++ mapping rules that are defined in (OMG, 1996).

2.2 The SOMT method

The SDL-based Object Modelling Technique (SOMT) has been developed to manage the entire development process with the key focus of integrating object-oriented analysis with SDL design (Telelogic, 1996). The main activities of the SOMT method are summarised below.

Requirements analysis

In the requirements analysis activity, the requirements of the system that is going to be built are captured. The system itself is viewed as a black box, where only the interaction with objects and concepts defined outside its boundaries are of interest. The foremost modelling concepts of the requirements analysis are use cases (using MSCs) and an object model, but additional concepts also exist.

System analysis

The system analysis describes the architecture of the system as well as the objects that are needed to provide its functionality. It is only the objects that must be present in the system that should be modelled at this stage; objects outside the system were modelled in the requirements analysis. Use cases and an object model are the primary models used within this activity.

System design

The purpose of the system design activity is to define how the system architecture is going to be implemented. It is necessary to define both the architecture of the resulting application and the interfaces between the different components. The main modelling notation for system design is SDL, where the architecture is defined using system and block diagrams, and the interfaces are reflected using signals and remote procedures.

Object design

During the object design the functionality of the system is defined in detail, and the behaviour of the active objects in the system is described using SDL process graphs.

Implementation

The goal of the implementation is to produce the final application. This is done by automatically producing code directly from the SDL design.

Implementation links and Paste As

One of the features of the SOMT method is implementation links (*implinks*); these are used to maintain a relationship between objects in different models to indicate that one object can be seen as the implementation of another.

The main purpose of implinks is to provide traceability of the development through the different activities mentioned above, thereby facilitating for example consistency checks etc.

One way to create implinks is to make use of the *Paste As* functionality that is defined by SOMT. This is a tool supported concept that allows a developer to copy an object in one model, and then paste it into another model as something else, while at the same time creating an implink between the two objects. By selecting one of these objects it is then possible to follow the link to the other object.

As an example, an object model class can be copied from the analysis model and then pasted as an SDL process type in the system design model. This means that the paste as functionality can be seen as the implementation of a set of transformation rules.

According to the SOMT method, it is possible to copy/paste as arbitrarily large collections of objects and relationships, but current tool support restricts this by only allowing single object entities to be copied/pasted as.

Introducing CORBA in SOMT

When the CORBA-oriented approach is introduced in SOMT, IDL is mainly used to describe the interfaces between the different parts in the system design. In fact, the definition of IDL interfaces replaces the ordinary system design activity that was mentioned above. Each of the components that are arrived at can then be implemented using the preferred language, and the subsystems that should be implemented using SDL are part of a larger system that is connected through CORBA.

Ordinarily, when going from analysis to design, the SOMT method advocates that objects model classes are pasted from the system analysis, and then pasted as SDL entities in the system design. In the CORBA-oriented approach, however, the object model concepts that are part of the component interfaces should first be pasted as IDL. Some specific examples of mapping object model classes to IDL are provided in Section 2.5.

Then, when the IDL description is completed and it should be implemented using SDL, an SDL stub is generated from it. Internal information that is not part of the interface definition can then be pasted as the SOMT method ordinarily describes.

Inherent in the IDL description (through modules and interfaces) is information about the system architecture, and this structure information is kept in the SDL system stub through the mapping rules that are described in Section 2.3. Once the system design has been performed, remaining activities such as providing the behaviour of the SDL system are performed as usual.

2.3 Mapping IDL to SDL

In order to create the implementation language stub it is necessary to provide mapping rules from IDL. In (OMG, 1996), several chapters are concerned with how different languages are mapped. For SDL, mapping rules have been proposed in (Born, 1996) and (Björkander, 1996).

Mapping Rules

The mapping rules from IDL to SDL described in (Björkander, 1996) can be summarised as follows:

- A module provides a namescope and a mechanism to group interfaces. As such it is mapped to a block type, where nested modules become nested block types.
- An interface is mapped to a process type. As the interface contains the attributes and operations that are available on an object, the corresponding process type contains the appropriate SDL definitions of these. Object references are mapped to PId values.
- An operation describes a service that is offered by an object, and it can be defined either as synchronous, where the client is blocked while the request is being handled by a server, or as asynchronous (using the keyword oneway), where the client making the request simply continues executing. In the former case, the operation is mapped to a remote procedure, while in the latter it is mapped to a signal. Raises and context expressions are still subject to be mapped, as are exceptions.
- An attribute is mapped to a declaration of a variable together with two remote procedures that are used to get and set the value of the variable. If the attribute is defined as readonly, the set operation is omitted.
- Interface inheritance must be flattened in SDL, where both operations and attributes defined in a base type have to be duplicated in a derived type, i.e., SDL inheritance cannot be used.
- Constants are mapped to synonyms.
- How basic types are mapped is shown in Table 1. Note that predefined types have the prefix 'CORBA_' in SDL. The type any currently have no suitable mapping in SDL.

Table 1 Mapping basic types

IDL Type	*SDL Type*	*syntype of*

long	CORBA_long	Integer
short	CORBA_short	Integer
unsigned long	CORBA_unsigned_long	Integer
unsigned short	CORBA_unsigned_short	Integer
double	CORBA_double	Real
float	CORBA_float	Real
char	CORBA_char	Character
boolean	CORBA_boolean	Boolean
octet	CORBA_octet	Octet

- Constructed types are mapped as follows:
    - An enum is mapped to a newtype with the appropriate literals.
    - A struct is mapped to a newtype with the corresponding struct.
    - A union is mapped to a newtype with a corresponding struct, where the first member of the struct is a tag matching the union's switch type.
- The template types are mapped as follows:
    - A sequence is mapped to the generator string.
    - A string is mapped to to the type CORBA_string, which is a syntype of Charstring.
- The complex declarators are handled as follows:
    - An array is either mapped to the generator array, together with an additional type to define its index range, or as the generator CORBA_array, which is defined to have a limited index range corresponding to that of the original IDL array.

Differences in the mapping proposals

The mapping proposals referred to above are very similar, but there are some minor differences. In (Born, 1996):

- All operations are mapped to remote procedures to avoid using channels and signal routes. This motive is questionable, and signals better convey the asynchronous nature of oneway operations.
- A context expression is mapped to an additional PId parameter of the operation for which it is defined.

- An exception is mapped to a struct value, which is stored in a new process instance in the server whenever an exception occurs. The client can then access the server to find out which exception has occurred. While the basic mapping is natural, it is not a good idea to pass a PId value (object reference) back to the client for it to examine. For example, it is not possible for the client to catch a system exception. A better mapping would probably be to pass the raised exception back in the form of a struct.
- All types are mapped as ASN.1 types. This is a good idea, but it should probably be possible to use either ASN.1 or SDL types in the mapping.
- Interfaces are mapped as process types on both the client and the server side. To include the process types on the client side seems strange and superfluous.

SDL System Structure
An SDL system that is based on the above mapping rules has a particular architecture; one package is used to define interface concepts, such as types, signals and remote procedures, whereas another is used to contain the structural information that can be derived from the IDL description. The interface package can be resused by other SDL clients that want to access services from this object implementation. Consider for example the SDL system that is shown in Figure 2.

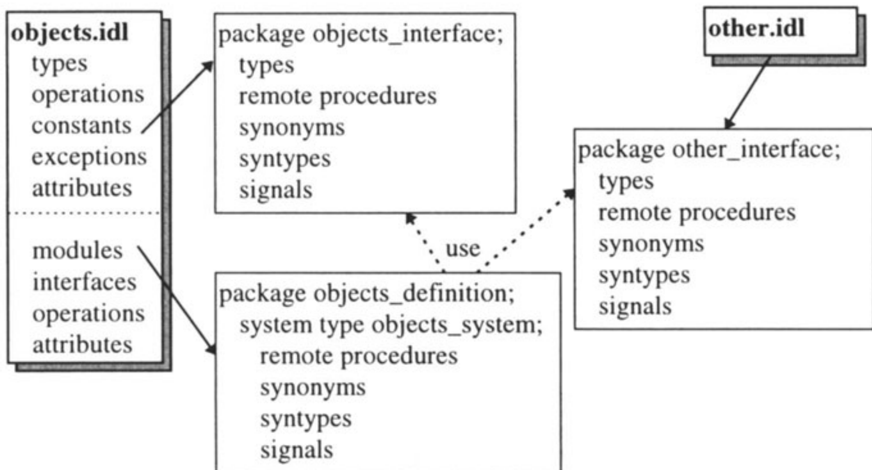

Figure 2 A schematic view of the SDL system structure.

In this example, an object implementation in SDL is created from the IDL file named *objects.idl.* The stub that is created will contain one package named *objects_interface*, and one package named *objects_definition*; this latter package also contains a system type named *objects_system.* The object implementation should make use of services that are defined in the IDL file *other.idl.* In order for the system implementing *objects* to access these services, the IDL description of

other is also converted to SDL, but only the interface package (*other_interface*) needs to be used.

2.5 Extensions to SDL-92

The fact that it is allowed to call remote procedures over system boundaries is part of a larger standardisation effort to harmonise signals and remote procedures (and is included in an addendum to (ITU-T Recommendation Z.100, 1995) that is called SDL-96). This harmonisation also makes it allowed to include remote procedures in channels, signal routes, and gates.

Additional extensions in the above mappings are for example some new types. One type that has been added in SDT is the type Octet that is actually part of (ITU-T Recommendation Z.105), and which is used to implement the IDL type octet.

2.6 Mapping Object Models to IDL

In Section 2.2 it was hinted that it is possible to paste an object model as IDL; the transformation rules that are considered here are very simple and straightforward. The strength of this approach is that for each object that is pasted, an implink is created between the copied class and the IDL entity, thereby facilitating for example trace mechanisms.

Classes

An object model class can be mapped to either a module or an interface. In the former case, the resulting module is always empty, even if the class contained both attributes and operations. In the latter case, however, all attributes and operations are mirrored using IDL syntax as far as possible. This latter mapping is shown in Figure 3. The comment that is included in the mapping is used to contain the optional implink that is created between the class and the interface.

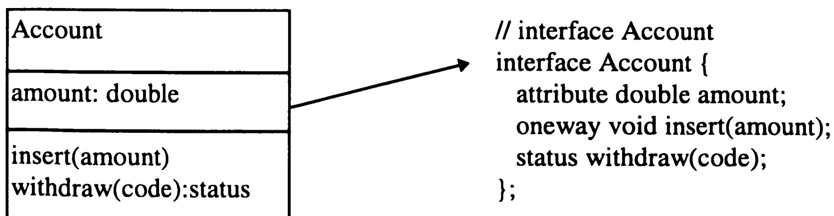

Figure 3 Mapping an object model class to an IDL interface.

It is not possible to express all information that is needed in the IDL description simply by mapping a class like this; some manual additions must always be made, such as type definitions, constants, and exceptions, but also simple matters like defining for each operation parameter whether it is an in, inout or out parameter.

Furthermore, it might be necessary to define an attribute as readonly, or whether an operation is oneway or not. All operations are considered asynchronous by default, unless it can be determined that it should be synchronous. This is done by either defining a return type for the operation, or by inserting {sync} or {async} after it.

Inheritance

The mapping of object model class inheritance is only applicable when an object model class is mapped to an interface, and in that case the class inheritance is mapped as interface inheritance, as is shown in Figure 4.

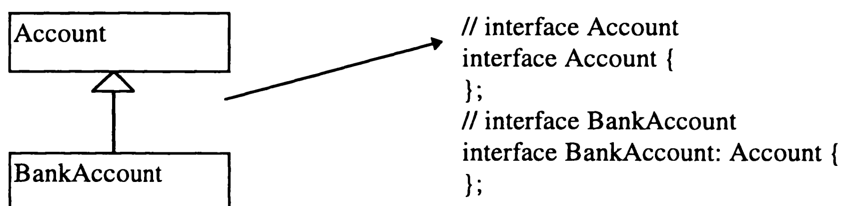

Figure 4 Mapping object model class inheritance to IDL inheritance.

Aggregation

When mapping aggregations, all object model classes that are not leaves are mapped as modules. However, the leaves themselves may be either modules or interfaces, and it is the responsibility of the user to decide which. Aggregations are thus mapped to nested modules, where the innermost layer may be either interfaces or modules. An example of aggregation mapping is shown in Figure 5.

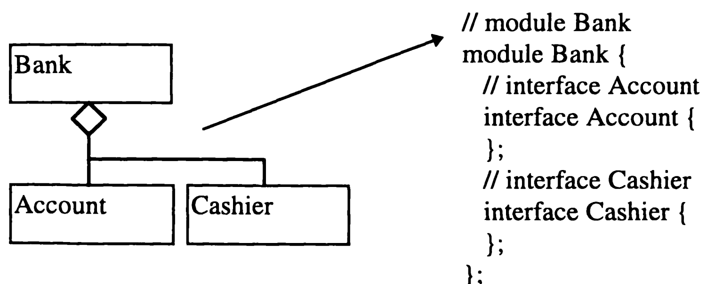

Figure 5 Mapping object model class aggregation to IDL.

3 IMPLEMENTING AN SDL APPLICATION

In this section the implementation of the CORBA-oriented approach is discussed in more detail. The created SDL application is supposed to be working as a client/server in a heterogeneous, distributed environment as only one of many such parts, i.e., the system has been decomposed into lesser parts that may or may not be implemented using different programming languages; the glue between all of these parts, however, is CORBA.

The tools that have been used in the implementation of the CORBA-oriented approach are SDT3.1 for modelling SDL and Orbix2.1mt (from IONA Technologies, Ltd.) to provide the CORBA implementation.

3.1 The system architecture

An SDL application can be used as both a server and a client, which is illustrated architecturally in Figure 6.

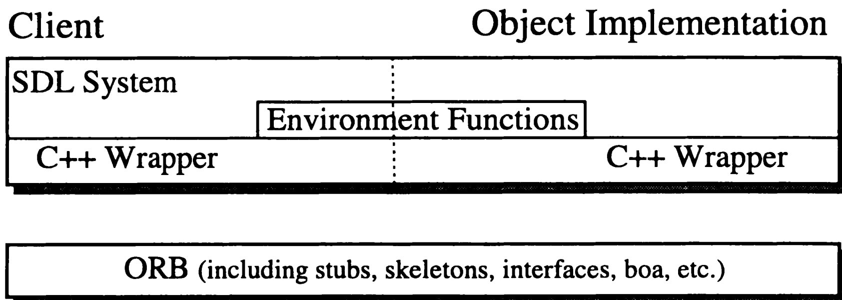

Figure 6 The system architecture of an SDL application.

An ordinary SDL application without the CORBA support only consists of the SDL system itself along with its environment functions that are used to connect the system to the outside world. In the CORBA-oriented approach, a wrapper is placed around the SDL system to imitate the behaviour of objects in a way that makes sense to the ORB. The C++ wrapper and its functionality is further discussed in Section 3.2. Orbix supports the C++ mapping of IDL, which is why the wrappers have been implemented using C++.

Based on the development process depicted in Figure 1, the development of the object implementation side starts with an IDL description. The complete process is then as follows:

1. The IDL description is used as the basis for an SDL stub, which is generated by an SDT specific IDL compiler. At the same time, a C++ wrapper for the object implementation side is also generated using the same IDL compiler.

2. The ORB also uses the IDL description to generate ORB specific C++ code, but has an ORB specific IDL compiler for this purpose.
3. A developer then provides the behaviour of the SDL system stub.
4. If any services from other servers are required (on the client side of the SDL application), their IDL descriptions should be converted to SDL using the SDT specific IDL compiler. Code is also generated for the C++ wrapper. As for the ORB, it uses its own IDL compiler to generate the necessary C++ code.
5. C code represented the designed SDL system is generated using the SDT C Code Generator.
6. The environment functions are predefined, and are always the same regardless of the SDL (CORBA) application; all system specific information is placed in the wrapper.

When the C and C++ code has been generated as above it is compiled and linked with a set of precompiled libraries (including an SDL kernel providing runtime support of the SDL system, as well as ORB specific libraries) to form an executable SDL application.

These steps are shown graphically in Figure 7.

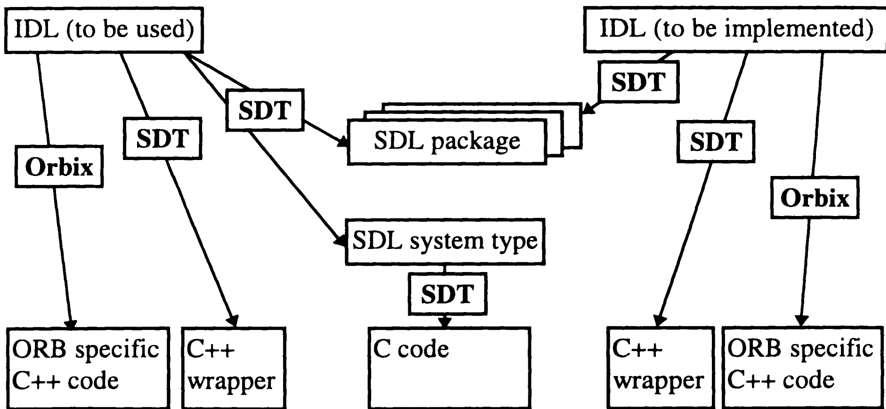

Figure 7 Generating code for the SDL application.

3.2 Wrapping the SDL system

In order to get a functioning SDL application, the behaviour of the SDL system wrapper is especially important. The wrapper can be said to consist of two parts: the environment functions and the class definitions that are generated from the IDL descriptions. The C++ wrapper on the client side has a behaviour that is very similar to the behaviour on the client side, even though there are some unique features, such as the need to be able to locate objects for particular requests.

Scheduling

Since the SDL system is capable of acting as a client, it has its own scheduling mechanism. Requests that made to other servers are sent through an environment function to the C++ wrapper, where the request is processed and then passed on through the ORB to the intended server.

At regular intervals, the scheduler calls another environment function to check whether there are events pending from the ORB. Should such an event be present, it is processed, and then sent to the appropriate process instance.

The entire SDL application is scheduled within a single UNIX process, and requests that are made internally in the SDL system are not managed by CORBA. Contrast this with the SDL-oriented approach, where all requests would be sent through CORBA, possibly from multiple UNIX processes.

Environment functions

The environment functions are responsible for managing the communication between the ORB and the SDL system. Two of these environment functions have already been mentioned above as responsible for passing requests to and from the SDL system.

Another environment is used to initialise the SDL system and its environment. In this case, it is particularly used to initialise the communication with the ORB.

The environment functions uses buffers to store all requests before they are treated, both for incoming and outgoing requests.

Objects versus process instances

When the ORB communicates with the object implementation, it only sees the C++ wrapper and the objects that are defined within it; the SDL system itself is completely hidden.

The SDL system, on the other hand, controls which objects that are present in the C++ wrapper by mirroring each process instance as an object. Whenever a new process instance is created in SDL a new object representing that process instance is created in the wrapper.

When a process instance is terminated, the object is also removed (when simulating the SDL system the object is preserved throughout the lifetime of the server in order to allow a trace of failed requests).

An object that receives a requests passes it along to its corresponding process instance (it first has to convert the request into a suitable format, i.e., a signal or remote procedure).

In SDL, all object references are represented using PId values. A C++ client, however, would still access the process instance of an SDL object implementation using its ordinary object references.

Multithreading

In order to handle requests correctly, it has been necessary to make use of a multithreaded ORB.

On the object implementation side, each new request from the ORB must be treated in a thread of its own. The reason for this is that a synchronous operation is mapped to a remote procedure in SDL, which in turn is implemented as the sending of two asynchronous signals (call and reply). When the request is received, a call signal is sent into the SDL system. While the operation is waiting for a reply, the SDL system must be able to execute other requests, which is not possible unless each request is executed in its own thread, while the SDL system is scheduled in a main thread.

On the client side, the situation is very similar. While waiting for a reply to a request made to the ORB, the SDL system must be able to continue to execute, which means that all such server requests must also be executed in separate threads.

The SDL system thus executes in one thread, and each request that is received by or sent from the SDL system results in a new, detached thread which disappears once the request has been handled.

The ORB is responsible for creating the appropriate threads for requests to the SDL system, while the C++ wrapper is responsible for creating threads for requests aimed at other servers. The C++ wrapper must also provide a sufficiently multithread-safe environment.

The C++ implementation classes

As part of the ORB specific code, a set of IDL C++ classes are generated which represents the IDL interfaces. It is then up to the developer to provide C++ implementation classes that implements these IDL C++ classes. The ORB provides mechanisms for connecting the IDL C++ classes and the C++ implementation classes to each other. In Orbix, either the BOAImpl approach or the TIE approach can be used for this task, and due to its greater flexibility the TIE approach is used in the SDT integration. Each IDL C++ class must have at least one corresponding C++ implementation class.

In the C++ wrapper, the C++ implementation classes are the most important part, as they provide the 'behaviour' of the SDL system. A request that is made to an object is passed to the appropriate process instance after having been processed. This processing is specific depending on whether the operation is asynchronous or synchronous. Due to the increased complexity when dealing with synchronous operations it is considerably easier to manage asynchronous operations.

For an asynchronous operation, the following steps have to be performed by an object:

1. Allocate memory for the signal.
2. Convert the parameters of the operation to SDL signal parameters.
3. Send the signal to the process instance corresponding to the current object.
4. Return, i.e., exit the request thread.

For a synchronous operation, on the other hand, some additional steps are necessary:

1. Allocate memory for the remote procedure call.
2. Convert the parameters of the operation to SDL remote procedure call parameters.
3. Send the remote procedure call to the process instance corresponding to the current object.
4. Wait for a reply, i.e., block the request thread until the appropriate remote procedure reply is received. It is necessary to pass information about the current thread's identity in the call/reply signals to ensure that the appropriate thread receives the correct reply (since the order in which replies are received is not guaranteed to be the same as the order in which they were sent).
5. Convert the SDL remote procedure reply parameters to C++ parameters.
6. Release the memory held by the reply.
7. Return, i.e., exit the request thread, and pass the obtained data back to the client.

Locating a server object

One particular problem that must be addressed is how an server object is located. In SDL, a request can be made either to a specific PId value, or without specifying a particular receiver. In the first case, the object reference of the receiver is already known, and there is no problem. In the second case, however, an appropriate receiver must first be located. There are two approaches to this problem:

- A bind concept can be introduced in SDL which would be responsible for finding a server implementing the required request. However, this solution does not remove the problem with requests that are made without specifying a receiver.
- When the C++ wrapper receives a request with no apparent receiver, an ORB specific bind command to find an appropriate server object can be performed. Once the request has been performed this object reference can then either be stored for subsequent requests of the same kind to reuse, or it can be released, thereby requiring a new bind command for each new request.

4 CONCLUSIONS AND FUTURE DIRECTIONS

The implementation that has been performed so far on the CORBA-oriented approach is by no means complete (as some IDL concepts are not yet supported), but still it must be considered an important contribution to the ways an interface can be implemented. Above all, this implementation shows that it is possible to use SDL in conjunction with CORBA using the CORBA-oriented approach. Together with (Olsen, 1995), where the SDL-oriented approach is demonstrated, this makes

SDL an important language to consider when dealing with distributed systems of different kinds.

One key issue of the CORBA-oriented approach is the that mapping rules between IDL and SDL need to be standardised. Currently, there are at least two different mapping proposals, and some work will have to be performed to gain a consensus on these, possibly through standardisation work in OMG or ITU.

As a continuation of the work performed hitherto, it is necessary to support all IDL concepts in SDL, and perhaps to update the methodology that has been presented in this paper.

A natural extension of this work would be to examine the SDL-oriented approach in more detail, providing methodology guidelines and implementations for when SDL should be targeted to CORBA. It is not expected that it will be possible to define general mapping rules for mapping SDL to IDL, as these will probably be implementation dependent.

5 REFERENCES

Björkander, M. (1996) Mapping IDL to SDL, Telelogic AB, Malmö.

Born, M. and Winkler, M. and Fischer, J. (1996) Formal Language Mapping from CORBA IDL to SDL'92 in Combination with ASN.1, Humboldt University, Berlin.

ITU-T Recommendation Z.100 (1995) CCITT Specification and Description Language (SDL). ITU-T, Geneva.

ITU-T Recommendation Z.105 (1995) SDL Combined with ASN.1 (SDL/ASN.1). ITU-T, Geneva.

Olsen, A. and Jorgensen, B.M. (1995) Using SDL for targeting services to CORBA, in *Bringing Telecommunication Services to the People, 3rd International Conference in Broadband Services and Networks* (ed. Clarke, A. and Campolargo, M. and Karatzas, N.), Springer-Verlag.

OMG (1996) The Common Object Request Broker: Architecture and Specification, Object Management Group, Framingham, MA (USA).

Telelogic (1996) The SOMT Method, in *SDT3.1 Methodology Guidelines: Part 1*, Telelogic, Malmö.

6 BIOGRAPHY

Morgan Björkander is a developer at Telelogic AB, and is mainly responsible for the CORBA support in SDL applications generated by SDT. He has also been involved in different RACE and ACTS projects focusing on service creation, for example SCORE and TOSCA.