1
Finite State Machine Based SDL

Dr. Richard O. Sinnott (GMD-FOKUS Berlin, Germany) and
Prof. Dieter Hogrefe (University of Lübeck, Lübeck, Germany)

1.1 Introduction

SDL [13] is a language for specifying and describing systems. The basic idea of SDL is to describe systems in the form of asynchronously communicating processes represented as extended finite state machines. For this reason SDL is particularly suited to model and develop parallel, e.g. distributed, communicating systems.

In this chapter we do not give a full presentation of all features of SDL. Rather we introduce those aspects of the language which will subsequently be used for emphasising the applicability of SDL together with its associated tool for developing specifications of distributed systems. To this end, we develop a specification of a component crucial to realising the dynamicity inherent to distributed systems: a trader. Specifically we show how the OMG trader [8] can be specified using SDL and its associated tools. Other examples of formal specifications of traders are given in [4, 6, 20, 23].

For a more detailed description of the SDL language the reader is referred to the recommendation Z.100 [13], one of the tutorial books, e.g. [5, 18, 19] or introductory articles [3]. As an aside we note that SDL has a wide body of literature associated with it. It is likely that this, along with its intuitive syntax and the availability of numerous tools for developing and reasoning about specifications [9, 24, 25], is the reason that SDL is one of the most popular specification languages around today.

1.1.1 History of SDL

Towards the end of the sixties it was identified that in many areas natural language was inadequate for describing complex behaviours. This was especially the case in telecommunications systems, where increasingly complex functions were required to be described and exactly interpreted. This problem was exacerbated by the international nature of telecommunications, i.e. where the description of these functions was not always in the mother tongue of the different manufacturers expecting to implement the functions, yet with the strict requirement that these implementations were expected to interwork with one another.

To address this problem, the International Telephone and Telegraph Consultative Committee (CCITT) – who are now called International Telecommunications Union (ITU-T) – identified that a standard specification language was needed. This language was to be used to precisely describe complex situations and be intelligible so that readers world-wide could interpret it unambiguously and uniquely. This was only feasible if the syntax as well as the semantics of the language was internationally standardised.

One of the fundamental design criteria in the development of SDL was its ease of use. As an aside we note that ease of use is an issue often ignored by developers of formal specification languages, leading to the reputation that formal methods are too often overly mathematical and obscure! A shortcoming that cannot be applied to SDL with its intuitive graphical notation.

All innovations that were incorporated into the initial design of the SDL language originated from groups of people involved in the specification of complex systems. Primarily these were employees from telecommunications companies involved in both hardware and software development. Universities and other scientific institutes initially played a subordinate role in the development of SDL. It wasn't until the end of the seventies, with the appearance of scientific institutes largely dedicated to telecommunications, that work on the mathematical underpinnings of SDL was made. This work was necessary for three main reasons. Firstly, several initial and incompatible versions of the language existed, with each having features incorporated for usage in their own particular problem domains. This was primarily caused by the intuitive interpretation model upon which the language was based which allowed for ad-hoc extensions and modifications. The effect of this was that almost twenty years went by before SDL reached a stable state. This period was especially problematic for users since existing specifications frequently became obsolete or invalid due to language changes.

Secondly, the development of language supported tools is very difficult if the language is not clearly defined. In particular, tools for the automatic analysis of specifications cannot be developed under these circumstances. It is also extremely difficult to "correctly" (whatever this term may mean) specify complex systems when the language used to describe those systems is itself not clearly defined.

Thirdly, the real objective of developing an artificial language for the description of complex systems precisely and unambiguously could not be attained if the language itself was open to interpretation. These three problems gave the impetus for embracing a formal syntax and semantics. The CCITT study group 10 published the first formal SDL language version in 1988. In 1992 this version was enhanced by object-oriented features which we discuss in Section 1.1.9. 1996 was a year of consolidation with few major changes but mostly error correction. SDL 2000 [14] brings more radical changes to meet the demands of the users. These include a new data model, harmonisation with the data typing language ASN.1 [10, 11] and alignment with the Unified Modelling Language (UML) [2]. Other useful features such as remote process creation have also been introduced.

We note that these language updates and improvements that take place every four years are another crucial factor to the success of SDL. The language and hence the associated tools evolve to incorporate current software development approaches. We focus in this paper on how the language and associated tools address developments in the area of object-oriented distributed systems.

The application range of SDL is broad. Prominent application examples within the standard area are the ISDN protocols [1], the signalling system No. 7 [16], intelligent network services [17] and next generation telecommunication services [21, 22]. Almost every major manufacturer of telecommunication systems uses SDL in one or the other form.

We begin with a brief overview of SDL itself.

1.1.2 Basic Concepts of SDL

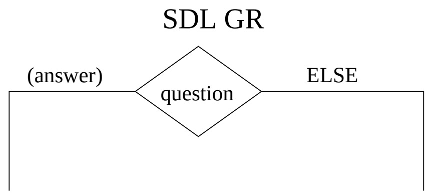
The basic element in SDL is the process. A process is represented in SDL as an extended finite state machine that can communicate with other processes and the environment of the specification by sending and receiving signals via channels and signal routes. An extended finite state machine differs from a finite state machine in that it can store not only its states but hold and manipulate data. This data exists in the form of values and variables.

The model on which SDL is based is determined by the characterisation of a process. A process is either undergoing a state transition or in a state and waiting for an input. There is exactly one input queue for each process. When the process receives an input during a state transition, the input can be stored in the input queue. The input queue and the process operate independently and in parallel. If two inputs reach a process simultaneously, they are buffered in the input queue in, in the general case, random order.

The dynamic behaviour of an SDL specification is described exclusively by processes that coexist with one another. A process can create another process with the created process immediately becoming equal in ranking to the creating process. A process can only disappear from the system through self-termination.

1.1.3 The Two Syntactic Forms of SDL: SDL/GR and SDL/PR

The SDL language has two syntactic forms, both of which are based on the same semantic model. One is called SDL/GR (SDL Graphical Representation), the other SDL/PR (SDL Phrase Representation). Each language element has a representation in SDL/GR and one in SDL/PR. An example of this is given in Figure 1.1 which illustrates a simple alternative that can be used within the specification of a state transition.



SDL PR

```
DECISION question
  (answer) : ...
  ELSE : ...
ENDDECISION;
```

Fig. 1.1. Simple alternative in SDL/GR and SDL/PR

A specification written in SDL/GR can be transformed exactly and semantically fully equivalent to an SDL/PR representation. SDL is one of the few languages that has a completely defined graphical syntax. Initially, only the graphic form of SDL existed since it was assumed that graphics would be a user friendly specification method.

With the increasing popularity of SDL however, there was a growing need for machine-based document processing. Due to the lack of high-quality graphic terminal equipment, the SDL documents had to be processed largely by hand, a very time consuming effort, particularly when modifying a document. This gave rise to the development of SDL/PR.

Today, SDL/GR can also be machine processed and SDL/PR appears to be superfluous. Even if this were the case, the existence of SDL/PR still has a positive side effect. Without a conventional form of the language, similar to programming languages, the development of a formal syntax and semantics would probably not have been feasible. We note that not all language element are represented differently between SDL/GR and SDL/PR. A subset of SDL/GR is syntactically equivalent to a subset of SDL/PR, e.g. the declaration of variables in both representations is identical. The examples in the following sections are limited to SDL/GR.

1.1.4 States and State Transitions

As stated, a process in SDL is given as an extended finite state machine, i.e. states and state transitions are used for describing its behaviour. In the theory of finite state machines, states and state transitions are often represented with the aid of directed graphs with the nodes representing states and the directed edges representing the state transitions. In the example given in Figure 1.2 which illustrates the state transition behaviour of a protocol entity during the connection set-up phase, the finite state machine has the state set $disconnected, waiting, connected$.

When an input is received, the finite state machine can leave its current state, attain a new state, and as a result produce an output. Such a state transition is represented in the graph by a directed edge, starting from the old state to the new state and the pair $Input/Output$.

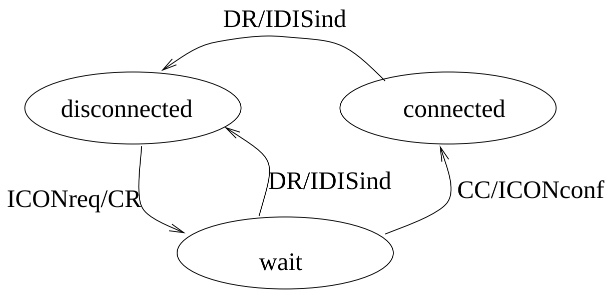


Fig. 1.2. State Transition Diagram for Simple Protocol

In SDL states are graphically described with the state symbol shown in Figure 1.3. Inputs, outputs and starting states also have special symbols defined.

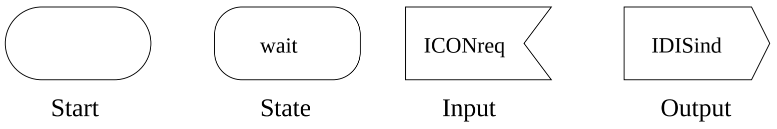


Fig. 1.3. SDL Constructs Associated With State Transitions

With the aid of finite state machines behaviour can be described. In the graphic description of the finite state machine in Figure 1.2 the viewer can at best implicitly determine that the disconnected state should be the first state. Generally, intuition cannot be relied upon. For defining the initial state there is a start symbol in SDL which is also shown in Figure 1.3.

Figure 1.4 shows how the behaviour in Figure 1.2 would be described in SDL in the form of a process. The frame around a process diagram is optional, if no further information is located in a document on the same side. If additional text is located on the same page, the frame clearly defines what actually belongs to the process.

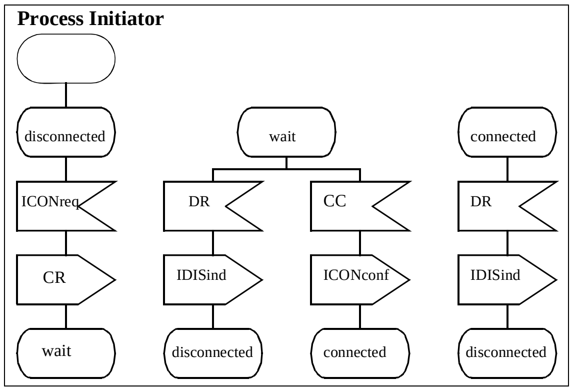


Fig. 1.4. SDL Representation of Simple Protocol

In SDL there are various possibilities to prevent repetition of identical specification sections. For example when input DR is received by the initiator process in states wait and connected, the same output is given (IDISind). To prevent this repetition multiple states can be defined in a single state symbol as illustrated in Figure 1.5 which shows an alternative but equivalent specification for initiator.

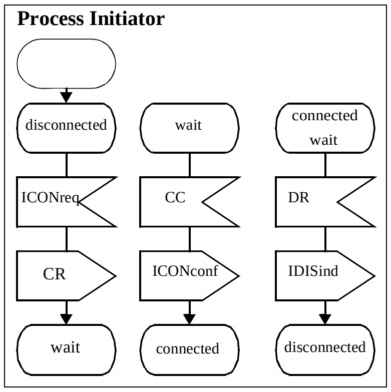


Fig. 1.5. Alternative Representation of Simple Protocol

A second possibility to avoid text repetition is the asterisk (*) notation. An asterisk in a state symbol signifies that the following inputs are possible in any state. In addition there is the dash notation (-). A - in a state symbol at the end of a transition means that a transition to the same state occurs. These examples illustrate that certain "syntactic sugar" exists in SDL to help make the life of a specifier easier.

1.1.5 Time Definition in SDL

In a specification, time conditions often need to be specified. In SDL this is implemented with the timer mechanism. A timer is similar to a signal. The timer mechanism stimulates a process as a function of the predefined time by placing a timer signal into the input queue of the process. The timer mechanism is explained through an example.

In our protocol specification the wait for the connection confirmation CC is to be timed. Upon expiry of the waiting period the connection is to be cancelled automatically. Figure 1.6 shows the required extensions to the initiator process given previously.

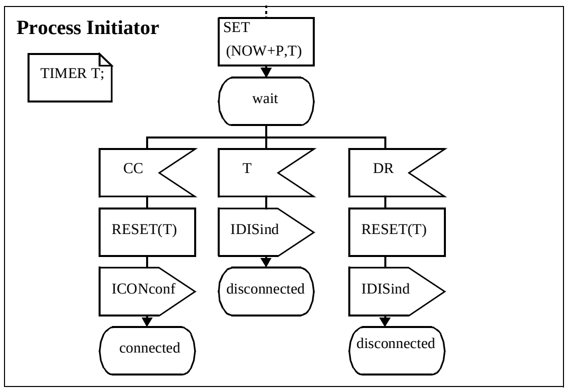


Fig. 1.6. Timed Extension to Simple Protocol

First a timer must be defined in the process diagram. This is done using a text symbol. This text symbol can contain also other information. e.g. the declaration of variables. Any number of timers can be defined in a process.

1.1.6 Declaration and Use of Data

As an extended finite state machine, a process can hold and manage data represented in the form of values and variables. The declaration of data in SDL/GR and SDL/PR is identical and initiated with the keyword DCL. Like the timer definition, a variable declaration is written into a text symbol which can be positioned anywhere in the diagram.

During a state transition a process can use and manipulate its local data. Data manipulation can be specified with the TASK symbol which can perform assignments as illustrated in Figure 1.7 where the counter variable is assigned the value 0. Of course the type must be compatible.

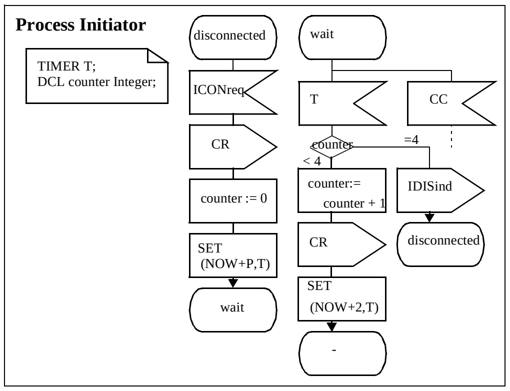


Fig. 1.7. Example of the Use of Data

State transitions can be controlled with the aid of variable values. Often it is desirable to specify different state transitions as a function of the content of a variable. In SDL this is implemented with the alternative construct, as shown in Figure 1.7. The question to be resolved through an alternative normally involves an operation on one or more variables. At the time the question is interpreted, one correct answer must be possible, otherwise the specification is no longer interpretable from this time forward. To ensure that at least one correct answer is possible, the ELSE clause is used in SDL if none of the other answers are correct.

1.1.7 Signals and Data

In Section 1.1.4 the concept of input and output in conjunction with the state transition behaviour of a process was introduced. In SDL, the "messages" related to an output and the corresponding input are called signals. Signals were used in Section 1.1.4 for initiating state changes. In SDL, inputs and outputs can also be used for transmitting data from one process to another. In this case the output has one or several assigned values. During the corresponding input, variables are specified that store the value of an incoming signal. Of course the types of the transmitted variables must agree with the types of the variables at the receiver. This is illustrated in Figure 1.8.

1.1.8 Specification Structuring and Process Communication

An SDL specification defines an abstract machine that receives inputs from its environment and produces outputs to the environment. This abstract machine is referred to as a system. The system contains everything that

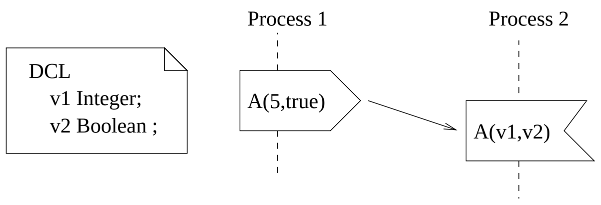


Fig. 1.8. Data Transfer in Signals

should be defined in an SDL specification, but nothing that should not be defined.

The system communicates with the environment via channels. From the viewpoint of the system, the objects in the environment have a process-like behaviour which means that communication with them is possible in the usual manner. The channels through which the system communicates with the environment form the logical interface to the environment.

The system itself consists of one or several blocks that communicate with each other and with the environment through channels. The blocks are the logical system components which provide specification structuring. A block can consist of several other blocks. This results in a tree-like structure. Ultimately, blocks contain processes.

Channels can be unidirectional or bi-directional. The communication direction is identified by an arrow. Near the arrows the names of the signals or potentially the name of the list of signals (signallist) that can be transmitted in that direction, must be given. An example of a basic SDL system is given in Figure 1.9, where for clarity the signals are omitted from the channels and signalroutes.

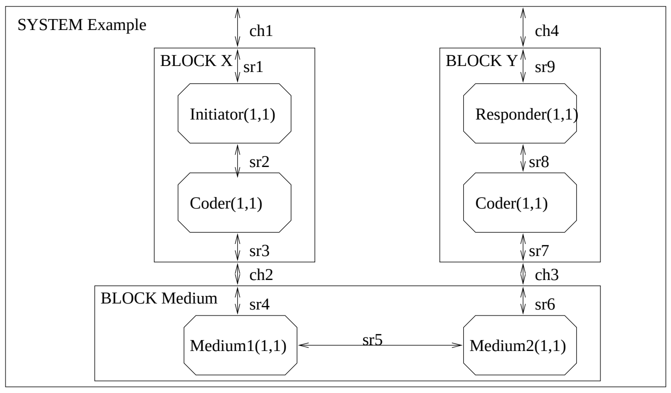


Fig. 1.9. Structuring of Basic SDL System

A channel can optionally delay the transport of a signal. A non-delaying channel has the arrows at the end, e.g. touching the blocks, whereas a delaying channel has the arrows somewhere in the middle. For delaying channels, a FIFO delay queue is associated with each channel direction. When a signal is transmitted to a channel, it is inserted into the corresponding delay queue. After an undefined but finite time, the first signal is removed from the delay queue and appears at the end of the corresponding channel.

The structure of the communication between processes is similar but the connections are called signalroutes. Signalroutes do not delay the transportation of signals. They can be created and deleted dynamically with the creation and deletion of a process.

Processes can only communicate with one another using signals if the appropriate signalroutes and channels (if the processes are in different blocks) exist. To overcome this restriction, SDL has introduced the concept of remote procedures. These can be exported and subsequently imported by blocks or processes. The advantage of remote procedures are that they facilitate communication between processes without the need for explicitly denoting the channels, signalroutes and associated signals to be carried. There are some disadvantages in the use of remote procedure calls in SDL though, e.g. the client side is blocked during the remote procedure call and they do not support exceptions.

There is clearly more to say about the semantics of communication, creation and termination of processes then we have space for here. Furthermore, a number of items in the example of Figure 1.9 have not been explained, e.g. the (1,1) behind the process name means that one instance of the process exists initially and the maximum number of instances is also one. The reader is referred to [13] for more explanation.

1.1.9 Object-Orientation in SDL

In 1992 SDL was enhanced by object-oriented features. This included specific language constructs and concepts for supporting object-orientation. In particular SDL supports interfaces, objects, classes, inheritance and subtyping. It should be noted that SDL uses a different terminology for these concepts for historical reasons. What in object-orientation traditionally is called a class, is called a type in SDL. Objects are called instances in SDL. We discuss the representation of objects and interfaces in SDL in section 1.2.

SDL defines various kinds of types including: system, block, process, service, procedure, signal and data. Instances of these types can be created (instantiated) which will have the same data structure and behaviour. As well as instantiation, types can be specialized as new types, e.g. definition of subtypes of a type. SDL is unique in comparison with most other object oriented languages in the sense that SDL offers numerous possibilities for specialised behaviour. In most other languages specialising behaviour is accomplished by redefining (overloading) virtual methods/operations in subtypes. In SDL specialisation of behaviour can be accomplished in numerous ways, e.g. by adding new transitions to a process type, or redefining virtual procedures. SDL also allows for constraints to be given on the specialisations, e.g. using the at least clause.

To allow type specifications to be used across several different specifications, type specifications can be placed in packages. Packages can then be then referred to in the scope of where the type specification is going to be used. Furthermore, to allow for generic type specifications, a type specification can be parameterised with so-called formal context parameters.

Consider the example of Figure 1.9. Before 1992, when object orientation was not part of SDL, the processes Medium1 and Medium2 had both to be specified completely, even if they were really the same, and just connected to two different signal routes. With the object oriented features of SDL 92 the same specification would look as in Figure 1.10.

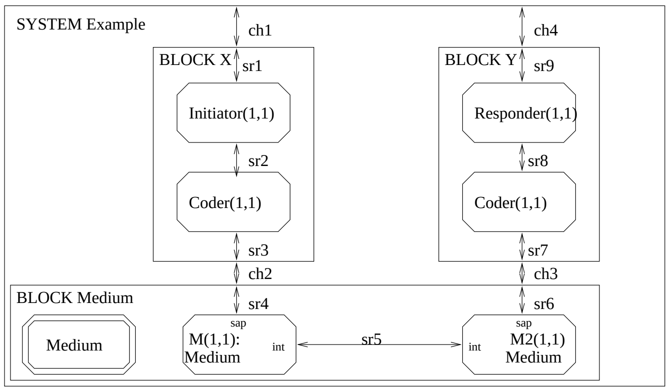


Fig. 1.10. Object Based Structuring of SDL System

Please note that the examples of Figure 1.9 and Figure 1.10 are incomplete and serve only as an illustration of the basics concepts. For example, process Coder is assumed to exist elsewhere in the specification.

In the example of Figure 1.10 the process type Medium is instantiated twice. In order to specify precisely how such an instance is connected to the infrastructure, i.e. through channels and signalroutes, so called gates have to be introduced. In this example these are called sap and int.

1.2 Applying SDL to Develop a Trader Specification

In this section we show how SDL and its associated tools can be applied to develop a trader specification. We begin with an informal description of trading services.

1.2.1 Introduction to Trading Services

Perhaps the key aim of distributed systems is to provide distribution transparent utilisation of services over heterogeneous environments. In order to use services, users need to be aware of potential service providers and to be capable of accessing them. Since sites and applications in distributed systems are likely to change frequently, it is advantageous to allow late binding between service users and providers. If this is to be supported, a component must be able to find appropriate service providers dynamically. The concept of trading has arisen to provide this dynamic selection of service providers at run time. The interactions that are necessary to achieve this are shown in Figure 1.11.

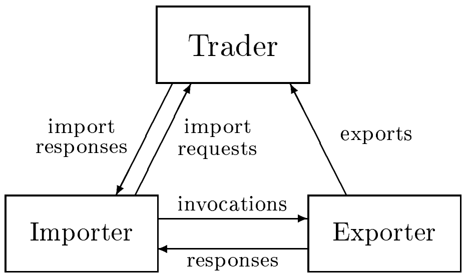


Fig. 1.11. A Trader and its Users

Here a trader accepts a *service offer* from an *exporter* wishing to advertise its services. A service offer contains the characteristics of a service that a service provider is willing to provide. We note here that the service provider need not necessarily be the exporter. Similarly the service user may not be the importer. The trader then stores these service offers for use by importers.

A trader accepts service requests from importers of services. These represent requirements on available services that a trader may or may not have access to. Upon receipt of a request from an importer, the trader searches its store of service offers to see if any offers match the importer's service request. If any matching offers are found they are returned to the importer, which may then interact directly with the service.

It should be pointed out that a trader might not itself have a service offer that matches an importer's request. In this case, a trader can check whether any other traders it "knows" might satisfy the import request. This is known as federated trading. For brevity, we consider only non-federated trading.

The OMG trading object service specification [8] identifies particular interfaces and associated operations that a trader should support. For brevity, we focus on two of these in particular and the two most basic operations required for trading: namely export and import†. These operations are offered as part of the *Register* and *Lookup* interfaces respectively. We also note that for simplicity we do not consider the multiple interfaces that are inherited by these interfaces. This simplification avoids the lack of multiple inheritance in SDL. It is possible to manually edit the IDL inheritance hierarchy to overcome this problem however.

The actual interfaces and operations themselves are defined in [8] using a combination of CORBA IDL and informal textual description. The *export* operation is given as:

```
module CosTrading –
        interface Register –
            // definition of data types

            OfferId export (in Object reference,
                     in ServiceTypeName type,
                     in PropertySeq properties)
                     raises
                      ( InvalidObjectRef,
DuplicatePropertyName,
                       // ... other exceptions );
            // other operations ...  ";
        // other interfaces ...";
```

The parameters associated with this operation include the $reference$ that can be used by a client (importer) to interact with that service. We note that the term *Object* is used here. In comparison with other object models, e.g. the ODP model, this is really an interface reference. The *type* parameter identifies the service type which contains the interface type of the reference

† or *query* as it is known in the OMG trader object specification

and a set of named properties that may be used in further describing the offer, i.e. it restricts what is acceptable in the *properties* parameter. The *properties* parameter is a list of named values that can be used for describing behavioural aspects, non-functional or non-computational apsects of the service offer. These properties must agree with those described in the *type* parameter.

The $OfferId$ returned for a successful export is the handle which can be used by the exporter to identify the exported offer during other operations, e.g. to withdraw or modify the offer.

Various exceptions can be raised by the trader upon invocation of this operation. For brevity we consider two: the *InvalidObjectRef* exception which is raised if an invalid reference is supplied, e.g. a *nil* reference is supplied; the $DuplicatePropertyName$ exception which is raised if the exporter submits two or more identical property names in the properties parameter.

The *query* operation of the *Lookup* interface is represented as:

```
module CosTrading –
    interface Lookup –

        // definition of data types

        void query ( in ServiceTypeName type,
                in Constraint constr,
                in Preference pref,
                in PolicySeq policies,
                in SpecifiedProps desired˙props,
                in unsigned long how˙many,
                out OfferSeq offers,
                out OfferIterator offer˙itr,
                out PolicyNameSeq limits˙applied )
            raises (
                // ... exceptions );
        // other operations ... ˝;
```

Here the importer supplies the *type* of the service that they are searching for. This parameter is crucial for future type safe interactions between importers and exporters. The *constraint* parameter can be used by the importer to capture aspects of the service they are looking for that are not represented in the signature of the service type. The $preference$ parameter is used to order offers that satisfy the *constraints*, i.e. so that they are presented in order of greatest interest to the importer. The *policies* parameter allows the importer to influence how the trader performs the search for compatible service offers. The *desired_props* parameter defines the set of properties to be returned with the matching object reference.

The *how_many* parameter can be used to state how many offers are to be returned.

The result of the *query* are a sequence of matching $offers$ and a reference to an interface $offer\_itr$ where other matching offers can be accessed. If the search was subject to any restrictions, e.g. policies related to cardinality limits were imposed, then the names of these policies will be returned. There are numerous possible exceptions that can be raised depending upon the parameters associated with the *query* operation. For brevity we do not discuss them here. The reader is referred to [8].

1.2.2 Development of the Trader Specification

Whilst it is quite possible to develop a specification of a trader through interpretation of the IDL and textual description given previously such an approach is not ideal. A better approach is to use tool support to automatically translate the IDL description to the appropriate target specification language. This is in accordance with the usage of implementation languages such as C++, Java etc for development of CORBA based distributed systems. The result of such an automatic translation should be client stubs and server skeletons that capture the syntactic aspects of the communication given by the IDL.

SDL is one of the few specification languages for which such an IDL mapping has been defined. Indeed, more than one mapping has been defined and implemented by different tools and tool vendors [7, 24]. We consider the mapping used by the Y.SCE tool [7] since it supports exceptions — a crucial feature present in nearly all IDL descriptions of distributed systems. An outline of the IDL to SDL mapping is presented in ??.

Before a mapping can be made, however, it is necessary that the syntax of the IDL is compatible with the syntax of the language to be mapped to. In the previous description, various SDL keywords are present which hinder the generation of syntactically correct SDL – this is unsurprising given that there are so many keywords in SDL. Specifically, the variable name *type* and the operation name *export* are reserved words in SDL. It is thus necessary to modify the IDL to overcome this problem, e.g. replace the variable named *type* with one named *atype* and the operation *export* with $trader\_export$.

The Y.SCE tool [7] generates packages ($name\_interface$ and $name\_definition$) which provide mappings for the IDL operations and parameters and the client stubs and server skeletons. We note here that tools such as [24] already provide an existing package (*idltypes*) which contains a mapping for many of the basic CORBA types such as short, long, Boolean etc. A snapshot of the resultant SDL code generated from this mapping focusing on the signals associated with the export operation as present in the *name_interface* package is shown in Figure 1.12.

```
signal pCALL_CosTrading_Register_export(CORBA_Object, CosTrading_ServiceTypeName,CosTrading_PropertySeq);
signal pREPLY_CosTrading_Register_export(CosTrading_OfferId);
signal pRAISE_CosTrading_Register_InvalidObjectRef(CosTrading_Register_InvalidObjectRef);
signal pRAISE_CosTrading_DuplicatePropertyName(CosTrading_DuplicatePropertyName);
signallist CosTrading_Register_INVOCATIONS = pCALL_CosTrading_Register_export, ....;
signallist CosTrading_Register_TERMINATIONS = pREPLY_CosTrading_Register_export,
                                              pRAISE_CosTrading_Register_InvalidObjectRef,
                                              pRAISE_CosTrading_DuplicatePropertyName, ...;
```

Fig. 1.12. Mapping for Trader IDL Operations

The resultant SDL server skeletons as found in the *name_definition* are represented in Figure 1.13. We note that here we show only those aspects of the *Register* interface dealing with the export operation. In reality, virtual procedures for all of the operations would be present along with the appropriate behaviour that enables the local procedures to be called.

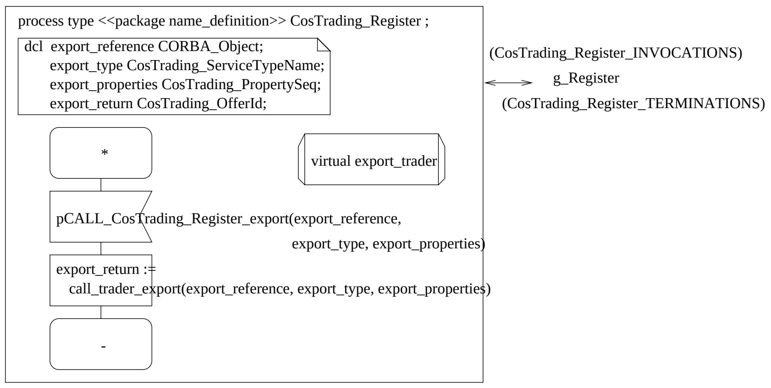

process type <<package name_definition>> CosTrading_Register ;
dcl export_reference CORBA_Object;
export_type CosTrading_ServiceTypeName;
export_properties CosTrading_PropertySeq;
export_return CosTrading_OfferId;
(CosTrading_Register_INVOCATIONS)
g_Register
(CosTrading_Register_TERMINATIONS)
*
virtual export_trader
pCALL_CosTrading_Register_export(export_reference,
export_type, export_properties)
export_return :=
call_trader_export(export_reference, export_type, export_properties)
-

Fig. 1.13. Server Skeletons Generated Through Mapping

Here a process type representing the *Register* interface is generated. This process type has gates (*g_Register*) connecting it which enable the input signals to be delivered, i.e. the client invocations, and output signals, i.e. server terminations, to be sent. Local variables are also declared in the process type. The values of these variables are set by the client invocation.

The behaviour of the process type itself is such that in all states it accepts the export signal (*pCALL_CosTrading_Register_export*) and calls a virtual local procedure before returning to the same state. The behaviour of this local procedure is minimal and consists of a single output to the *Sender* of the invocation. This is illustrated in Figure 1.14.

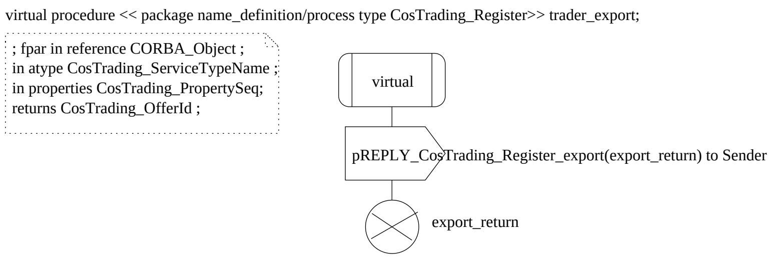

virtual procedure << package name_definition/process type CosTrading_Register>> trader_export;
; fpar in reference CORBA_Object ;
in atype CosTrading_ServiceTypeName ;
in properties CosTrading_PropertySeq;
returns CosTrading_OfferId ;
virtual
pREPLY_CosTrading_Register_export(export_return) to Sender
export_return

Fig. 1.14. Server Side Operations to be Implemented

By default this operation returns the successful return result, i.e. it returns the *pREPLY_CosTrading_Register_export* signal to the invoking exporter. The operations associated with the trader object are implemented by inheriting and redefining these procedures. It is also possible to modify the trader behaviour so that the operations are not available in all states.

Whilst it is possible to develop clients within the SDL specification itself through specifying the appropriate behaviour in the generated SDL stubs (not shown here), another approach is to treat the clients as being external to the specification itself as shown in Figure 1.15.

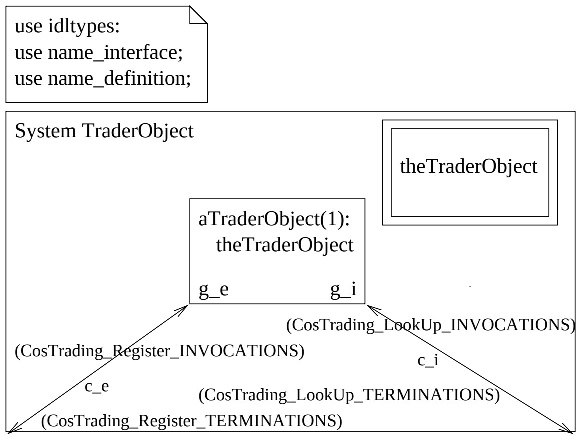

use idltypes:
use name_interface;
use name_definition;
System TraderObject
theTraderObject
aTraderObject(1):
theTraderObject
g_e
g_i
(CosTrading_LookUp_INVOCATIONS)
(CosTrading_Register_INVOCATIONS)
c_i
c_e
(CosTrading_LookUp_TERMINATIONS)
(CosTrading_Register_TERMINATIONS)

Fig. 1.15. Design of Trader Specification

Here, the trader server object interacts with the importers and exporters via channels connected to the environment. Through this approach we overcome one of the limitations of SDL for distributed systems development, namely that it does not allow for the dynamic communications to be set up between processes existing in different blocks, e.g. via the dynamic creation of channels. We note that it is possible to achieve this dynamicity through exported and imported remote procedure calls since the channels are not required to exist, however, as stated remote procedures do not currently support the raising of exceptions.

We note here that the existing package and those that are generated are inherited by the trader system. These will subsequently be used (inherited) in the specification of the trader interfaces.

Having an IDL mapping to SDL is a useful starting point to develop a specification, however, it does not necessarily provide enough information which will lead to a final successful design. Central to this issue is the disparity in the notion of objects having multiple interfaces. For example, it is quite possible to specify independent processes representing the behaviour of the different trader interfaces. The relation between these interfaces is up to the specification designer. Of course, given that the *Register* interface accepts *export* requests which can subsequently be imported by interacting with the *LookUp* interface, there must obviously be some form of interaction between the associated processes — either directly or indirectly. It might be the case that the *Register* process maintains some form of database of exported offers which can be queried by the *LookUp* process. Deciding upon such an approach will likely lend itself to a non-scalable design with different interfaces managing their own information which might be needed elsewhere, i.e. by other interfaces.

A further vagueness of an IDL based description is the lifecycle of the object and the interfaces it supports. It is often the case that some form of control over different interfaces is required, e.g. so that they can be created or deleted as part of the lifecycle of the object itself. One way in which these issues can be overcome is through a process modelling a central object core. This is responsible for the coordination between the separate processes implementing the interfaces to the object, as well as the lifecycle of the object as a whole. One such structure is shown in Figure 1.16.

In Figure 1.16 we note that at system start up time only a single core object process exists. This process is used to create instances of the interfaces, i.e. *Register* and *LookUp* associated with the trader object. We note that we could simply assume that a single core object and single instances of the interfaces exist, however, having creator processes, e.g. $theCore$ allows for the process identifiers to be obtained both for the creating process ($OffSpring$) and the created processes ($Parent$). Possession of process identifiers is the SDL equivalent of possession of an interface reference in the CORBA domain. This allows, amongst other things for communications to be checked between core objects and the supported interfaces, e.g. to ensure that the signals sent and received are from the expected source.

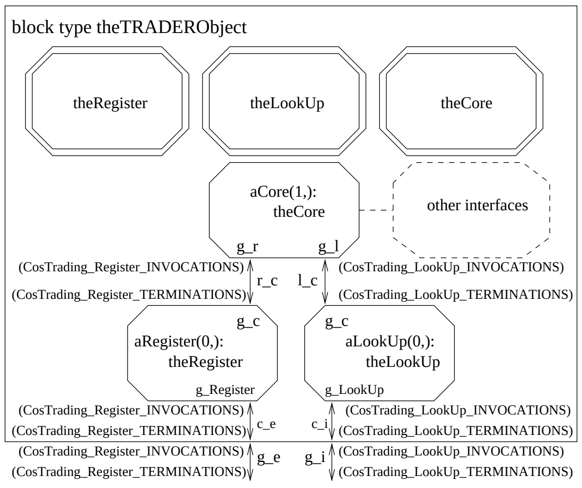

block type theTRADERObject
theRegister
theLookUp
theCore
aCore(1,): theCore
other interfaces
g_r
g_l
(CosTrading_Register_INVOCATIONS)
(CosTrading_Register_TERMINATIONS)
r_c
l_c
(CosTrading_LookUp_INVOCATIONS)
(CosTrading_LookUp_TERMINATIONS)
g_c
aRegister(0,): theRegister
g_Register
g_c
aLookUp(0,): theLookUp
g_LookUp
c_e
c_i
g_e
g_i

Fig. 1.16. Decomposition of Trader Object

As stated earlier the specification of the server behaviour is achieved by inheriting and redefining the appropriate virtual procedures. Given that a core object has been introduced, it is necessary to extend the process types representing the server interfaces by adding the necessary gates (g_c) to support the interactions with the core object as shown in Figure 1.17.

Several things should be noticed in Figure 1.17. Firstly, we note that for brevity we have also included the behaviour of the redefined procedure $trader\_export$. We also note that the process inherits the appropriate process in the $name\_definition$ package, i.e. $CosTrading\_Register$.

The redefined procedure itself checks the details of the client invocation. Specifically, it ensures that a non-null reference is attached which can subsequently be used by the importers to interact with the service, otherwise it raises the $pRAISE\_CosTrading\_Register\_InvalidObjectRef$ exception and returns a null export identifier ($nullId$). If a non-null reference is passed by the client, the redefined $trader\_export$ procedure then checks whether any of the properties that have been supplied have identical names. If so it raises the $pRAISE\_CosTrading\_DuplicatePropertyName$ exception and returns a null export identifier ($nullId$). If no exceptions are raised then the request is forwarded to the core object (itself represented by the $Parent$

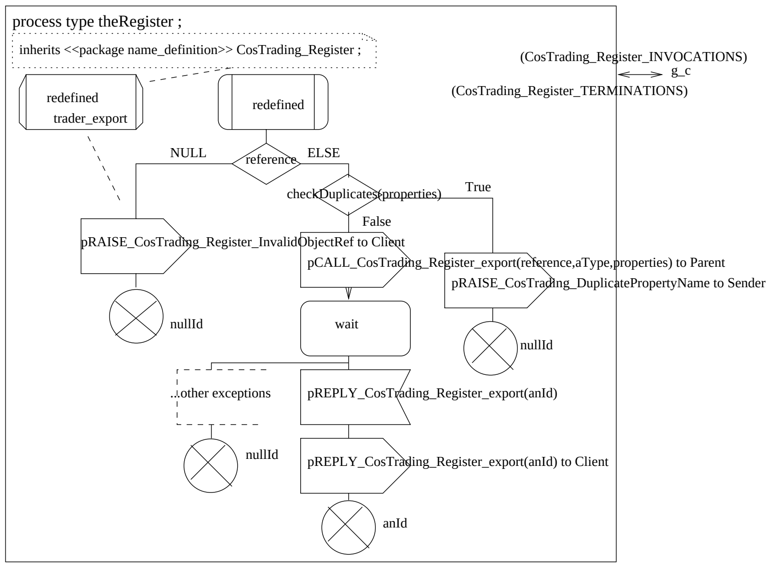

process type theRegister ;
inherits <<package name_definition>> CosTrading_Register ;
(CosTrading_Register_INVOCATIONS)
g_c
(CosTrading_Register_TERMINATIONS)
redefined trader_export
redefined
NULL
reference
ELSE
checkDuplicates(properties)
True
False
pRAISE_CosTrading_Register_InvalidObjectRef to Client
pCALL_CosTrading_Register_export(reference,aType,properties) to Parent
pRAISE_CosTrading_DuplicatePropertyName to Sender
nullId
wait
nullId
..other exceptions
pREPLY_CosTrading_Register_export(anId)
nullId
pREPLY_CosTrading_Register_export(anId) to Client
anId

Fig. 1.17. Implementing the Trader Register Interface

process identifier). The core object itself can raise other exceptions (not further specified here) which also result in a *nullId* export identifier being returned to the client. If the export operation was successful however, then the (*pREPLY_CosTrading_Register_export*) reply is received with an appropriate export identifier (*anId*).

We note that for brevity we omit the task box which is used to assign the *Client* as the *Sender* of the export invocation. Also we omit the details of the *checkDuplicates* operation which checks whether a sequence of properties has two or more properties with identical names.

A simplified example of the structure of the process type representing the core object itself is given in Figure 1.18.

Several things should be noted here. Firstly, to simplify the diagram we have omitted the initial starting behaviour of the core object. This would typically include the creation of the interfaces used by the trader object (*LookUp* and *Register*) and storage of the information associated with them, e.g. the process identifiers for the created offspring. In addition we assume that the variable declarations contains definitions for all of the local variables used.

The process type itself has the gates associated with it that are used for interacting with the *LookUp* and *Register* interfaces, i.e. *g_l* and *g_r* respectively which support the appropriate signal lists.

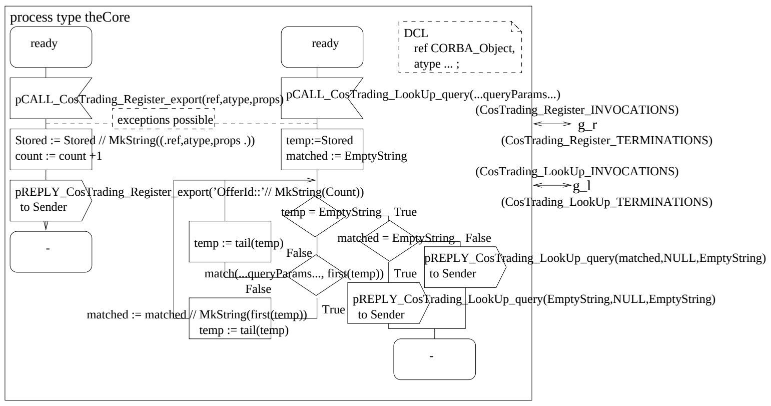

process type theCore
ready
ready
DCL
ref CORBA_Object,
atype ... ;
pCALL_CosTrading_Register_export(ref,atype,props)
pCALL_CosTrading_LookUp_query(...queryParams...)
(CosTrading_Register_INVOCATIONS)
g_r
(CosTrading_Register_TERMINATIONS)
exceptions possible
Stored := Stored // MkString((.ref,atype,props .))
count := count +1
temp:=Stored
matched := EmptyString
(CosTrading_LookUp_INVOCATIONS)
g_l
(CosTrading_LookUp_TERMINATIONS)
pREPLY_CosTrading_Register_export('OfferId::'// MkString(Count))
to Sender
temp = EmptyString
True
temp := tail(temp)
matched = EmptyString
False
-
False
pREPLY_CosTrading_LookUp_query(matched,NULL,EmptyString)
to Sender
match(...queryParams..., first(temp))
True
False
pREPLY_CosTrading_LookUp_query(EmptyString,NULL,EmptyString)
to Sender
matched := matched // MkString(first(temp))
temp := tail(temp)
True
-

Fig. 1.18. Implementing the Trader Core Object

Upon reception of the export signal from the *Register* process, the core object checks the details of the parameters and that the signal itself is from the correct source, i.e. from one of the create and known interfaces of the trader object. If the information associated with the request is not satisfactory, then the appropriate exception is raised, otherwise the core object stores the information associated with the offer. It might be the case that some process modelling a repository is used to store the service offers. Instead here we represent the stored offers as a local variable represented as a sequence of service offers (*Stored*). An export identifier is then returned to the *Register* interface which then forwards it onto the initial invoker, i.e. the invoker.

As with the export invocation, upon reception of a query signal (from the *LookUp* process), the core object checks the details of the invocation and can raise the appropriate exception. For simplicity the *queryParams* is used to represent the collection of input parameters associated with the *query* operation. If the details of the request are satisfactory, then the core object seraches its stored offers. It creates a local copy of the stored offers and traverses this searching for matching offers. If none are found, e.g. no service offers have yet been exported or no match can be found, then an empty sequence (*EmptyString*) is returned, otherwise the sequence of matching offers are returned. For simplicity we return a *Null* reference to the offer interator interface and restrict ourselves to the case where no policies have been applied. It might be the case that additional alternatives are included in the core object behaviour, e.g. to check if the number of matching offers has reached the maximum limit as defined by the *how_many* parameters of the *query* request. We also assume that the operation *match* is defined which can be used to judge whether the details of a query request are satisfied by an existing export offer.

1.2.3 Conclusions

This chapter has provided a brief introduction to the specification language SDL. We have shown its applicability to developing realistic specifications of distributed systems through applying it together with its associated tools to develop a trader specification. We also used current approaches to distributed systems more generally, i.e. through the generation of appropriate generation of client stubs and server skeletons whose behaviour is to be implemented.

Such an approach has many direct advantages and certain disadvantages. The main advantage from such an approach is that it allows for a direct relation between a specification and an implementation to be ascertained, e.g. tree and tabular combined notation (TTCN) [12] based tests derived from such specifications can be executed against the corresponding implementations [21]. Having an IDL basis for specification and implementation corresponds to having a common level of abstraction — at least at the syntactic interworking level. Through this, specifications can be used as a realistic part of software development. Whilst abstraction is a powerful tool for developing formal specifications, unfortunately it is often the case that too much abstraction results in models of systems that bear little or no relation to the software being developed. On the one hand this can be seen as a good thing, e.g. if requirements capturing is the aim of the specification, however, formal methods will only really be truly accepted if they are seen to help and improve the development of software iteslf. Starting from a common IDL basis thus represents a unique opportunity for formal methods.

The downside of such a direct relationship between specification and implementation are that, in development of realistic systems, the specifications themselves can become very large. As a consequence, the models become more difficult to deal with, e.g. to check for properties of the specification such as dropped signals via tool support. The problem of state space explosion is an ever present issue that has to be addressed. From this, many of the misnomers of the application of formal methods into the software development process need to be considered, e.g. that they can guarantee perfect systems. Rather, a more pragmatic approach should be adopted. This might include vigorously tested and validated components of some larger system, or for the generation of use cases represented as message sequence charts [15] for the system as a whole.

Bibliography

Ulf Behnke and Michael Geipl. Development of broadband ISDN telecommunication services using SDL'92, ASN.1 and automatic code generation. In Gregor von Bochmann, Rachida Dssouli, and Omar Rafiq, editors, Proc. Formal Description Techniques VIII, pages 237–252. Chapman-Hall, London, UK, 1996.

G. Booch, J. Rumbaugh, and I. Jacobsen. Unified Modelling Language Semantics and Notation Guide 1.0. Rational Software Corporation, San Jose, California, 1997.

Rolv Braek. SDL basics. Computer Networks and ISDN Systems, 28(12):1585–1602, 1996.

J.S. Dong and R. Duke. An object-oriented approach to the formal specification of ODP trader. In Proc. IFIP TC6/WG6.1 International Conference on Open Distributed Processing, pages 341–352, September 1993.

Jan Ellsberger, Dieter Hogrefe, and Ferenc Belina, editors. SDL – Formal Object-Oriented Language for Communicating Systems. Prentice-Hall, Englewood Cliffs, New Jersey, USA, 1997.

J. Fischer, A. Prinz, and A. Vogel. Different FDTs confronted with different ODP-Viewpoints of the Trader. In FME'93: Industrial Strength, Formal Methods, First International Symposium of Formal Methods Europe, pages 332–349. Lecture Notes in Computer Science, 1993.

GMD Fokus, Berlin, Germany. Y.SCE Manual, 1999. More information under http://www.fokus.gmd.de/research.

Object Management Group. Trading Object Service: v1.0, OMG Trading Function Module. Object Management Group, Inc., Framingham, MA, March 1997.

Humboldt University, Berlin, Germany. SDL Integrated Tool Environment, 1999.

ISO/IEC. Information Processing Systems – Open Systems Interconnection – Specification of Abstract Syntax Notation One (ASN.1). ISO/IEC 8824. International Organization for Standardization, Geneva, Switzerland, 1990.

ISO/IEC. Information Processing Systems – Open Systems Interconnection – Specification of Basic Encoding Rules for Abstract Syntax Notation One (ASN.1). ISO/IEC 8825. International Organization for Standardization, Geneva, Switzerland, 1990.

ISO/IEC. Information Processing Systems – Open Systems Interconnection – Conformance Testing Methodology and Framework – Part 3: The Tree and Tabular Combined Notation (TTCN). ISO/IEC 9646-3. International Organization for Standardization, Geneva, Switzerland, 1991.

ITU. Specification and Description Language. ITU-T Z.100. International Telecommunications Union, Geneva, Switzerland, 1996.

ITU. Specification and Description Language. ITU-T Z.100. International Telecommunications Union, Geneva, Switzerland, 2000.

ITU-T. Message Sequence Chart (MSC). ITU-T Z.120. International Telecommunications Union, Geneva, Switzerland, 1996.

ITU-T. Specification of Signalling System No. 7. Recommendation Q.701-795. ITU-T, Geneva, Switzerland, 1996.

F. Lucidi, A. Tosti, and S. Trigila. Object oriented modelling of advanced IN services with SDL-92. In Zmago Brezocnik and Tatjana Kapus, editors, Applied Formal Methods in System Design, pages 17–26, Maribor, Slovenia, June 1996. Action COST 247.

Anders Olsen, Ove Færgemand, B. Mœller-Pedersen, Rick Reed, and John R. W. Smith, editors. Systems Engineering using SDL-92. North-Holland, Amsterdam, Netherlands, 1994.

Rolv Bræk and Øystein Haugen, editor. Engineering Real Time Systems – An Object-Oriented Methodology using SDL. Prentice-Hall, Englewood Cliffs, New Jersey, USA, 1993.

R.O. Sinnott. An Architecture Based Approach to Specifying Distributed Systems in Lotos and Z. PhD thesis, Department of Computing Science and Mathematics, University of Stirling, UK, May 1997.

R.O. Sinnott and M. Kolberg. Creating Telecommunication Services based on Object-Oriented Frameworks and SDL. In M. Raynal, T. Kikuno, and R. Soley, editors, Proceedings of Second IEEE International Symposium on Object-Oriented Real-Time Distributed Computing, St. Malo, France, May 1999.

R.O. Sinnott and M. Kolberg. Engineering Telecommunication Services with SDL. In P. Ciancarini, A. Fantechi, and R. Gorrieri, editors, Proceedings of Formal Methods for Open Object-Based Distributed Systems, Florence, Italy, February 1999.

R.O. Sinnott and K. J. Turner. Applying the Architectural Semantics of ODP to Develop a Trader Specification. Computer Networks and ISDN Systems: Special Edition on Specification Architecture, March 1997.

Telelogic AB, Malmö, Sweden. Telelogic Manual, 1999.

Verilog, Grenoble, France. ObjectGeode Manual, 1999.