ENGINEERING TELECOMMUNICATION SERVICES WITH SDL

Dr Richard Sinnott

GMD Fokus
Kaiserin-Augusta-Allee 31
Berlin, Germany
sinnott@fokus.gmd.de

Mario Kolberg

Dept. Electronic and Electrical Engineering
University of Strathclyde
Glasgow, Scotland
mkolberg@comms.eee.strath.ac.uk

Abstract: If formal techniques are to be more widely accepted then they should evolve as current software engineering approaches evolve. Current techniques in the development of distributed systems use interface definition languages (IDLs) as a basis for the underlying communication and also as an abstraction tool. Object-oriented technologies [6] and the idea of engineering software through frameworks [5] are also widely accepted approaches in developing software. In this paper we show how the formal specification language SDL and associated tool support have been applied in the TOSCA[1] project to engineer telecommunication services using these current techniques.

INTRODUCTION

Open distributed systems, i.e. (extendable) systems that interoperate to achieve some overall goal, represent a prime example of the kind of area where formal techniques could have a significant role to play. Whilst current technologies such

[1] This work was undertaken as part of the Advanced Communication Technologies and Services (ACTS) TINA Open Service Creation Architecture (TOSCA) Project. The project is funded under ACTS proposal AC237.

P. Ciancarini et al. (eds.), *Formal Methods for Open Object-Based Distributed Systems*

as CORBA [3] have addressed many of the issues involved in developing distributed systems, e.g. remoteness of components and their potential heterogeneity, such technologies fall short of being the final solution to building truly open distributed systems. CORBA allows system interconnectivity to be achieved, i.e. sub-systems will understand the messages that are sent to them, but this does not mean that they will interoperate correctly, i.e. work together to achieve some predefined goal. What they lack is behaviour.

Formal techniques offer a means whereby behavioural descriptions can be given both precisely and concisely. Unfortunately, most developers of open distributed systems rarely if ever apply formal techniques in the development of software. Why? Some of the more common reasons are:

- they are based on mathematical notations that are difficult to understand;
- they produce models of systems that often bear no relation to the software itself [2];
- the models of systems produced are not usually re-used, whereas software – especially that based upon current practices such as object-oriented technology – is generally expected to be;
- they lack tool support for both developing and reasoning about the specifications.

In this paper we attempt to show through example, how SDL and associated tools as used in the TOSCA project address these issues. The rest of the paper is structured as follows. Section 2 provides an outline of the TOSCA project and the approach adopted to service creation together with an outline of the TINA architecture and the TINA object definition language (ODL). Section 3 provides an outline of the ODL/IDL mapping rules to SDL used in TOSCA. Section 4 presents the tool chain used in TOSCA. Section 5 provides an example of the development of a framework using this tool chain. Section 6 shows how the framework developed can be specialised to produce a model of a service. Finally, section 7 offers some conclusions and identifies areas of future work.

THE TOSCA APPROACH TO SERVICE CREATION

The TOSCA project proposes an approach to service creation which should provide both for rapid service provisioning and for high service quality. The approach assumes that for certain categories of service, a flexible and reliable software ***framework*** is developed. The concept of framework based software engineering has arisen to help to realise the holy grail of software engineering: *re-use*. Frameworks are a natural extension of object-oriented techniques. Whilst object technology provides a basis for re-use of code, it does not provide features to capture the design experience as such. Frameworks have developed to fulfil this need.

[2] Note this is often desireable if a high-level (abstract) model of the system, e.g. a business model, is made.

A framework can be regarded as a collection of pieces of software or specification fragments that have been developed to produce software of a certain type or niche [5]. A framework is only partially complete. Typically, they are developed so that they have holes or flexibility points in them where service specific information is to be inserted. This filling in (*specialisation*) of the flexibility points is used to develop a multitude of services with differing characteristics.

In TOSCA, this specialisation may be done by non-technical people, e.g. business consultants, through paradigm tools. Paradigm tools offer a graphical and intuitive means whereby services can be designed. Thus the service designer should not necessarily have to consider the lower level behaviour of the service to be able to create one. Rather, they should be provided with a high-level representation of the service components and the ability to *tune* their behaviour and how they are composed with one another. We shall see how this tuning is achieved in section 6.

Once the design of the service is complete, in the first instance, it is necessary to provide some immediate feedback to ensure that the service behaviour is as desired. This is achieved through (graphically) animating the service behaviour. Once the basic functionality of the service is satisfactory to the service designer, a more detailed check on its behaviour is required, i.e. it has to be validated. We do not address validation in this paper. Instead we focus on the development of frameworks and how they can be subsequently specialised.

Frameworks based on the TINA Architecture

The development of frameworks within TOSCA is based around the TINA architecture, or more specifically the Service Architecture [11] and Network Resource Architecture [12] of TINA. The Service Architecture introduces the underlying concepts and provides information on how telecommunication applications and the components they are built from, have to behave. Central to the Service Architecture is the concept of a *session*. This is defined as:

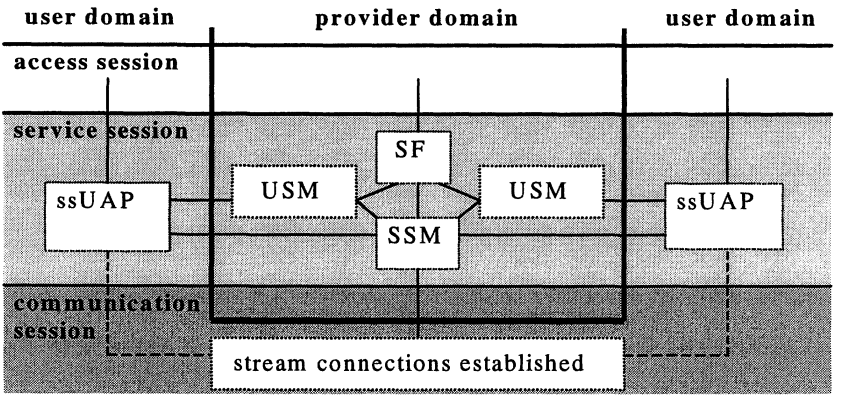
> *the temporary relationship between a group of resources that are assigned to fulfil collectively a task or objective for a time period.*

Three sessions are identified:

> **access session**: this represents mechanisms to support access to services (service sessions) that have been subscribed to.
>
> **service session**: includes the functionality to execute and control and manage sessions, i.e. it allows control of the communication session.
>
> **communication session**: controls communication and network resources required to establish end to end connections.

Currently, the service session has been the main area upon which frameworks are being developed in TOSCA. The relation between the three sessions is depicted in Figure 1.

Figure 1 Relation Between the TINA Sessions.



Here the service session user application (ssUAP) represents the users interface to the service, i.e. it determines how they may participate in the service. The Service Factory (SF) is used to create instances of services when requested to do so by components in the access session: namely user agents. Broadly speaking, an instance of a service typically consists a Service Session Manager (SSM) to control the global service behaviour, and a collection of User Session Managers (USM) – one of each is used to control each users participation (and state) in that service.

Typically, users can join services, suspend, resume or terminate their participation in services. The logic associated with these requests are processed in the service session, e.g. whether the user is able to resume themselves in the service at that time. If successful, the appropriate connections operations are made on the communication session, e.g. resume my previously suspended connections.

It is important to note that this architecture does not overly constrain the kinds of services that can be created from it. Rather, it acts as a template for a multitude of services, e.g. multimedia conferencing services, chatline services or newsflash services to name but three. Indeed even within these three services there exist a plethora of variations. In multimedia conferencing for example, there might be differing roles, e.g. chairman, observer, participant. These differing roles might result in differing expected functionalities, e.g. only chairman can invite (or suspend or terminate) other users, only participants can vote. Users might be able to have differing charging (or billing or accounting) possibilities, e.g. reverse or split charging, or other variations.

As well as these role specific specialisations, numerous others are possible also, e.g. only start the service if a certain number of successful responses to the invite have been received. Quit the service if the number of users falls below a certain level (or if the total charges generated from using the service falls below a certain level). It is precisely these variations on the general theme that paradigm tools are expected to capture whilst the general theme itself is represented by the framework.

To engineer frameworks it is thus necessary to have a core behaviour. In TOSCA this core behaviour is based around the informal (textual) description of the behaviour of the service session components, along with the TINA ODL and IDL for those objects.

TINA ODL

The TINA ODL language [13] arose in part, out of the different interpretations of objects that exist and the need to distinguish between client and server interfaces. In the Open Distributed Processing (ODP) reference architecture [15], objects have multiple interfaces. In the CORBA world, however, ODP interfaces correspond to CORBA objects. TINA ODL is a superset of IDL that allows for IDL interfaces to be grouped into **object** structures.

Object structures in ODL distinguish between client and server interfaces. This is achieved through labelling interfaces as either **required** or **supported**. Required interfaces are those interfaces an object needs to be supplied from its environment to function correctly. Supported interfaces are those interfaces an object offers to its environment. All objects should support an **initial** interface. This is a **supported** interface that is returned when the object is instantiated. As well as the operational, i.e. RPC-like interfaces, ODL also provides stream interfaces used for sending and receiving information flows. Currently the modelling of stream interfaces has not been addressed in our work.

The ODL language also provides a **group** structure that allows objects (**components**) to be collected together. The interfaces associated with these objects, or a subset of them can be declared as contracts with the environment of the group. They thus represent the externally visible interfaces to the group. All groups have one object that acts as a **manager**.

FROM CORBA IDL/TINA ODL TO SDL

Given the importance of ODL and IDL in the syntactic specification of objects in TINA, and the importance of IDL in the specification of interfaces (as well as for communicating with ORBs) generally in distributed systems development, mappings to a formal language are essential if formal methods are to be advocated to software developers. SDL is one of the few formal languages for which an ODL and IDL mapping has been made [Born97]. Along with this mapping, there are

numerous other advantages in using SDL to develop frameworks. The language itself has many features that make it suitable for framework development and subsequent specialisation. These are discussed in more detail in [9].

The following table summarises some of the main features of the ODL/IDL to SDL mapping used (and implemented) in TOSCA.

Table 1 Summary of the ODL/IDL to SDL Mapping.

ODL Structure	SDL Mapping
Group type	Block type
Object type	Block type
Interface type	Process type
Object Reference	Pid
Oneway (asynchronous) Operation	Signal prefixed with pCALL_
Operation (synchronous)	Signal pair. The first signal is prefixed with pCALL_, the second signal prefixed with pREPLY_ or pRAISE_ (if exception raised)
Exception	Signal prefixed with pRAISE_
Basic IDL types, e.g. long, char, float,...	Syntype
Any	not supported
Enum	Newtype with corresponding literals
Typedef	Syntype
Struct	Newtype with corresponding structure
Constant	Synonym

Other mappings have been made from IDL to SDL [1], however these are based largely around the remote procedure call concept of SDL. The remote procedure concept in SDL is a shorthand notation and is based on a substitution model using signals and states. More precisely, remote procedures are decomposed into two signals. The first carries the outgoing parameters (**in** or **inout**) and the second the return value of the procedure and all **inout** parameters. These signals are sent via implicit channels and signalroutes. There are several problems with mapping IDL operations to remote procedures. For example, they prohibit the raising of exceptions – an essential feature in realistic distributed systems. Also, the client side of the remote procedure call is blocked until the server side returns.

One point worth noting here regarding the mapping is the modelling of object types through blocks in SDL. In SDL, it is not possible to create blocks dynamically, although this is one extension to the language that may well be incorporated in the next version of SDL (SDL-2000). Only processes can be created dynamically. As we shall see however, it is possible to model the effect of the dynamic creation of objects (blocks) in SDL through the structuring and definition of the associated processes.

As with other IDL language mappings, client stubs and server skeletons are generated. These act as templates whose behaviour is to be filled in through inheritance. These stubs and skeletons are placed in two SDL packages (*Name_Interface* and *Name_Definition*). The Name_Interface package contains the interface specifications in the form of data types, signals, remote procedures, signallists etc. The Name_Definition package contains the structure information that is inherent in the IDL description in the form of modules, objects and interfaces, as well as a system type representing the IDL description.
As an example of the kind of SDL, we focus on the user session managers (USM). In particular those that might arise in a multimedia conferencing application. A USM is modelled as a group that consists of both a service specific part (user framework specific – UFS) and a generic part (not given here) which are themselves groups. We note that the UFS group may be specialised whilst the generic part may not. We shall consider the structure of the UFS group in more detail in section 5. For brevity, we consider only the framework specific part of the USM. The simplified ODL for the USM is:

```
group USM
{ components UFS, ...;
  manager UFSmgr;
  contracts i_UFSmgr, i_Callback, i_ControlWindowHandler ...;
};
```

We shall discuss the interfaces used to interact with the user application (i_Callback and i_ControlWindowHandler) in more detail in section 6 when we consider the specialisation of the USM. The UFSmgr object is responsible for controlling the objects in the specialisable part of the USM. In reality this means that it should – amongst other things - be able to terminate, suspend or resume existing all objects it controls, or add new (named) objects[3] to those it currently knows about. This implies that all objects controlled by the UFSmgr support this basic *lifecycle* functionality, i.e. upon reception of certain signals all objects can suspend, resume or terminate themselves. To achieve this, all objects inherit from interface i_CO_lifecycle. The IDL for this interface is:

```
interface i_CO_lifecycle
{ void initialiseObject(in PropertyList initInfo, in Object mgrRef);
  void suspendObject();
  void resumeObject(in Object mgrRef);
  void terminateObject();
};
```

When initialised or resumed, objects need to be made aware of the reference for their managers. This allows for later checks on arriving invocations, i.e. to check that they originated from their manager. As well as supporting this core

[3] Typically the object name (a string) and a null reference are passed in. The object name and the PId for the created object (or null if it could not be created) is then returned.

functionality, the interface to the UFSmgr (i_UFSmgr) supports other operations. The default behaviour for the UFSmgr is that it allows a user to suspend and terminate their participation in the current session. The UFSmgr may also be specialised. We consider here the operation to specialise the start up of user sessions. This operation and how it may be specialised are considered in more detail in section 6. The IDL for the i_UFSmgr interface is:

```
interface i_UFSmgr : i_CO_lifecycle
{ void suspendSessionRequest(); //called by user to suspend their session
  void terminateSessionRequest();//called by user to terminate their session
  void suspendAll(); ... //called by SF to suspend USM and all associated
objects
  void requestObject(inout NamedObject obj);   //called to create window
handlers
  oneway void ufsstart(); ... //used by paradigm tool for specialisation of
USM start
};
```

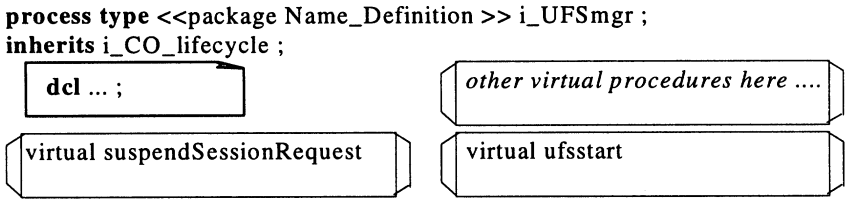
When mapped into SDL the packages that are generated contain block types for the USM which itself contains block types for the UFS group and the generic part (not shown here). The UFS block type contains block types for the UFSmgr (and other objects as we shall see in section 5). Figure 2 gives a snapshot of the information in package Name_Interface focusing on the i_UFSmgr signals.

Figure 2 Example of the Contents of the Name_Interface Package.

```
Package Name_Interface
signal pCALL_i_UFSmgr_suspendSessionRequest;
signal pCALL_i_UFSmgr_terminateSessionRequest;
signal pCALL_i_UFSmgr_ufsstart;
signal pCALL_i_UFSmgr_suspendAll;
signal pCALL_i_UFSmgr_requestObject(NamedObject);
// and associated pREPLY_ signals - but not for ufsstart (oneway)
signallist i_UFSmgr_INVOCATIONS =
  pCALL_i_UFSmgr_suspendSessionRequest,
  pCALL_i_UFSmgr_terminateSessionRequest, pCALL_i_UFSmgr_ufsstart,
  pCALL_i_UFSmgr_suspendAll, pCALL_i_UFSmgr_requestObject..;
signallist i_UFSmgr_TERMINATIONS =
 pREPLY_i_UFSmgr_suspendSessionRequest,
 pREPLY_i_UFSmgr_terminateSessionRequest,
 pREPLY_i_UFSmgr_suspendAll, pREPLY_i_UFSmgr_requestObject... ;
```

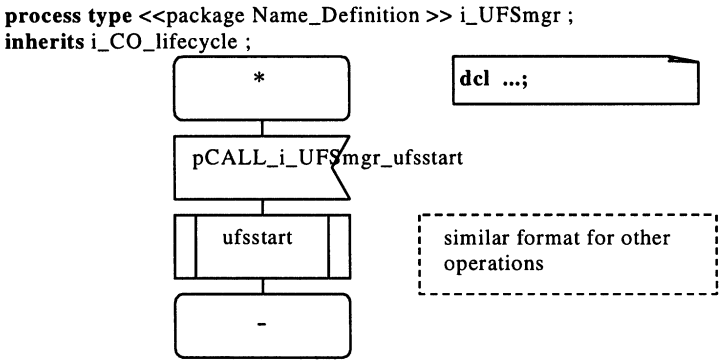
This package is then *used* in the definition of the Name_Definition package. Figure 3 gives an example of the kind of SDL generated focusing on the i_UFSmgr interface of the UFSmgr object:

Figure 3 Example of the Contents of the Name_Definition Package.



The virtual procedure for the ufsstart (and all oneway operations) consist of a virtual start transition followed by an immediate exit. In non-oneway operations, the generated procedures contain a pREPLY signal of the appropriate kind. Along with the virtual procedure definitions, signals and states are also generated that result in the procedures being called. An example of the format of the signals is given in figure 4.

Figure 4 Example of Signals and Procedure Calls Generated.



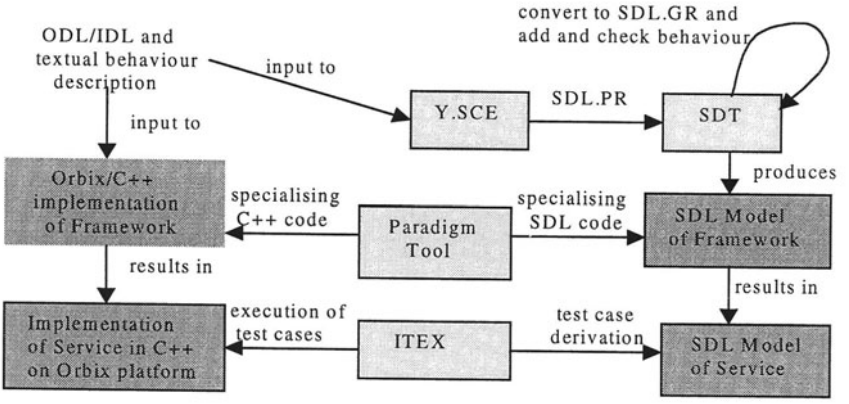
As we shall see in section 5, this default behaviour of accepting all signals in all states can be modified (restricted) through inheritance and redefinition.

Having a mapping from ODL/IDL to SDL is only a basis from which specifications can be developed. Tool support is essential if frameworks and their specialisation are to be developed. SDL has arguably the most developed toolsets of all formal techniques used today.

TOOL SUPPORT FOR SPECIFICATIONS IN TOSCA

TOSCA has developed a tool chain that allows for both the development of specification frameworks from ODL and IDL descriptions through to their specialisation and the subsequent verification of the created service. Figure 5 highlights the current tool chain used in TOSCA.

Figure 5 The Tool Chain in TOSCA.



Here the Y.SCE tool [16] allows (amongst other things) ODL and IDL descriptions to be developed (or imported) and subsequently mapped to SDL in PR format. These SDL fragments are then themselves imported into the Telelogic TAU toolset [10]. This toolset consists of a collection of tools that allow SDL specifications to be both specified, checked, e.g. simulated and validated (using SDT) and subsequently tested (using ITEX). More information on the TOSCA tool chain with particular emphasis on deriving tests to run against the (CORBA based) implementations of the service can be found in [7]. The resulting SDL model of the framework is represented as a package in SDL. This package is then used by associated paradigm tools to develop complete models of services.

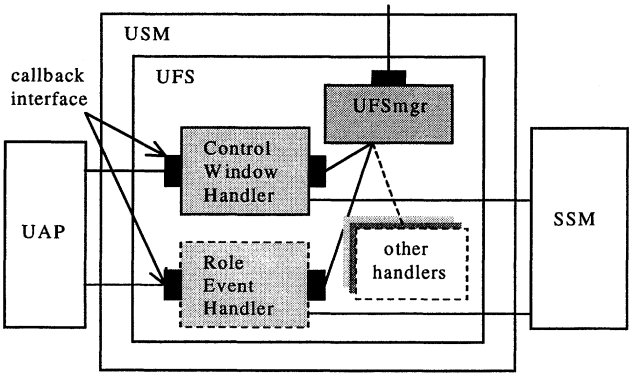
As stated previously, paradigm tools are used by potentially non-technical users to develop services. TOSCA has implemented two paradigm tools that can be used to produce intuitive (graphical) models of the services. We consider one in particular here based on the functional block paradigm. This paradigm provides service designers with a list of *basic events* at which the behaviour of the service can be defined. These are the key points at which the designer can intervene and customise how the service will behave. The basic events thus correspond to the framework flexibility points. Numerous basic events have been identified, e.g. starting/stopping the service, starting/stopping user sessions, etc. We focus on the start up of user sessions in section 6, or more specifically, the specialisation of the IDL operation *ufsstart* given earlier.

Once the service design is complete, the paradigm tool outputs both the specialising C++ and SDL. The SDL is then imported into the SDL toolset SDT and used to develop an SDL system from the model of the framework. The completed specification of the service is then used as a basis for reasoning about and subsequently verifying the C++ implementation of that service.

EXAMPLE OF FRAMEWORK DEVELOPMENT

As an example of framework development, we consider a multimedia conference service based on the TINA service session components. As discussed, the USM interacts with the user application (and vice versa) to allow users to participate in service sessions. Since it is not known *a priori* what the functionality of the user might be, e.g. if they are chairman of the conference then they may well have different functionality[4] than if they were merely observers in a conference, it is necessary to define the interface to the components of the USM in such a way that they can be extended. Thus, it should be possible to dynamically extend the user application with new buttons (and associated callbacks) if the specialisation so desires. The default functionality of the USMs are that they allow a user to terminate and suspend their participation in a given service, i.e. the default user application is such that they have a control window with two buttons: teminateMe and suspendMe. This default behaviour for dealing with these termination and suspension requests in the USM is included in a Control Window Handler. Additional functionality is achieved through dynamically adding new objects and associated callbacks to the USM (via the paradigm tool). A Role Event Handler object is used as a placeholder for inserting this new behaviour. The structure of the USM considering only the specialisable parts is shown in figure 6.

Figure 6 Simplified Structure of the USM.

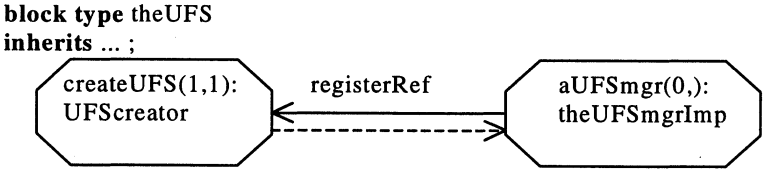


We note here that the dotted lines indicate that the objects do not initially exist, they are created only when specialisation takes place. As stated, USMs are created by the service factory. This occurs when either a request to start a service occurs or when users wish to join into an already existing service. Since it is not (currently) possible to dynamically create blocks in SDL, an alternative solution is to have process instances existing in those blocks that exist at system start-up. As an example, at system start-up, the UFS block contains a creator process used solely to create instances of the UFSmgr which in turn creates all other processes it

[4] and hence buttons on the window of their user application.

needs, e.g. the necessary handler objects. Once the UFSmgr has created all necessary objects, it returns the reference to the procedure of the **initial** interface to that object. Figure 7 shows the simplified structure of this for the UFS object, i.e. showing only the process instances.

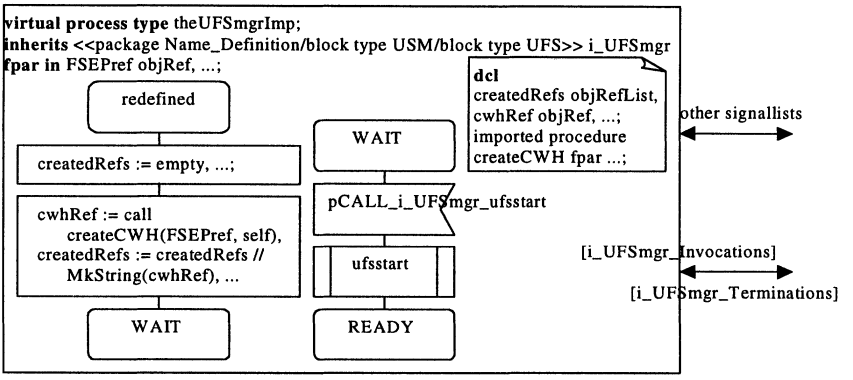
Figure 7 Overcoming the Lack of Block Creation in SDL.



Typically, these creator processes (UFScreator) support a single exported procedure which has to be imported into those blocks wishing to create instances of the exporting block. For example, the service factory will import the exported remote procedure for the UFSmgr (createUFS) and the UFSmgr will import the exported remote procedures for the different window handlers etc. Through this approach, the perception of dynamic block instantiation can be achieved.

As an example of the way in which the generated SDL server skeletons can have their core behaviour inserted, i.e. the behaviour before they are specialised, we consider the i_UFSmgr interface of the UFS object given previously. As stated, the default behaviour for the UFSmgr is that it creates a control window handler only. A simplified example of the structure of this object is given in figure 8.

Figure 8 Structure of Basic UFSmgr.



This process type is parameterised with (amongst other things) the reference to the user application[5] (FSEPref). When an instance of this process type is created, initialisation of local variables is done, e.g. the list of created references is set to

[5] This was passed in when the initial call to the service factory was made.

empty, and the default behaviour of creating a control window handler is made. As discussed, this requires that the necessary exported remote procedure is imported. Following this default behaviour, the UFSmgr is ready to be specialised, i.e. it is in a state where it can accept signal pCALL_i_UFSmgr_ufsstart.

As stated, the specialisable procedures have null behaviours, i.e. start and exit. This allows for the behaviour of the framework as a whole to be checked without necessarily having any specialisation taking place, e.g. the basic USM behaviour (and SSM and SF) behaviours can be checked to ensure the framework as a whole correctly represents the informal (textual) requirements. Once the core behaviour has been specified and verified, the framework can be saved as a package and *used* in defining services, i.e. SDL systems.

EXAMPLE OF FRAMEWORK SPECIALISATION

Both simple inheritance and virtual inheritance are used to specialise the components in the framework. Simple inheritance was used at the upper block level, e.g. the USM block level. Subsequent block types, e.g. the UFS block type as well as process types and procedures were reused by virtual inheritance. Hence the UFSmgr process type given above was declared as virtual. This was necessary since virtual inheritance allows the commuication links, i.e. channels and signalroutes in the framework to be reused (and possibly extended). Virtual inheritance does not, however, allow for multiple redefinitions in one scope (e.g. different types of USM at system level, for chairman or observer roles etc). As a result, it was not possible to use virtual inheritance for the top-level block types. simple inheritance was used instead.

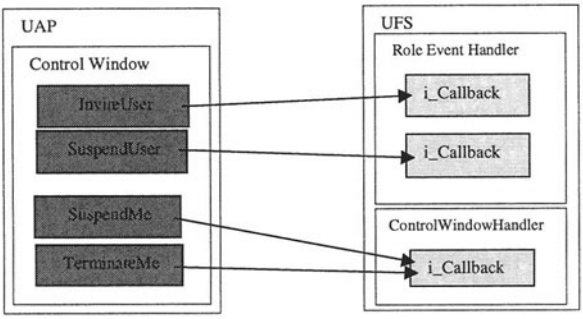
As an example specialisation of the framework we specialised the USM (UFSmgrs) so that three roles were given: chairman of the session, and participant and observer in the session. Each role has a set of privileges or characteristics attached to it which are defined by the paradigm tool. Each member in a session is assigned one of these roles and hence may only perform the corresponding activities. We consider the specialisation of the framework whereby the chairman role extends the basic USM functionality by allowing for invitations to be sent out and for suspending other users in the session.

The Control Window on the user application supports an interface that enables widgets (buttons) to be added dynamically (i_DynamicWidgets) to the user interface. In addition, the handler associated with this window (Control Window Handler) supports an i_Callback interface which is used for receiving events from the user application.

If a button is pressed on the user application, a specific signal is sent to the corresponding i_Callback process instance. For the buttons common to all Control Windows, the signal is sent to the i_Callback process instance within the Control Window Handler block. The i_Callback process instances in the Role Event

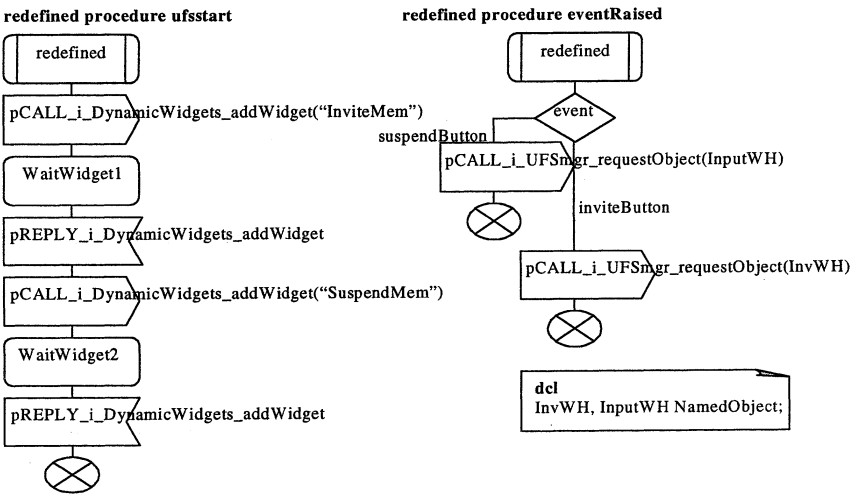
Handler block are used for buttons specific to a particular member role. Figure 9 illustrates the relation between buttons and i_Callback process instances.

Figure 9 Buttons on the Control Window and associated i_Callback process type instances.



Role dependent buttons are dynamically added to the window at system start-up. To do this the addWidget procedure in the i_DynamicWidgets process of the handler is called from the ufsstart procedure. In the addWidget procedure a new instance of an i_Callback process type in a Role Event Handler is requested. Upon reception of that process reference it calls the addWidget operation in the Control Window with this reference as a parameter. To perform this specialisation, procedure ufsstart (in the UFSmgr) and procedure eventRaised (in the i_Callback process of the handler object) need to be specialised. This is shown in figure 10.

Figure 10 Specialised procedures ufsstart and eventRaised.



Here the ufsstart procedure calls the addWidget procedure twice to create two buttons on the Control Window, namely one to invite a member and the second to suspend a member from the session. If one of these buttons are pushed then the event is sent to the i_Callback process and handled in its eventRaised procedure. Thus if the Invite Button is pressed an appropriate handler object and associated window are created. This window can then be used for invitations (which are processed by the created handler object). Similarly, if the suspend user button is pressed, an appropriate handler object and associated window are created. The user is then prompted for the name of the member to be suspended.

There are numerous other specialisations of the framework possible. These are not only based around specialising different flavours of USMs but also include specialising the SSM in order to implement specific policies for the service as described in section 2. Most of these require only minimal work to specialise the framework, e.g. a single procedure requires specialisation.

Once the framework has been specialised by the paradigm tool, in the first instance, it is animated to give the service creator feedback on its functionality. As well as the user interface creation being animated, e.g. new buttons being created on their application, we have focused - amongst other things - on producing graphical animations of the interface to the communication session, e.g. showing the connections between users in the session and how they are modified when new users join, or existing users suspend or resume their participation in sessions. It is important to note that the objects performing the animation, i.e. the GUIs, are themselves CORBA objects. Currently C++ wrappers are used to tie the C code generated by the SDL tools (namely the SDT Simulator) to the CORBA world, i.e. the GUIs. Another area of TOSCA work is investigating how the code generation of the SDL tools can reflect the SDL system more closely. Thus rather than generate a large C file for the whole system, collections of files are generated that reflect the SDL system structure more closely, e.g. files for the blocks (SDL models of CORBA objects). If successful, this would then allow the generation of CORBA object implementations directly from their SDL models.

CONCLUSIONS

SDL is a formal specification language with an easily understandable graphical syntax and considerable commercial tool support. As the language is reviewed and updated every four years, it has incorporated many features that make development and subsequent usage of frameworks straightforward. In particular, its support for object-oriented concepts such as inheritance and its support for re-use through the package construct.

Given the importance of IDL in the development of distributed systems, and ODL in the development of telecommunication systems, mappings from ODL/IDL to SDL have been given [1,2]. The mappings used [2] and implemented in TOSCA

[16] differ (and are improved) from others [1] in that, amongst other things, exceptions are supported.

We have shown through an example how it is possible to develop realistic (multimedia) telecommunication services in SDL using a framework and paradigm based approach. Starting from an ODL/IDL description of the syntactic aspects of the components in the framework, tool support was used to generate SDL stubs and skeletons which were subsequently enriched with behaviour specifications. We have also highlighted how the framework itself could be specialised to particular services through paradigm tools.

As discussed, the development of frameworks is, by and large, a non-trivial activity. As well as capturing a design that can be re-used, frameworks have to have well defined points in which their behaviour can be modified or extended. Thus it is quite possible to destroy the integrity of a framework through erroneous specialisations. To address this issue, we have considered a small set of predefined flexibility points (operations) associated with the framework components. We showed how one of these (ufsstart) could be used to produce different flavours of USM.

Currently, our work has focused mainly on the simulation and graphical animation of the created services. The next phase of our work will focus more on their validation. This validation activity will consider both the validation of isolated services, and the implementation of services interaction management techniques to support the interworking of services in an environment where other services exist that might adversely influence one another. Some of the issues associated with service interaction in a TINA world are discussed in more detail in [4].

More information about the current status of the work in TOSCA can be found at: http://www.teltec.dcu.ie/tosca/

Acknowledgements

The authors are indebted to the partners in the TOSCA project. The TOSCA consortium consists of Teltec DCU, Silicon & Software Systems Ltd, British Telecommunications, University of Strathclyde, Centro Studi e Laboratori di Telecommunicazioni SpA, Telelogic, Lund Institute of Technology, GMD and Ericsson.

References

[1] M. Björkander, Mapping IDL to SDL, Telelogic AB, 1997.

[2] M. Born, A. Hoffmann, M. Winkler, J. Fischer, N. Fischbeck, *Towards a Behavioural Description of ODL*, Proceedings of TINA 97 Conference, Chile.

[3] *The Common Object Request Broker Architecture and Specification*: Revision 2.0, Object Management Group, Inc., Framingham MA., July 1995.

[4] M. Kolberg and E. Magill: *Service and Feature Interactions in TINA,* submitted to Feature Interaction Workshop'98, Lund, Sweden 1998.

[5] R. Johnson and V. Russo, *Reusing Object-Oriented Designs,*Urbana, Ill., May 1991.

[6] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, W. Lorensen, *Object-Oriented Modelling and Design*, Prentice Hall 1991.

[7] I. Schieferdecker, M. Li, A. Hoffmann, *Conformance Testing of TINA Service Components - the TTCN/CORBA Gateway*, Proceedings of the Intelligence in Networks & Services Conference 1998, Antwerp, May 1998.

[8] International Consultative Committee on Telegraphy and Telephony - *SDL - Specification and Description Language*, CCITT Z.100, International Telecommunications Union, Geneva, Switzerland, 1992.

[9] R. Sinnott, *Frameworks: The Future of Formal Software Development*, to appear in Semantics of Specifications, Journal of Computer Standards and Interfaces, July 1998.

[10] Telelogic AB, *Getting Started Part 1 - Tutorials on SDT Tools*, Telelogic AB, 1997.

[11] TINA-C, *Service Architecture*, version 5.0, 16 June 1997.

[12] TINA-C, *Network Resource Architecture*, Version 3.0, February 1997.

[13] TINA-C, TINA Object Definition Language MANUAL, version 2.3, July 1996.

[14] TOSCA Consortium Deliverable 6, *Initial Approaches to the Specification and Validation of TINA Services, Internal Deliverable* AC237/GMD/WP3/DS/R/009/a1.

[15] *Basic Reference Model of ODP -Part 2: Foundations*, ISO/IEC International Standard 10746-2, ITU-T Recommendation X.902, Geneva, Switzerland 1997.

[16] For more information see http://www.fokus.gmd.de/minos/y.sce.