Frameworks: The Future of Formal Software Development?

Dr. Richard O. Sinnott,
GMD-Fokus,
Kaiserin-Augusta-Allee 31,
D-10589 Berlin,
Germany,
email sinnott@fokus.gmd.de

It could be argued that the primary issue to be dealt with in software engineering today is re-use of software. Current software development rarely, if ever, starts from nothing. Unfortunately, the same cannot be said for the development of specifications. To overcome this problem, various works have attempted to show how specifications can be built using architectural principles. We discuss one such approach in particular, the Architectural Semantics of Open Distributed Processing. We show the limitations of this work with regard to the *architecting* of specifications and propose a new approach, based on frameworks. To highlight the approach we use the work currently being done in the TOSCA project in its development of a service creation and validation environment for telecommunication services.

Keywords: Formality, Frameworks, Open Distributed Systems, SDL.

1.0 Introduction

The development of software for open distributed systems is a complex activity. There are a multitude of issues that have to be addressed to ensure that the subsystems of the system under development interwork correctly to achieve their goals. Remoteness of software, potential for partial failure, concurrency, language and system heterogeneity are just some of the many direct problems facing distributed systems developers.

Whilst current technologies such as CORBA [Corba] have addressed many of the issues in remoteness and heterogeneity of languages and systems, such technologies fall short of being the final solution to building truly open distributed systems. To justify this, it is worth noting exactly what is meant by an *open* system since numerous interpretations of this term exist in the context of distributed systems. Leopold et al. [Leopold] identify several definitions that can be considered as correctly interpreting the term *open* as found in distributed systems literature:

- an environment and its model are easily obtainable and well documented;
- the model and environment exist on many operating systems and work with many programming methodologies and languages;
- several manufacturers support and control the market;
- the environment and its model are developed after open debating.

We regard the openness of a distributed system as the extendability of that system. That is, how new resource-sharing systems can be added to a system without disrupting existing services. Typically this is achieved through making available the descriptions of specific entry points (interfaces) into the system. From these, it should be possible to add new (possibly heterogeneous) hardware and software to the system.

Recent areas of research [Ansa, OdpRm1] into open distributed systems are replacing the notion of *interconnectivity* of computers by *interworking* of enterprises. We note the distinction between these two concepts as they pertain here. Interconnectivity of computers can be regarded as the ability of computers to communicate successfully with one another. This might be realised, for example, through ensuring that they use similar communication protocols as might be found in the Open System Interconnection (OSI) reference model [OsiRm]. Interworking is wider in scope than message passing capabilities though. That is, whilst communication is essential for the successful interworking of separate enterprises, it is only a basis on which the interworking can be established. Interworking may thus be regarded informally as the integration of enterprises to achieve some commonly agreed goal.

The fundamental problem that must be overcome to develop truly open distributed software is one of *semantics*. Current technologies such as CORBA, are limited by the fact that their openness is defined largely in terms of syntactic aspects of systems, namely by interface definition languages (IDL). Whilst this allows the issues of implementation language and system heterogeneity to be overcome, the approach does not lend itself to building truly open systems according to the above definition. Rather, the approach lends itself to interconnectivity as opposed to interworking issues.

To overcome this problem it is essential that behavioural aspects of systems are dealt with.[1] Formal modelling languages offer a means whereby behaviour can be expressed both precisely and concisely. Unfortunately such techniques are not widely exploited throughout the software engineering community, or at least not as much as we would advocate. There are numerous arguments put forward to reject such techniques - although proposals for improved techniques for expressing behaviour and semantics as precisely or concisely are rare. One of the these that we try to address in this paper, is the actual difficulty in developing specifications.

[1] Also non-functional aspects of systems such as timeliness, cost or resource capacities are essential considerations for successful interworking [RosPhd] but they are outside the scope of this paper.

Various works have been proposed and investigated to overcome this difficulty. One of these was the Reference Model for Open Distributed Processing (ODP-RM) [OdpRm2,OdpRm3]. These documents describe semantics in a rigorous although informal manner using stylised natural language. Natural language is limited for specifying semantics though, especially when complexity increases. ODP recognised this and advocated the use of formal description techniques. The Architectural Semantics [OdpRm4,OdpRm4a] was the result of this endorsement. The initiators of the Architectural Semantics identified numerous potential advantages for the work which we briefly discuss in section 2.1. One of these was that it would allow a structured (architectural) approach to developing specifications of open distributed systems. Whilst this idea of architecture based specification is an appealing one [RosKjt1,RosKjt2,Kjt], the Architectural Semantics work fell short of completely fulfiling this goal. We discuss the issues related to this shortcoming in section 2.1 and then propose a similar but more prescriptive approach based on the idea of frameworks. In particular we focus on the development of frameworks based on the Telecommunications Information Networking Architecture (TINA) in the context of the TOSCA project.

The rest of the paper is structured as follows. Section 2 gives an overview of the architectural semantics and its advantages and, especially with regard to this paper, its limitations. Section 3 introduces the concept of a framework and argues that they offer a means whereby the intention of the architectural semantics in terms of specification development can be realised. Section 4 looks at the requirements placed upon formal languages if they are to be used to successfully model frameworks. Section 5 highlights how this approach is being adopted in the TOSCA project. Finally section 6 draws some preliminary conclusions on the viability of frameworks as a mechanism for producing software in a formal manner.

2.0 Introduction to the ODP-RM

The ODP-RM is an architecture developed for creating standards for distributed systems. Here the term architecture implies that it consists of a collection of concepts with associated structuring rules that can be used for modelling and reasoning about distributed systems and the standards used to describe them. The ODP-RM itself is divided into four parts:

Part 1: Overview and Guide to Use - As its title suggests, this document provides introductory material on the ODP-RM framework family of standards.

Part 2 : Foundations - This document contains the definition of concepts and gives the framework for descriptions of distributed systems. It also introduces the principles of conformance and the way they may be applied to ODP. In effect this document provides the basic vocabulary with which distributed systems may be reasoned about and developed, i.e. it is used as the basis for understanding the concepts contained within Part 3 of the ODP-RM.

Part 3: Architecture - This document contains the specification of the required characteristics that qualify distributed system as open, i.e. constraints

to which ODP systems must conform. The main features of this document include transparencies, functions, conformance issues and viewpoint languages. Functions and transparencies aid in overcoming (masking) aspects of distribution, e.g.the potential remoteness of components.

ODP uses the notion of a viewpoint as it recognises that it is not possible to capture effectively all aspects of design in a single description. Each viewpoint captures certain design facets of concern to a particular group involved in the design process. In doing so it is argued that the complexity involved in considering the system as a whole is reduced. ODP recognises five basic viewpoints, each with its own associated language. These are the enterprise, information, computational, engineering and technology viewpoints. Each of these viewpoint represent a different abstraction of same original system; however, there is likely to be common ground between them.

Part 4: Architectural Semantics -This document [OdpRm4,OdpRm4a] contains a formalisation of a subset of the ODP concepts and structuring rules. The formalisation is achieved through interpreting these concepts and structuring rules in terms of the constructs of formal languages. The formal languages applied so far have been Estelle [Estelle], LOTOS [Lotos], SDL'92 [Sdl92] and Z [Zstandard].

2.1 Introduction to the ODP Architectural Semantics

The architectural semantics is divided into two areas. One part focusing on some of the basic ODP-RM Foundation concepts [OdpRm4] and one part focusing on (certain of) the viewpoint languages of the ODP-RM Architecture [OdpRm4a]. The effort involved in developing an architectural semantics has many advantages. These are discussed in detail in [RosKjt1,RosKjt2,RosKjt3,Kjt]. Amongst these are:

- it provides clear and concise statements in a given formal language - a formalisation of concepts which then acts as a more precise definition of the given concepts. In doing so it requires a more in-depth consideration of the textual definition of each concept than might otherwise have been achieved. In doing so it helps improve the text of the architecture itself.
- it offers the basis for comparison of different formal languages when used to provide formal descriptions of the same standard, i.e. it helps in identifying which language is most suitable for a given problem domain.
- it offers a basis for dealing with conformance, consistency and compliance.
- it allows the limitations of the formal languages used to be identified and documented for use by language developers.
- it offers a more structured approach to specification development so that reuse can be achieved.

This latter bullet point is the most directly relevant with regard to this paper. The intention is that by specifying the lower level concepts and the rules about how

they may be structured with one another to form more complex structures, the development of specifications is made easier. The analogy here would be an electronic engineer who works at an architectural level. The engineer does not have to re-specify the most basic of components such as flip-flops and NAND gates, but rather may use these as building blocks to create more complex components. An approach using LOTOS to do exactly this may be found in [RosMsc,RosKjt4].

Unfortunately, it was not the case that the architectural semantics supported this notion fully. The problem was one of prescription. Whilst precise [2], the concepts found in the Foundations document [OdpRm2] are largely non-prescriptive. As a result the formalisation of the concept is largely non-prescriptive. A representative example is the concept of *communication*. This is defined in X.902 as:

> *The conveyance of information between two or more objects as a result of one or more interactions, possibly involving some intermediate objects.*

The formalisation of this concept in LOTOS is given in [OdpRm4] as:

> *The conveyance of information (via value passing) between two or more interacting objects. It is not possible to write directly, cause and effect relationships. It should also be pointed out that the synchronisation itself may be construed as communication.*

As can be seen, such formalisations whilst useful do not lend themselves to re-use directly. We note here that this lack of prescription is only natural since the ODP-RM is not an architecture for any particular system. It is an architecture for a multitude of systems and hence does not prescribe explicit behaviours (or communications).

The Architecture document [OdpRm3] is more prescriptive especially in the computational viewpoint. Unfortunately, the level of prescription is given in such a way that it is not directly usable. For example, many of the concepts and rules are based on syntactic aspects of interfaces, e.g. operational interfaces should have uniquely named operations in the context of that interface. Whilst useful and important for writing specifications, such information is not in itself usable in terms of specification fragments that can be used directly in a specification. Similarly, the computational viewpoint explicitly denotes the kinds of actions that can be associated with (computational) objects and interfaces. Unfortunately, this information is also not directly usable for developing specifications since the ordering of these actions and their effects is not given, i.e. the behaviour. Rather, the focus is placed more on providing well-defined concepts and constructs to create and compose such fragments, as opposed to providing real specification fragments.

From these discussions it is apparent that the goal of developing specifications through an architectural approach based on ODP is somewhat limited. Despite the

[2] Although we note here that some problems have been found through the architectural semantics work.

problems of ODP prescription, the goal of architectural specification is still a desirable one. We propose that an approach based on frameworks is more likely to succeed.

3.0 Introduction to the Concept of Frameworks

The concept of framework based software engineering has been developed 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.

A framework can be regarded as a collection of pieces of software (or specification[3] fragments) that have been developed to produce software of a certain type or niche [ToscaD6]. A framework is only partially complete. Typically, they are developed so that they have holes or flexibility points in them where service[4] specific information is to be inserted. This filling in of the holes is used to develop a multitude of services with (slightly) differing characteristics. Of course, the number of holes in a framework is directly proportional to the amount of work required to complete the service. A framework might be all but complete except for one hole, e.g. where an isolated choice of behaviours that the service can exhibit is possible. Typically though there are likely to be numerous holes in frameworks that have to be filled. It might be the case that the holes are in some way dependent upon one another. For example, a framework might have holes left to deal with costing and performance issues of the service, e.g. a higher cost means a higher throughput or picture resolution. From this we can classify different models of framework based on the type of holes they leave.

3.1 Framework Classification

Frameworks can be classified in many ways. Most obviously they can be classified based on the expected services to be generated from them, e.g. multimedia conference services, multimedia on demand services, telephony services, ... Another more abstract way in which frameworks can be classified is based on their types of holes. We consider this way since it opens up some discussions on what frameworks really are as well as the issues in their (formal) modelling. Frameworks can be classified into:

- frameworks with holes that can only be filled by certain (well-defined) behaviours. These behaviours are determined in advance, i.e. when the hole was left. This corresponds to the service framework offering several non-deterministic choices of behaviour and the filling in of the hole corresponds to selecting one of these possible choices. This approach is very direct but unlikely to be the norm. Having such a degree of prescription goes against the nature of frameworks to a

[3] We note here that the term specification might equally apply to business or system specifications.
[4] In the following we use service to mean both software and/or specifications.

certain extent, i.e. they are intended to be abstract models of numerous services as opposed to models of a service with different possible behaviours that have to be selected. Nevertheless such an approach allows validation of the services created from frameworks to be achieved most straightforwardly.

- frameworks with holes based on delegating behaviour. Thus a hole here might consist of a behaviour that accepts a message and redirects it (if necessary) to some framework specialising component. Thus potentially complex systems can be decomposed into less complex sub-systems thereby enabling unnecessary information, and associated semantics, to be abstracted from (hidden). The principles of abstraction and decomposition are presented in Part 2 of the ODP-RM [OdpRm2]. In terms of framework validation, this model is simplest to validate since the detailed processing associated with (certain) inputs can be omitted.

- frameworks with holes left where the detailed modelling of the service behaviour has not been done. This type of hole requires a certain amount of engineering to be filled. This model of a hole is more likely to represent the true nature of frameworks as discussed above. Several potential problems with this model of a framework have to be addressed. Firstly, checks are required to ensure that the holes left are filled in correctly. The level of checking can be done in several ways depending upon how the hole itself is represented.

  - it might be the case that the checks are done on a syntactic level, e.g. through ensuring that the inserted behaviour is syntactically compatible with the hole it fills in (ala C++ and virtual functions). This model is likely to be relatively easy to achieve, however, this level of checking is unlikely to be sufficient to achieve interworking, i.e. this approach only supports interconnectivity issues.

  - it might be the case that holes have abstract models of the behaviours that can be inserted into them. This is arguably the most useful (and probable) model of a hole but it raises several complex issues that have to be addressed for such an approach to be realised.

We discuss these approaches and their SDL representations in sections 4 and 5.

3.2 Development of Frameworks

Frameworks can be developed in numerous ways. Perhaps the most obvious of ways is to take an existing service and extract its generic features so that classes of similar services can be produced. This notion of taking a developed service and producing a framework is termed generalisation .

Generalisation may be regarded as capturing the main features of a given application (or system or enterprise or ...) in such a way that the design experience can be re-used. This can be at many levels of abstraction. It might be the case for

example that certain objects that comprise a service are common to a collection (class) of similar services. It might be the case that certain interfaces or operations contained within an interface are common to a class of services. For example, in a multimedia conference system there are likely to be common operations to start, stop, suspend and resume the conference session say. Having a generic model of a basic service enables a potentially broad class of services dealing with multimedia conferencing to be created, e.g. where different policies apply regarding the existence of conference chairmen say, or how invitations are handled.

If generalisation can be regarded as developing a general model of a class of services and embodying them in a framework, then specialisation can be regarded as taking the abstract framework and making it less abstract. Typically this is likely to be through supplying behaviours at flexibility points. Just as generalisation can be made in several stages to result in more and more abstract frameworks, so specialisation can be done in several stages; each stage resulting in a more deterministic framework and hence a narrower set [5] of services that can be created from the framework.

A framework only has a finite number of holes. Each specialisation of a framework fills in one or more of these holes either completely or partially. It might well be the case that the user can instantiate a framework without having to supply all of the holes associated with that framework. This can be achieved for example through having default hole behaviours, e.g. *this service behaviour is not yet implemented*, comments might be supplied when the user wishes to see what the behaviour of a particular service instance actually is at a particular stage of the specialisation process.

There are many other issues that are associated with frameworks. For example, the modification of existing frameworks, i.e. re-use of a framework where the intended level of re-use was not initially identified. These modifications might be through extending or reducing the functionality of the framework. Similarly, combination of frameworks is likely to be an important aspect in the success of frameworks. We note here that ODP provides the basic conceptual concepts for addressing these issues, e.g. behavioural compatibility, conformance, refinement, incremental modification, subtyping, and composition. For brevity we do not focus on these issues but consider features of the formal language SDL that can be used for modelling frameworks.

4.0 Formal Languages for Modelling Frameworks

The previous discussions have introduced a broad idea of frameworks in terms of the approaches of generalisation and specialisation. We consider now how these aspects impose requirements on formal languages used to model frameworks. Given the strong relation between frameworks and object orientation, languages used to formally describe frameworks should support object orientation. Whilst

[5] In the sense of the services being more similar.

numerous formal languages have encompassed aspects of object orientation to varying degrees, few have had such investments or done as much to incorporate object oriented modelling approaches as SDL [Sdl92]. Our focus in the following sections is thus on the extent that SDL supports framework modelling and usage.

4.1 Aspects of SDL for Dealing with Generalisation

It could be argued that generalisation requires two features in a language. Firstly, there is the need for modelling the behaviour identified as being generic. Without being more prescriptive on the form of this behaviour it is difficult to state precisely what features a language is expected to possess. Different formal languages have their own advantages and disadvantages for specifying different kinds of behaviours. Secondly there is the need for the selected omission of behaviours identified as service specific, i.e. the language should have features for modelling holes. SDL has several concepts that are applicable to the modelling of holes. We discuss some of these briefly.

4.1.1 Generic Types and Formal Context Parameters

Generic types are types that refer to names not bound to complete definitions. These names are termed the formal context parameters of the type. They are enclosed in angled brackets and specified immediately after the type declaration. Examples of generic types include: system types; block types; process types; service types; procedures; signals and sorts. Formal context parameters can be processes, procedures, remote procedures, signals, variables, remote variables, timers and sorts.

To use a generic type requires that actual context parameters are supplied that satisfy any constraints associated with the context parameters. These constraints can include signature constraints, e.g. the actual parameter must possess a particular operation, or by an *atleast* constraint which is a requirement on any actual context parameter to be a subtype of the type stated in the constraint. An example of a generic type is:

```
Process Type InvHandler
    < SIGNAL setUp(InvPolicy)
      newtype InvPolicy
        operators
          getMinforStart: InvPolicy -> Integer;
      endnewtype >
```

This process type can be used to represent different kinds of invitation handling policies for the starting up of services. All of these invitation handling policies should support a check on the minimum number of accepted invitations before that service can be started. We note that the semantics associated with this operation is established when the generic type is actualised which is itself dependent upon the logic of the service itself, e.g. two people at least are required for a multimedia conference, nobody (zero) required for starting up a chatline service.

4.1.2 Virtual Types

Perhaps the most direct mapping of flexibility points into SDL is through virtual types. These are types in a supertype that may be redefined by a subtype. It is possible that a subtype may also allow virtual types of the supertype to be redefined. Where this is the case the virtual types of the subtype are prefixed with *redefined*.

There are some constraints on the redefinitions of virtual types. The minimum (and default) constraint is that any redefinition of a virtual type must be a specialization of the virtual type itself. Virtual types thus represent concepts in SDL where specific behaviours can be inserted. An example of a virtual type is given by:

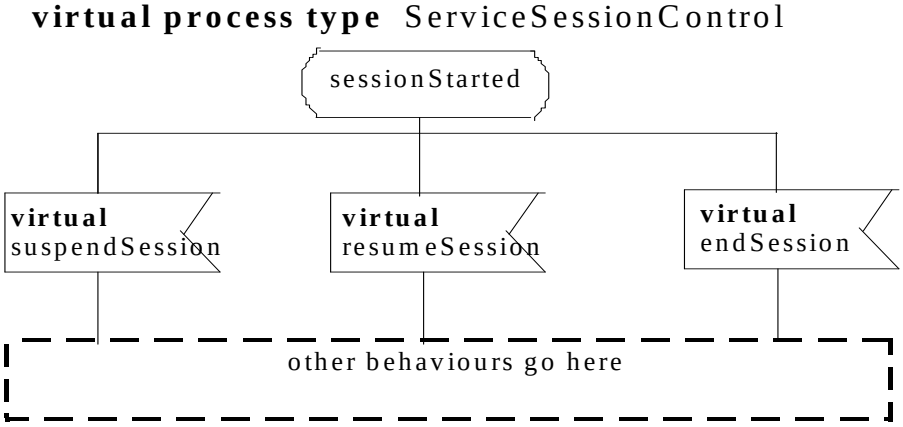

Figure 1: Virtual Types in SDL

Here the transitions related to suspending, resuming and ending a session are made virtual so that the behaviour of the (virtual) process type can be modified (redefined).

4.1.3 Packages

Packages are used for collecting together types that can re-used by systems. This is achieved through a *use-clause*. It is likely that the package construct can be used to re-use basic (generic) models of services. This might include having specific process types representing the objects that are common to the class of services that can be generated from that framework.

4.1.4 Addressing of Delegated Components

If the hole is based on the delegation of behaviour, then this implies that components in the framework send signals to and get signals from the environment (of the framework) whenever service specific behaviour is required. The components the behaviour is delegated to may or may not be explicitly labelled in SDL. Requirements can be placed on the recipient of the delegated behaviour, e.g. the receiving process set is named or certain channels are used to transfer the message. An example of how delegation can be achieved is shown in Figure 2. For brevity we do not show any checking on the delegation of behaviour.

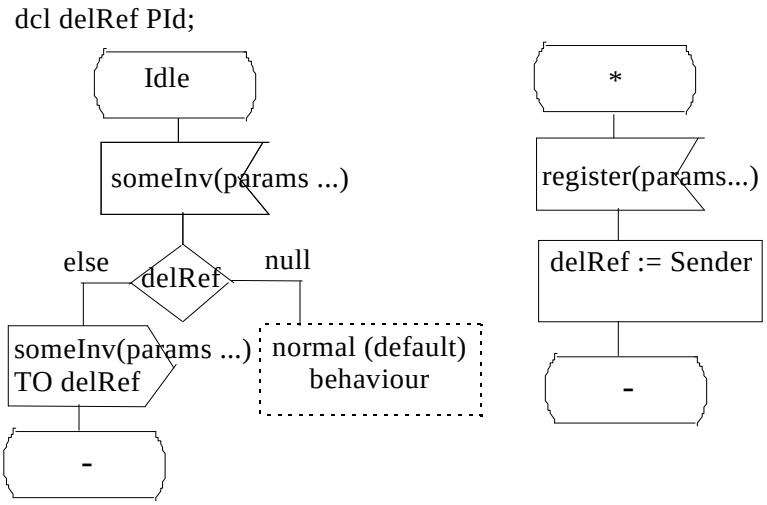

Figure 2: Delegation in SDL

Here if the process identifier has not been set, i.e. it is *null* then the basic process behaviour occurs, otherwise invocations redirected to the relevant (delegated) process.

4.2 Aspects of SDL for Dealing with Specialisation

From the previous sections, it is apparent that SDL has certain features that can be applied to model the generic features of services, i.e. it allows for behaviour to modelled and for unspecified aspects of the behaviours to be left. The next question is how can these unspecified behaviours be filled in.

4.2.1 Actualisation of Formal Context Parameters

Supplying actual types that satisfy the constraints given by formal context parameters of generic types can be used to specialise a framework. This model of framework specialisation makes specific the abstract types that could be inserted into the framework. It should be noted that it is quite possible for a generic type to be only partially actualised, i.e. not all formal context parameters required for making that higher order type a first order type have been supplied. An example of actualisation of formal context parameters is given by:

process type IH1 **inherits** InvHandler < setUp1, InvPolicy1 >

For brevity we assume the signal setUp1 and data type InvPolicy1 are defined in some surrounding scope.

4.2.2 Specialization of Virtual Properties

A type may be specified as a specialization of another type. A subtype may have properties in addition to the properties of the supertype and it may redefine virtual local types and transitions (see section 4.1.2). Subtypes are specified through prefixing the supertype with inherits. As discussed previously, it is possible for subtypes of types containing virtual types themselves to be supertypes of other types through redefinitions. This approach to layering of the type hierarchy is conducive to having multi-levels of framework specialisations. An example of the specialisation of virtual properties is given by:

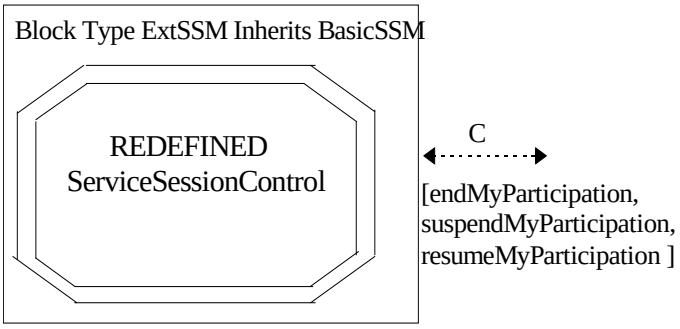

Figure 3: Specialization of Virtual Types in SDL

Here the virtual process type defined in section 4.1.2 is extended to allow for the suspending, stopping and resuming the usage of services by individual users. For brevity we do not show the detailed representation of the extension (and the redefinition of the virtual transitions) or blocks where the virtual process is defined.

4.2.3 Existence of Delegated Components

Specialisation of a framework that uses delegation requires that the components are supplied that receive the signals that have been delegated from the framework component. These delegated components should satisfy any constraints that have been established, e.g. be a member of the appropriate process set.

5.0 Application of Frameworks in the TOSCA Project

The TOSCA project is concerned with the creation and validation of services based on the Telecommunications Information Networking Architecture (TINA). This architecture is based on the principles of ODP, e.g. it considers viewpoint languages and objects having potentially more than one interface. TINA is more prescriptive than ODP however in that explicit IDL and *textual* descriptions of the expected behaviour of many of the architectural components have been identified.

TINA itself is decomposed into four main parts: Service, Network, Management and Computing Architectures. In TOSCA, the main focus is on the Service Architecture [TinaSA]. 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.

TINA has identified certain components that one would expect to find in these sessions and prescribed IDL to describe them. The TOSCA project is considering how frameworks can be created based on the access and service session components. The relation between TINA sessions and some of the components are shown in Figure 4.

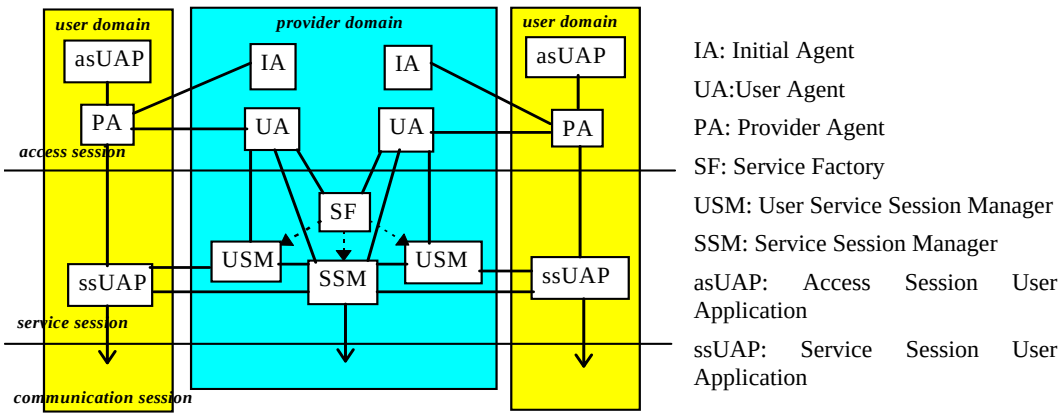

Figure 4: TINA Service Architecture and Associated Components

As discussed previously in section 4, there are several ways in which aspects of frameworks can be represented in SDL. All of these approaches are currently being invesitgated in TOSCA. The underlying model for all of them is based on the IDL from the Service Architecture. In TOSCA, tools have been developed that allow IDL (and the TINA Object Definition Language ODL) to be mapped into SDL [Y.sce]. As with other CORBA language bindings, the mapping produces client stubs and server skeletons. These then form the basis for the behavioural description of the service components. Unlike other tools, the mapping of IDL to SDL within Y.SCE allows for the handling of exceptions: an essential feature in object-oriented distributed systems.

We note here that the availability of an IDL to *formal* language mapping is unique to SDL. Other mappings are likely to prove difficult if not impossible to achieve due to the lack of support for interface references as first class citizens, i.e. they can be passed around as parameters. In SDL they are represented by process identifiers.

As discussed though, IDL is only a (syntactic) basis for understanding systems: semantics are essential if truly open systems as defined in section 1 are to be developed. Whilst arguably, not as rigourous as other formalisms, SDL does allow behaviour to be modelled in an intuitive way. It is often the case though that even with intuitive representations of behaviour, the complexity of the systems being described result in specifications that become unwieldy (and hence complex). TOSCA has recognised this and developed an approach whereby frameworks can be specialised in a user-oriented manner. This is achieved through a graphical user interface (paradigm tool [Cadenza]) that allows for frameworks to be specialised (and subsequently services to be created) in a graphical and intuitive manner. This user interface is designed for usage by potentially non-technical people to develop services in a rapid fashion. The output of this user interface is SDL that specialises the framework. Once specialised, the framework can then be validated using appropriate SDL tools. Currently TOSCA is investigating the exact role of this validation. Two main ways are being considered: on-line and off-line validation. On-line validation is likely to be in the form of animation showing the user in a graphical way what happens when a specialised framework (service) is simulated. This can only be regarded as a partial validation though since it does not check all behaviours of the specification. Nevertheless it serves as a useful guide for the service designer to see that the service does what they actually want. Off-line validation is likely to entail checking for specific properties of the specification. One example of such as property is deadlock freedom (implicit signal consumption).

6.0 Conclusions

This paper has attempted to show how specifications can be architected using an approach based on frameworks. Whilst initial results in terms of the general

principles behind frameworks and their modelling in SDL have been encouraging, the next major test is - as with formal methods generally - one of scalability. The services that TOSCA wishes to model and validate are *real world* software engineering activities, requiring the modelling of numerous complex objects interacting in non-trivial ways, e.g. where services might influence the behaviour of one another in potentially undesirable ways. Currently different techniques for addressing such complexity are being investigated in the context of service validation, i.e. avoiding the well known problem of state space explosion. Initial ideas to avoid this are based around partial validation techniques, where framework components and their specialisations are validated in isolation, i.e. where only environment interactions are modelled and not the complete environment.

The question might be asked as to whether SDL is the best language to be used for expressing semantics or behaviour more generally. The language has an intuitive representation but can sometimes be verbose. The language lacks features such as invariants *ala* Z. The aim of this paper was not to decide upon which language is best suited for expressing behaviour though - as the Architectural Semantics work has shown[6], different languages have their own advantages and disadvantages in terms of abstraction, decideability, tractability etc. Rather our intention was to show how specifications might be engineered. We believe that the concept of frameworks and their modelling using the features of SDL are one such approach to achieving this.

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

6.1 Acknowledgements

The author is indebted to the partners in the TOSCA project and fellow GMD colleagues. 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. The project is funded under ACTS proposal AC237.

7.0 References

[Ansa] ANSA, *The ANSA Reference Manual*, Architecture Projects Management Limited, Poseidon House, Castle Park, Cambridge, UK, 1989.

[Cadenza] For more information see http://www.teltec.dcu.ie/tosca

[Corba] Object Management Group, *The Common Object Request Broker Architecture and Specification: Revision 2.0*, Object Management Group, Inc., Framington, Ma., July 1995.

[Estelle] ISO/IEC, Information Processing Systems - Open Systems Interconnection - *Estelle - A Formal Description Technique Based on an Extended State Transition Model*, International Organization for Standardization, ISO-9074, Geneva, Switzerland, 1989.

[6] This is highlighted in the development of information and computational specifications.

[Kjt] Kenneth. J. Turner, *Relating Architecture and Specification*, Computer Networks and ISDN Systems: Special Edition on Specification Architecture, March, 1997.

[Leopold] H. Leopold, G. Coulson, K. Frimpong-Ansah, D. Hutchison and N. Singer, *The Evolving Relationship between OSI and ODP in the New Communications Environment*, Technical Report MPG-93-16, University of Lancaster, England, 1993.

[Lotos] ISO/IEC, Information Processing Systems - Open Systems Interconnection - *LOTOS - A Formal Description Technique based on the Temporal Ordering of Observational Behaviour*, International Organization for Standardization, ISO-8807, Geneva, Switzerland, 1989.

[OdpRm1] ISO/IEC, *Basic Reference Model of ODP: Overview and Guide to Use*, International Standard 10746-1, ITU-T X.902, Geneva, Switzerland 1997.

[OdpRm2] ISO/IEC, *Basic Reference Model of ODP: Foundations*, International Standard 10746-2, ITU-T X.902, Geneva, Switzerland 1996.

[OdpRm3] ISO/IEC, *Basic Reference Model of ODP: Architecture*, International Standard 10746-3, ITU-T X.903, Geneva, Switzerland 1996.

[OdpRm4] ISO/IEC, *Basic Reference Model of ODP: Architectural Semantics*, International Standard 10746-4, ITU-T X.904, Geneva, Switzerland 1997.

[OdpRm4a] ISO/IEC, *Basic Reference Model of ODP: Architectural Semantics Amendment*, Working Document 10746-4.1, ISO/IEC JTC1/SC21/WG7 N10532, Geneva, Switzerland 1997.

[OsiRm] ISO/IEC, Information Processing Systems - Open Systems Interconnection - *Basic Reference Model, International Organization for Standardization*, ISO-7498, Geneva, Switzerland, 1994.

[RosKjt1] Richard O. Sinnott and Kenneth. J. Turner, *Applying Formal Methods to Standard Development: The Open Distributed Processing Experience*, volume 17, Computer Standards & Interfaces, pages 615-630, 1995.

[RosKjt2] Richard O. Sinnott and Kenneth. 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.

[RosKjt3] Richard O. Sinnott and Kenneth. J. Turner, *DILL Specifying Digital Logic in LOTOS*, Proceedings of Formal Description Techniques VI, R.L. Tenney, P.D. Amer, U. Uyar, pages 71-86, Elsevier Science Publishers, 1994.

[RosMsc] Richard O. Sinnott, *The Formally Specifying of Electronic Components in LOTOS*, Masters Thesis, Dept. of Computing Science and Mathematics, University of Stirling, 1994.

[RosPhd] Richard O. Sinnott, *An Architecture Based Approach to Specifying Distributed Systems in LOTOS and Z*, PhD Thesis, Dept. of Computing Science and Mathematics, University of Stirling, 1997.

[Sdl92] ITU-T, International Consultative Committee on Telegraphy and Telephony, *SDL Specification and Description Language*, International Telecommunications Union, CCITT Z.100, 1992.

[TinaSA] TINA-C, *Service Architecture version 5.0*, Telecommunications Information Networking Architecture Consortium, 16 June 1997.

[ToscaD6] TOSCA Deliverable 6, Service Creation: The TOSCA Paradigm and Framework Approach, August 1997. See http://www.teltec.dcu.ie/tosca.

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

[ZStandard] ISO-IEC, *Information Technology - Programming Languages their Environments and System Software Interfaces*, Z notation, ISO/IEC CD13568.