Engineering of Interworking TINA-based Telecommunication Services

Mario Kolberg[1], Richard O. Sinnott[2] and Evan H. Magill[1]

[1]Dept. of Electronic and Electrical Engineering
University of Strathclyde
Glasgow, Scotland
{m.kolberg, e.magill}@eee.strath.ac.uk

[2]GMD-FOKUS
Kaiserin-Augusta-Allee 31
Berlin, Germany
sinnott@fokus.gmd.de

Abstract- **This paper describes a Service Creation approach being developed in the EU funded ACTS TOSCA (TINA Open Service Creation Architecture) project to rapidly develop validated TINA based multimedia telecommunications services. The approach is based around object-oriented software frameworks in SDL which are specialized towards services by means of graphical paradigm tools.**

Further, in TOSCA, the need for service interworking across service provider domains via federation has been recognized in order to allow users to join service sessions offered by providers they are not customers of. However, service interworking may cause undesired behavior - the so called *service interaction* phenomenon. This paper focuses on this issue and the underlying technology of the service creation approach with emphasis on how service federation has been implemented.

I. Introduction

To date the work on the Service Architecture within TINA-C has concentrated mainly on the engineering of *individual* services (a standalone offering such as multimedia videoconference or chatline). In this paper we argue that this is insufficient, and that interworking *between* services must also be addressed. We know from experience in established architectures that the interworking of services can cause undesired behaviors [1,15]; a phenomena known as *service interaction*. Indeed initial studies have shown that service interactions are also a potential problem between interworking TINA-based services [4].

Even though the service interaction problem provides the motivation for our work, in this paper we will not be discussing feature interaction as such, but describing our work on service creation and service interworking. More precisely, our report will concentrate on the service federation mechanism [11]. This work is being carried out in the EU funded ACTS TOSCA project.

II. The TOSCA Approach to Service Creation

The aim of the TOSCA project is to develop a service creation environment that enables multimedia-based telecommunication services to be produced in an effective manner, that is they are created rapidly but not at the expense of their reliability [8]. Indeed, the created services are rigorously validated. Validation of services implies that formality is introduced into the service creation process. In TOSCA we used the formal language SDL [5] to model our services. Few formal techniques have adopted current techniques in software development as strongly as the ITU-T Specification and Description Language. As well as providing a state of the art tool support [10], SDL offers ODL [13] and CORBA IDL [2] mappings which are an essential feature given that the syntactic aspects of TINA-based systems are expressed in ODL/IDL. An overview of the advantages and disadvantages of applying SDL to service creation is given in [9].

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 services, a flexible and reliable software *framework* is developed. The concept of framework based software engineering has arisen to help realize 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.

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 [3]. 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 (*specialization*) of the flexibility points is used to develop a multitude of services with differing characteristics.

In TOSCA, this specialization 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 behavior of the service to be able to create one. Nor should they necessarily be experts in implementation languages such as C++, distributed architectures such as CORBA, or specification languages such as SDL. Rather, they should be provided with a high-level representation of the service components and the ability to *tune* their behavior and how they are composed with one another. We show how this tuning is achieved in section V.

0-7803-5785-X/$10.00 © 1998 IEEE

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 [7]. 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 with relation to aspects of *service interworking*.

III. The TINA Architecture and Service Federation

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 Fig. 1.

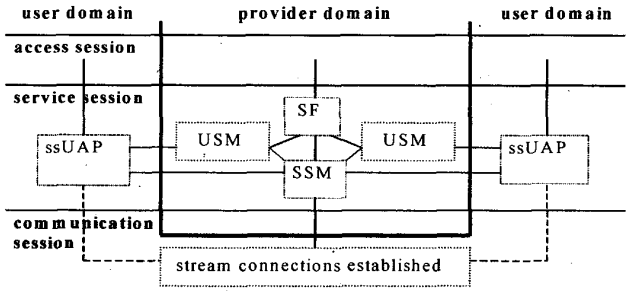

Fig. 1. Relation between the TINA Sessions.

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 behavior, and a of User Session Managers (USM); one of each is used to control each users participation in that service.

A. Service Federation

Implicitly, the scenario above assumes that only users subscribed to the same retailer can participate in particular service sessions. This is a serious drawback in a real-life competitive business environment where potentially many service providers might exist. In this case, customers of different service providers would not be able to communicate with one another through those services. Hence some mechanism is needed to allow users associated with different service providers to join 'shared' service sessions. The federation concept was developed [11] to address this issue. Federation allows users to join services offered by remote providers, i.e. providers they are not customers of.

A federation can be divided into two stages: firstly the domain federation and secondly the service federation. The domain federation establishes an environment for two or more service providers to transparently offer services across their domains by setting up an access session. The service federation in turn establishes a service session across two or more retailer domains. In fact, two separate service sessions are set up (one at each location) which interwork. As the access session is the foundation for a service session, the domain federation is the base for and governs the service federation. Since TOSCA is concentrating on the service session, the work on federation has thus far been focused on service federation.

In order to allow services to interwork, a new component is introduced in the TINA Service Architecture [11]: the PeerUSM (Peer User Service Session Manager). The interworking of federated services is achieved through PeerUSMs. In practice, information which needs to be passed to a session member associated with a remote domain is sent to the local PeerUSM which forwards it to the remote PeerUSM which in turn passes it on to its SSM or the relevant USM. Fig. 2 illustrates this scenario.

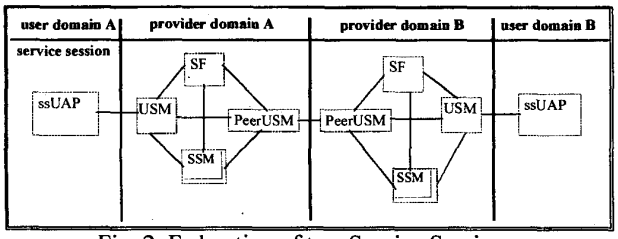

Fig. 2. Federation of two Service Sessions.

While working in a federated service session, each component in the other domain (SSM or USM or SF) is only accessible through the PeerUSM. There is no direct communication between components belonging to different domains other than the PeerUSMs. The PeerUSM itself appears as a 'normal' USM to other local components, i.e. local USMs and the SSM.

Using the initial specification of the federation concept in the TINA Service Architecture we implemented the service federation concept and hence defined the PeerUSM in more detail. In the following we discuss this work.

Since the PeerUSM may be an access point to a number of users in the other domain, the PeerUSM needs to be able to forward incoming messages, e.g. operation invocations, terminations or exceptions, to their correct destination. From this it follows that the PeerUSM needs to be able to distinguish messages by their final destination. To achieve this the PeerUSM offers interfaces for each user in the other domain. These interfaces are created dynamically by the PeerUSM manager when users join a federated session. Figure 3 gives an overview of the internal structure of the PeerUSM.

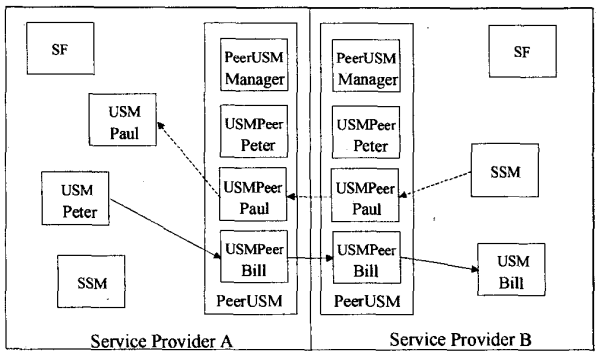

Fig. 3. Overview of PeerUSM.

Here the solid line indicates the sending of a message from Peter's USM in the service provider domain A to Bill's USM in the service provider domain B. The dashed line shows the SSM in service provider domain B accessing Paul's USM in service provider domain A. In other words, messages sent from a local USM or SSM to a remote USM or SSM are sent via the local PeerUSM to the remote PeerUSM and finally onto their final destination, i.e. the remote USM or SSM.

Beside supporting some special methods called during initialization and termination, PeerUSMs offer the same external interfaces as other USMs. This is necessary to support distribution transparencies, e.g. masking the remoteness of location between services. Thus components should be unaware that they are talking to a PeerUSM and not to a 'normal' USM. Another point worth noting here is that sending messages from one PeerUSM to its counterpart in the other domain, rather than directly to the destination component in the remote domain, potentially allows for implementing specific PeerUSM-PeerUSM communication protocols. One such protocol could be the TINA Retailer-to-Retailer Reference Point (RtR) when this becomes available. In other words, a local PeerUSM could map the messages it receives into messages understood by a remote PeerUSM.

TINA states that the services in a federation should be of compatible types[1]. However, as we shall see in section VI, the service interaction problem still exists in this case. As stated, services in TOSCA are (rapidly) created and subsequently validated through specializing object-oriented frameworks using graphical paradigm tools. Hence, the first steps taken in TOSCA in investigating the service interaction phenomenon as it might arise in a TINA environment, has been through the creation of object-oriented service frameworks. In the following, after giving some necessary background information on the structure of the framework, we focus on the design of the PeerUSM in section IV.B.

IV. Example of Develpoing an Object-Oriented Framework

If developing software is a complex activity then developing frameworks is more complex again. Developing a framework so it removes large parts of the problem of service design, thus expediting the creation process, whilst still offering a means to create numerous different kinds of services is an especially challenging activity. Producing successful frameworks requires that the points where design decisions can be made, are flexibility points. Using frameworks to produce services then requires that these flexibility points are made available so that new design choices can be taken to produce new services or service flavors. Perhaps the hardest part of the framework development process is the identification of these flexibility points [3].

To engineer frameworks it is thus necessary to have a core behavior. In TOSCA this core behavior has been based around the informal description of the behavior of the service session components, i.e. the ssUAP, USM, SSM and SF; their ODL (Object Definition Language) description [13] and parts of an existing TINA-based multimedia-conferencing implementation [14].

[1] In reality this compatibility relationship is given in terms of the PeerUSMs supporting and requiring the same feature sets. Feature sets themselves are given through syntactic structures in ODL and IDL.

The USM and SSM components in the framework are decomposed into generic and specific parts with the generic parts being fixed and the specific parts being incomplete in the framework and thus specializable by the paradigm tool. Each of the service parts has a manager (UFSmgr, USPmgr for the USM and GFSmgr, GSPmgr for the SSM). These managers are responsible for lifecycle and initial access to the managed objects, e.g. the managers are able to initialize, suspend, resume or terminate the objects they manage, or provide references to the objects they manage on request. When the manager is told to suspend, resume or terminate, it also suspends, resumes or terminates the objects it manages respectively. As we shall see in section IV.A, these manager operations and the initialization of the manager together with the objects it is to create (and subsequently manage) correspond to framework flexibility points.

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 connection operations are made on the communication session, e.g. resume my previously suspended connections.

It is important to note that the architecture of the framework 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 specializations, 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.

A. Design of the Framework USM in TOSCA

In TOSCA we focused on a small set of flexibility points. This set of flexibility points allowed us to produce a multitude of different services with different types of behavior. Specifically, we chose the following flexibility points:

- the start up, suspension, resumption and termination of users sessions;
- the start up, suspension, resumption and termination of service sessions.

In producing a framework it is necessary to have fixed places where the flexibility points are to exist. Flexibility points cannot simply be placed anywhere in the design of the framework. Rather, flexibility points may only be filled in (specialized) at certain fixed times. Thus it is necessary to represent the points of flexibility directly in the design of the framework, but the actual behavior associated with these flexibility points is effectively NULL until they are specialized. To achieve this we introduced appropriate IDL operations that were associated with the appropriate objects in the framework design. As an example we consider the USM and the representation of the flexibility points concerned with the start up, suspension, resumption and termination of users sessions. The simplified ODL for the USM modeled in the TOSCA framework is:

```
group USM {
  components UFS, ...;
  manager UFSmgr;
  contracts i_UFSmgr, i_DynamicWidgets, i_ControlWinHandler, ...;
};
```

The UFSmgr object is responsible for controlling the objects in the specializable 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[2] 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 initialized 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 functionality, the interface to the UFSmgr (i_UFSmgr) supports other operations. The default behavior for the UFSmgr is that it allows a user to suspend and terminate their participation in the current session. The IDL for the i_UFSmgr interface is:

[2] Typically, the object name and a null reference are passed in and the name and PId of the created object returned.

```
interface i_UFSmgr : i_CO_lifecycle {
  void suspendSessionRequest();        // called to suspend a user
  void terminateSessionRequest();      // called to terminate a user
  void suspendAll();  // used to suspend USM and associated objects
  void requestObject(inout NamedObject obj);
                    // called to create window handlers
  oneway void ufsstart();
                    // not implemented in framework – specialized!
  oneway void ufssuspend();            //        „
  oneway void ufsresume();             //        „
  oneway void ufsstop();               //        „
};
```

We point out here that we define several operations whose sole purpose is to act as a placeholder in the framework design through which the specialization can be achieved, i.e. these represent the points of flexibility that enable us to have different service behaviors. The other behaviors may be implemented directly.

B. Design of the PeerUSM in TOSCA

The PeerUSM is similar in terms of its interfaces to a USM. As already stated, it also contains a manager (PeerUSMmgr) to support startup, termination, suspension and resumption as well as getting initial access to the other parts of the PeerUSM. Further it supports an object which emulates a normal USM (USMPeer). Central to this object is an interface (i_USMPeer) which supports operations to suspend or terminate a user, as well as for creation and deletion of additional objects, e.g. objects used to handle user (ssUAP) inputs. In short, the PeerUSM encompasses *all* operations which can be called on a 'normal' USM, i.e. the interfaces supported by the PeerUSM should allow for *potential* inter-domain interactions to be as expressive as single domain interactions. A simplified ODL description of the PeerUSM and an IDL description of the interface i_USMPeer can be found below.

```
group PeerUSM {
      components USMPeer, PeerUSMmgr...;
      manager PeerMgr;
      contracts i_PeerMgr, i_USMPeer, i_CWHPeer...;
};

interface i_USMPeer : i_CO_lifecycle {
      void suspendSessionRequest();   // called to suspend a user
      void terminateSessionRequest(); // called to terminate a user
      void requestObject(inout NamedObject obj);
                        // called to create additional objects ...
};
```

Although the PeerUSM supports roughly the same interface as a USM, its internal behavior is different. Instead of performing the actual actions triggered by the arriving invocations, terminations or exceptions, a PeerUSM forwards the received messages to its counterpart in the remote domain which then forwards the message onto the appropriate object. In order to be able to do so each i_USMPeer in the same domain as its assigned USM knows the reference to the manager in the USM. This reference is provided on creation of the i_USMPeer interface. Similarly, the i_USMPeer interface in the other domain is initialized with the reference of its counterpart.

Further, in the TOSCA framework, the SSM in a service session knows about all users and also holds a reference to their USMs. Consequently, in a federated service session the SSM knows about all users, both local and remote. On request it can provide a USM with the reference to a USM of another user. In order to make the federation fully transparent for individual USMs, the SSM returns the reference to the i_USMPeer interface of the requested user if the associated USM is located in the remote domain. Thus USMs can access other USMs in the usual way, regardless of whether they actually belong to different domains. When a user intends to joins an already federated service session, the request is directed to the domain which issued the corresponding invitation. After checking the invitation and subsequently joining the user into the session (including the necessary components in the PeerUSMs) the SSM of the remote session is notified of the new member.

As the i_USMPeer interface is the contact point to a USM in the remote domain, it has to reflect any changes in the status of the USM in the session, e.g. the USM gets suspended or resumed. Thus, whenever a user (and hence the corresponding USM) suspends his participation in the session (or gets suspended by another member) the corresponding interfaces in the PeerUSMs are suspended also. This allows for denying any requests which might be sent to a suspended user from the remote domain. Once the user resumes the participation, the interfaces in the PeerUSMs are resumed also. Similarly, when a user is terminated from the session, the interfaces in the PeerUSMs related to him are terminated as well. Once all members of a local domain in a federated session are terminated, the whole local session is terminated as is the pair of PeerUSMs. In other words is the federation is terminated. However, remaining members in the session in the other domain are not affected and continue their participation in the session.

C. Specifying the Behavior of the Framework Components

The first stage in developing a formal specification from an ODL description is to map the ODL descriptions to the formal language of choice. We note here that relatively few ODL to formal specification language mappings have been made. SDL is one of the few languages that support a mapping. In TOSCA the Y.SCE tool [16] was used to import the ODL description and generate the associated SDL stubs and skeletons. A discussion of the mapping rules and its advantages to other mappings as well as its shortcomings can be found in [8].

The following table only gives a very short summary of the main features of the ODL to SDL mapping used (and implemented) in TOSCA.

TABLE I
SUMMERY OF THE ODL TO SDL MAPPING

ODL Structure	SDL Mapping
Group type	Block type
Object type	Block type
Interface type	Process type
Object Reference	Pid
Oneway Operation	Signal prefixed with pCALL_
Operation (synchronous)	Signal pair. The first signal 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, short, char, float,..	Syntype
Any	Not supported
Enum	Newtype with corresponding literals
Typedef	Syntype
Struct	Newtype with corresponding structure
Constant	Synonym

Through the mapping client stubs and server skeletons are generated. These act as templates whose behavior 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. An example of the contents of this package is given in Fig. 4.

```
Package Name_Interface
signal pCALL_i_UFSmgr_suspendSessionRequest;
signal pCALL_i_UFSmgr_terminateSessionRequest;
signal pCALL_i_UFSmgr_ufsstart;   // and ufsstop, ufsresume, ufssuspend
signal pCALL_i_UFSmgr_suspendAll;
signal pCALL_i_UFSmgr_requestObject(NamedObject);
// and associated pREPLY_signals - but not for ufsstart, ufsstop etc (oneway)
signallist i_UFSmgr_INVOCATIONS =
 pCALL_i_UFSmgr_suspendSessionRequest,
 pCALL_i_UFSmgr_terminateSessionRequest,
 pCALL_i_UFSmgr_ufsstart, // and ufsstop, ufsresume, ufssuspend
 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... ;
```

Fig. 4. Example of the Name_Interface Package.

This package is then *used*[3] in the definition of the Name_Definition package. Fig. 5 gives an example of the kind of SDL generated focusing on the i_UFSmgr interface of the UFSmgr object.

[3] That is the SDL 'use' mechanism is applied.

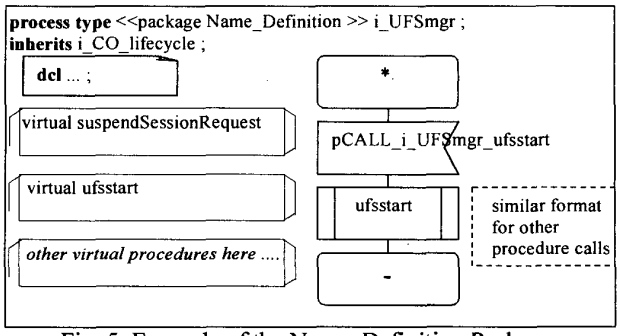

Fig. 5. Example of the Name_Definition Package.

The virtual procedure for the ufsstart (and all oneway operations) consist of a virtual start transition followed by an immediate exit. In non-oneway IDL operations, the generated procedures contain a *pREPLY_* signal of the appropriate kind. Along with the virtual procedure definitions, signals and (asterisk) states are also generated that result in the procedures being called.

As an example of the way in which the generated SDL server skeletons can have their core behavior inserted, i.e. the behavior before they are specialized, we consider the implementation of the i_UFSmgr interface (*i_UFSmgrImp*) of the UFS object given previously. The default behavior for the UFSmgr is that it creates a control window handler only. A simplified example of the structure of this object is given in Fig. 6.

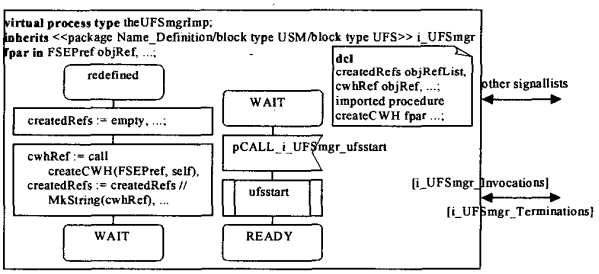

Fig. 6. Structure of Basic UFSmgr.

This process type is parameterized with (amongst other things) the reference to the user application. When an instance of this process type is created, initialization of local variables is done, e.g. the list of created references is set to empty, and the default behavior of creating a control window handler is made. As discussed, this requires that the necessary exported remote procedure is imported. Following this default behavior, the UFSmgr is ready to be specialized, i.e. it is in a state where it can accept signal *pCALL_i_UFSmgr_ufsstart*.

As stated, the specializable procedures have null behaviors, i.e. start and exit. This allows for the behavior of the framework as a whole to be checked without necessarily having any specialization taking place, e.g. the basic USM behavior (and SSM and SF) behaviors can be checked to

ensure the framework as a whole correctly represents the informal (textual) requirements. Once the core behavior has been specified and verified, the framework can be saved as a package and *used* in defining services, i.e. SDL systems.

V. Specialising the Framework towards Services

As an example of framework specialization we focus on a simple video conference service. This service is characterized by having two different kinds of user: *participants* and *guests*. There may be up to 5 participants and up to 10 guests using the service, i.e. in the service session, at a given time. Audio-video connections exist between all parties in the session.

In TOSCA, a "kind" of user is represented by a user role which has a set of privileges and characteristics attached to it. Each member in a session is assigned a role on joining the session and hence may only perform the activities associated with that role. In the case of the video conference service, the invoking member, i.e. the user who starts up the service is automatically assigned the role participant. The distinction between a participant and a guest is that a participant may invite other participants or guests. A participant may also suspend any guest in the session. Further, when the last participant terminates their user session the whole service session terminates.

As discussed previously, the framework already caters for session members being able to suspend and resume their participation in the session. However, in this particular video conference service we wish to have the additional functionality of all participants being able to invite new members to the session as well as being able to suspend guests from the session.

To achieve this service specialization we *use* the SDL package representing the framework. Both simple and virtual inheritance are used to specialize the components in the framework. Simple inheritance is 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 are reused by virtual inheritance. This is necessary since virtual inheritance allows the communication links, i.e. channels and signalroutes in the framework to be reused (and possibly extended). Virtual inheritance does not, however, it allows for multiple redefinitions in one scope. This is needed for having different types of USMs at system level for being able to specify multiple roles of users in a session, e.g. participants and guests in our example. As a result, it is not possible to use virtual inheritance for the top-level block types: simple inheritance is used instead.

Creating the role participant and guest requires that the USM block type in the framework is specialized to two new block types: partUSM and guestUSM.

To realize the cardinality constraints explained above, the SSM needs to be specialized as well. The SSM knows about all members and their current state in the session and is responsible for allowing new users to join the session. The SSM for this video conference service is thus specialized to restrict the maximum cardinality of participants to 5 and of guests to 10 and the minimum cardinality of participants and guests to 1 and 0 respectively.

As stated previously, the operation ufsstart in a UFSmgr is executed whenever a new instance of the USM is created, i.e. a user has joined the session. For the guest role, this procedure needs to be specialized so that audio-visual connections to all users are set up when the guest joins. For this, the SSM is firstly queried to find the users active in the session, i.e. get the information on all participants and guests. This is then followed by a call to the SSM to make the appropriate multimedia connections on the communication session between the newly joined guest member and all previously joined users. This is illustrated in Fig. 7.

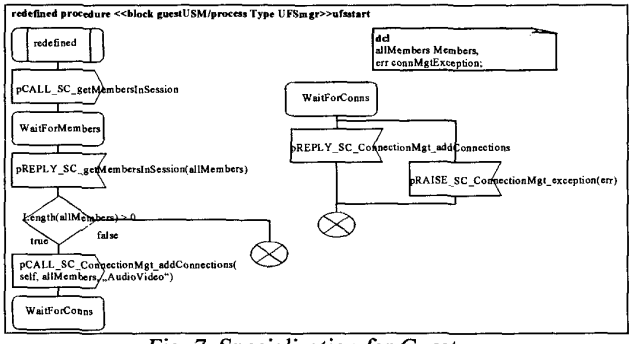

Fig. 7. Specialization for Guests.

The specialized *ufsstart* procedure in the partUSM block creates an invitation window handler to allow participants to invite other users as well as adding new buttons on the participant control window, e.g. a button to suspend another user. Adding buttons dynamically is supported by the control window and its associated handler by offering special interfaces to add widgets to the window and also to receive events from the window, i.e. button presses. For brevity we do not focus on the mechanisms in the framework to support the adding of widgets to the ssUAP. A detailed discussion on the possibilities of specializing the user interface according to role specific privileges in the TOSCA framework can be found in [7,8].

It is important to note here that the actual calling of the *ufsstart*, *ufsstop*, *ufsresume* and *ufssuspend* procedures always occurs at fixed points in the overall behavior of the framework. That is, *ufsstop*, *ufsuspend*, *ufsresume* are only called when a valid terminate, suspend or resume request respectively has been received by the UFSmgr, i.e. when it is in state ready/suspended, ready or suspended respectively.

Because of the way the PeerUSM is modeled in TOSCA, it does not require specialization for this example since it already supports all messages which can be sent to a USM by an external source (SSM, etc.). Moreover, the internal behavior of the PeerUSM does not require changing since it only forwards signals to its counterpart, i.e. the remote

PeerUSM, which then forwards the signals onto the referenced object. In other words, specialization of USMs does not necessarily require changes to the PeerUSMs. Specialization or rather extension of the PeerUSM is only necessary if the specialization of the framework involves adding new objects to the USM which can be accessed from outside the USM. For example, a conference chairman might request that a new voting window is created on the ssUAPs of the session members. This would then require that an associated voting handler object is created in the USM for the session member. In this case, the PeerUSM needs to be extended to intercept and forward the additional signals between the SSM and USMs. Using virtual inheritance of the PeerUSM is one way this extension can be achieved.

Once the framework has been specialized, in the first instance, it is animated to give the service creator feedback on its functionality. As well as animating the user interface creation, e.g. the new windows and buttons that are being added to the ssUAP, we are producing graphical animations of the interface to the communication session. That is, animations of 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. Currently, the framework supports Java based GUIs. Work is on-going to support CORBA based GUIs. A detailed discussion on the animation activity can be found in [7,8].

VI. Issues Related to Federation

The framework and a number of services developed from it were used to conduct an initial case study on federation of these services. Even though the services created so far from the TOSCA framework are not extremely rich in terms of their functionality, a number of issues related to the interworking of these services have been identified.

For example, even the very basic service created in the previous section can cause problems if two instances of it are to interwork. Imagine a scenario where a number of participants and guests are joined in a non-federated session of this service. If a participant invites a guest to the session who is not customer of the service provider used (provider A) a federation between the current service session and a newly created service session at the guest's local provider (provider B) is set up. However, since the invoking member of the service automatically becomes a participant, the invitee is not joined in as a guest as specified by the invitation but as a participant because he is the invoking member of the local service session. Furthermore, if during the process of joining the user, the invitation is checked more rigorously, that is the role of the new member is compared with the role specified in the invitation, the user about to join may be denied access to the session altogether. Clearly, a straightforward solution for this problem could be to implement a policy by which the role of the invoking users overwrites the role provided in the invitation. However this kind of ad-hoc solutions are not applicable to a broad range of services as is shown in the next example.

A videophone service specified from the framework allows only two members in the session at any one time: a caller (who starts the service) and a callee (who is invited to join the service). The caller and callee are automatically assigned their roles on starting and joining the service respectively. In other words, a conversation between two members is only established if the invoking user becomes a caller and the second user, the invitee, a callee. Clearly, in a federation of two sessions of this videophone service the same problem as described in the videoconference service above exists. If the problem is solved in the same way as described above, that is by overwriting the role in the invitation, a second caller (!) is joined in the session. Obviously, this does not allow for setting up a videophone conversation. These two examples show that ad-hoc solutions might be working for a few specific cases, but fail to solve the service interaction problem generally.

The two examples also demonstrate that even if two service sessions of exactly the same service are federated there are potential service interaction problems. Many more problems are anticipated if a federation between two service sessions which differ slightly in their functionality is set up. For example, consider a slight difference in the maximum cardinality of a specific member role. A conflict arises if a service session of service X and service session of service Y are federated. For instance service X may allow up to 5 participants in a session and service Y up to 10. Imagine the federated session has currently 5 members. If an invitation has been sent by a user at provider X, the invitee needs to direct his request to join the session to service X. In this case, the validity of the invitation is checked locally, i.e. in service X, which results in a user maximum cardinality exception being raised. The outcome of this is that the user is denied access to the service session. On the other hand, if the invitation had originated at provider Y (which means that service Y processes the validity of the invitation reply) then no maximum cardinality exception would be raised and the user could join the service session. However, when service X is notified that a new user has joined this would violate the maximum cardinality constraint causing an unwanted service interaction.

A further example occurs in a federation between two sessions of conference services which differ in their behavior when the last participant leaves the session. One service may terminate when this occurs whilst the other may not. Many more such examples could be imagined.

VII. Conclusions

This paper has tried to give a flavor of the tools and techniques currently being applied in the TOSCA project to develop telecommunication services based around the TINA architecture. We have seen how it is possible to take a semi-formal description given in TINA ODL, CORBA IDL and

informal text to develop an object-oriented framework in SDL. We have also highlighted how this framework can be specialized to create instances of services. We produced an instance of a video conference service with two role types having different characteristics, i.e. behaviour.

We have also shown how the TINA concept of the PeerUSM component may be specified and used to support interworking, i.e. federation, of services. However, despite the fact that TINA is designed to avoid service interaction problems as experienced in established telecommunication networks, we have shown by example that some issues do remain. We note that it is quite possible to resolve service interactions through refining service models or designs. Some of the previous service interaction scenarios could be avoided through changing the service design; that is by ad-hoc solutions. However, these solutions are not applicable generally as has been shown by example of the videophone service. Clearly, it are general solutions which need to be found if the federation concept is to be applied in an open competitive market as targeted by TINA. Future work in TOSCA will concentrate on developing such an approach.

It should be stated that the definition of the TINA Retailer-to-Retailer Reference Point is crucial if service interactions are to be avoided. Basing this only on syntactic considerations, e.g. feature sets represented in ODL or IDL, is, as we have shown, insufficient. This is true even in the case of identical services interworking. We note however, that having a common syntactic basis is an essential starting point for successful service interworking, i.e. understanding the messages that are sent and received. The problem of service interaction can only be resolved when the behavioral aspects of the services are considered. The next phase of the TOSCA work is in applying the SDL framework together with the associated SDL tools to investigate these issues.

ACKNOWLEDGMENTS

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] E.J. Cameron, N.D. Griffeth, Y.-J. Lin, M.E. Nilson, W.K. Schnure and H. Velthuijsen, *A feature interaction benchmark for IN and beyond*, In W. Bouma and H. Velthuijsen (eds.) *Feature Interactions in Telecommunications Systems*, IOS Press, Amsterdam, 1994, pp. 1-23.

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

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

[4] M. Kolberg and E. Magill: *Service and Feature Interactions in TINA.*. In K. Kimbler and L.G. Bouma (eds.) *Feature Interactions in Telecommunications and Software Systems*. pp. 78-84, IOS Press, Amsterdam, 1998.

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

[6] R. Sinnott, *Frameworks: The Future of Formal Software Development*, Journal of Computer Standards and Interfaces, special edition on Semantics of Specifications, August 1998.

[7] R. Sinnott, M. Kolberg, *Business-Oriented Development of Telecommunication Services with SDL*, Proceedings of OOPSLA-98 Workshop on Precise Behavior Specifications of OO Systems and Business Specifications, Vancouver Canada, October 1998.

[8] R. Sinnott, M. Kolberg, *Engineering Telecommunication Services With SDL,* in Formal Methods for Open Object-Based Distributed Systems, P. Ciancarini, A. Fantechi and R. Gorrieri (Eds.), pp. 187-203, Kluwer Academic Publishers, 1999

[9] A. Olsen, D. Demany, E. Cardoso, F. Lodge, M. Kolberg, M. Björkander, R. Sinnott, *The Pros and Cons of Using SDL for Creation of Distributed Services*, Proceedings of Sixth International Conference on Intelligence in Services and Networks (IS&N), Barcelona, Spain, April 1999, in press.

[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, *Specification of the Service Session Framework Targeted at the Eri-TIMMAP Platform*, AC237/ETL/WP2-4/PI/I/036/A4.

[15] S. Tsang, E.H. Magill and B. Kelly, *The Feature Interaction Problem in Networked Multimedia Services - Present and Future*, BT Technology Journal, Vol. 15, No. 1, January 1997, pp. 235-246.

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