Using roles with types and objects for service development

Rolv Bræk

NTNU department of Telematics and Sintef Telecom and informatics, Trondheim Norway
e-mail: rolv.braek@informatics.sintef.no

Introduction

Due to increasing user expectations, ICT convergence, de-regulation and the vast possibilities of the ubiquitous Internet, the ability to quickly develop and deploy new services will become an increasingly important competitive factor in the future. Whether the solution will be a “smart” or a “stupid” network may be debated to lengths, but there is no question that flexibility to provide an open range of services will be a key feature. Considering the wide range of multi-media services enabled by new technology, safe and rapid service development will pose a formidable challenge. Even in traditional “single services networks” where “services” are seen as features of a basic service, e.g. the basic telephone call, this has been difficult to achieve. “Intelligent Networks” were introduced to improve the service flexibility in such networks, but are inherently constrained by the basic call assumption, and not well suited for the emerging “open services network”. An open-ended approach where service modules can be developed and combined in very flexible ways, by different suppliers, to accommodate completely new multi-media services is needed. This paper is about achieving open ended-ness and flexibility using roles, types and objects.

Roles are increasingly popular modeling notions. Unfortunately the term “role” is used to mean different things in different approaches, and it is not always clear how it relate to other important notions such as types or classes and objects. The paper seeks to clarify what roles are and how they can be utilized to achieve open-ended flexibility in systems and services. It will not go deeply into the technical details, but concentrate on overall principles and how they may be utilized in combination with existing methods and languages, in particular UML [OMG 99], SDL [ITU 93] and MSC [ITU 96b].

It is not reasonable to believe that service logic in the future will fit into a monolithic pre-defined framework, like the IN architecture. It is more likely that service logic will be provided by open-ended “object communities” that are allowed to evolve and change more freely as long as the result is useful and without undesirable effects. One idea that will be presented here is to associate required and provided behaviour roles with types, so that provided and required roles may be aligned when objects of the types are linked together. Not only does this enable an incremental validation of links to take place, it also opens the possibility that objects may learn to play roles dynamically and thereby dynamically adapt to new services and situations.

About roles, types and objects in general

According to normal terminology, objects are phenomena; i.e. tangible entities with substance and behaviour, while types are concepts representing the features shared by all objects belonging to the type and roles are properties that may be associated with both objects and types.

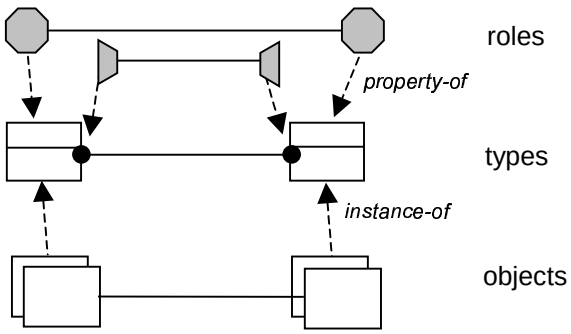

Figure 1 Roles, types and objects

Roles

The notion of role is used extensively in everyday talk either to characterize objects in relation to other objects, like “father”, or to indicate their function/responsibility in an organization, like “project leader”.

A classical example of the functional use is found in the theatre. The ensemble of a theatre consists of actors (objects) while the plays they perform are defined in terms of roles (properties) the actors shall enact (exhibit) during the play. Roles are described (in plays) independently of particular actors, but in order to be performed, they must be assigned to actors. Actors, on the other hand exist independently of particular roles, but are able to play many different roles. Note that roles does not fully specify the actor behaviour, but give room for “interpretation” allowing actors to put some of their own personality into the play.

Family relationship is a classical example of relational roles. “Father”, “son” and “cousin” all denote roles that a person may play in relation to other persons and not the person itself. Each person is normally able to play many different relational roles, and will do so slightly differently from other persons, depending on personality.

Clearly these two notions of role are very useful when seeking to describe and understand reality, and therefore they have gradually made their way into modeling techniques and methods. UML for instance, uses roles in both meanings. An AssociationEnd has a role name, which denote a relational role. A ClassifierRole represents the role of an object in a collaboration, which is a functional role, see [OMG 99], [Rumbaugh 99].

Relational roles will be termed *association roles* in the following, in keeping with UML terminology[1]. They originate from data modeling, in particular Entity-Relationship (E-R) modeling, where they have been used to name roles for the endpoints of relationships, with only informal semantics carried by the name. UML has taken this a bit further, allowing static interfaces to be specified for AssociationEnds.

Functional roles will be termed *service roles* in the following to indicate that they represent the part an object plays in a service, function or task. They originate from several sources. Within telephone engineering, for instance, it has been a long-standing convention to describe telephone services using role names like the “A-subscriber” and the “B-subscriber” assuming that a person may play both roles. Similar informal use of service roles is found in many strands of life, for instance to denote roles in a human organization such as “team-leader”, “secretary”, or “accountant”. A more rigorous use of service roles is a

[1] Strictly speaking UML reserves the term “AssociationRole” for associations linking ClassifierRoles in Collaboration diagrams.

key element of the OORAM approach [Reenskaug 95]. In UML, Collaboration diagrams may be used to describe service roles statically (as ClassifierRoles), while Sequence diagrams may used to describe dynamic role behaviour. Several ClassifierRoles may be associated with a Classifier, but a ClassifierRole can bind to only one Class, so ClassifierRoles are not really independent concepts that can be re-used in different Classes. If ClassifierRoles are to model service roles in general, this UML constraint must be removed.

Obviously a Class must provide all the operations and accept all signals specified in its Classifier roles, but apart from this neither UML nor OORAM, deals with role behaviours and how role behaviours compose into Class behaviours in terms of state machines.

In the SISU method [Bræk 93] the notion of *role-behaviour* was introduced to give dynamic semantics to interface roles. A role-behaviour was used to define the observable behaviour of an object at an interface. The original idea was that behaviour roles would help to simplify validation of interfaces between objects by reducing the state space needing to be explored. In order to be faithful to the full object behaviour, roles has to be *input consistent*, and it turned out that lack of input consistency was a strong indicator that errors like unspecified reception and deadlock would occur in any context where objects of the type were used. A simple rule was formulated that could be checked for each type with very limited effort, even manually, without needing to compose objects and performing state space exploration. Moreover it was possible to use the ideas constructively to avoid introducing errors in the first place.

The overall use of roles in a methodology was further elaborated in TIMe [Bræk 99], but TIMe did not elaborate on role-behaviours.

Here we shall use 3 notions of role-behaviours:

- An *association role* describes the visible behaviour of one object at an association end (which may be associated with an interface). A special case is an *interface role*, which describe the visible behaviour at an interface. Note that it is possible for several associations to use the same interface, and in that case, the interface role will be a composition of all association roles (see below) on that interface.

- A *service role* describes the visible behaviour of one object in a service, function or task.

- A *session role* (or *service association role*) is the visible behaviour of one service role in one association. A special case is an *interface session role*. A given service role may have several associations, or interfaces, with other service roles, and each correspond to a service, or interface, association role.

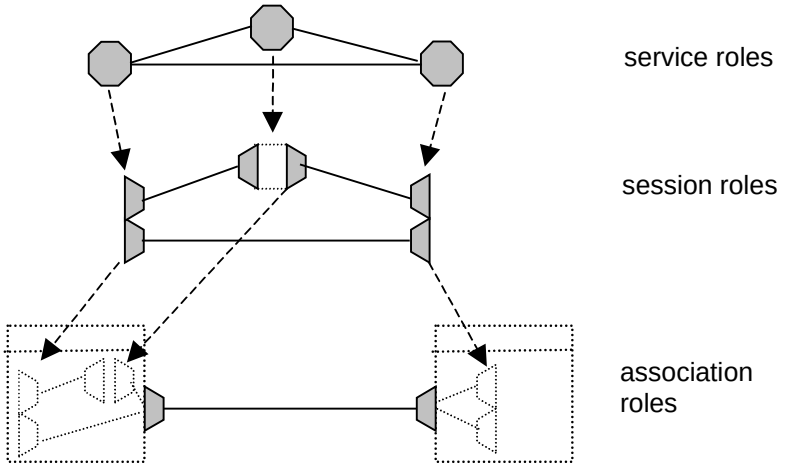

Figure 2. Illustration of role relationships

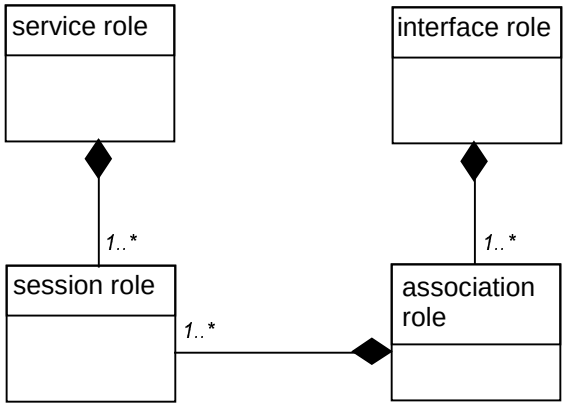

Figure 3. Role relationships

All these roles can be seen as subsets of the object properties observable from a particular angle, like a projection. Just like projections, roles are useful in several ways:

- *Architecture definitions.* Interfaces are central to system architectures, and often more important than types, because interface roles can be used to define interfaces precisely without binding the types of objects that uses the interface. This is a central point in a Plug-and-Play architecture. Static interface definitions like those of CORBA IDL, UML and SDL is a necessary minimum, but not sufficient to guarantee that objects will inter-work correctly in all cases. A better guarantee can be given if interface definitions also specify the dynamic interface role-behaviour.

- *Reuse*. Roles are reuse entities in their own right. This follows from the independence between roles and objects and the fact that a given role sometimes is to be provided by many different objects and types. All types accessing a given interface, for instance, must provide the same interface roles. Roles can also be used as search criteria to find types and objects that can play given roles, for instance as part of a trader's functionality.

- *Design synthesis*: When designing a type, the roles it shall provide serve as specifications for the type design. It should be noted here that role-behaviours only specify behavior *properties*. They are not behaviour design units. Design synthesis requires that information is added, and that roles are

composed. Therefore the insight of a human designer is needed in most cases, although precisely defined role-behaviours may help to greatly facilitate the design process. Behaviour design synthesis is an open research area where partial solutions exist, but general approaches are yet to be found.

- *Design verification*. A given design is verified against the roles it shall provide. It means either to check that the roles can be derived from the design by making projections, or to check that the roles will be performed by the design. As explained in [Bræk 93], roles also enable useful checks on the internal design consistency to be performed.

- *Validation of associations and links*[2]. Association roles can be used like plugs and sockets to validate associations (between types) and links (between objects). It has been shown in [Bræk 93] that roles help to reduce the state space of a reachability analysis.

Types (or classes) and roles

A type (or class) represents the features that all objects considered as its instances have in common. A type definition will normally have two main parts: a context part defining the external associations and interfaces, and a content part defining the internal construction in terms of attributes, operations and behaviour. This general pattern for a type definition is illustrated in Figure 4.

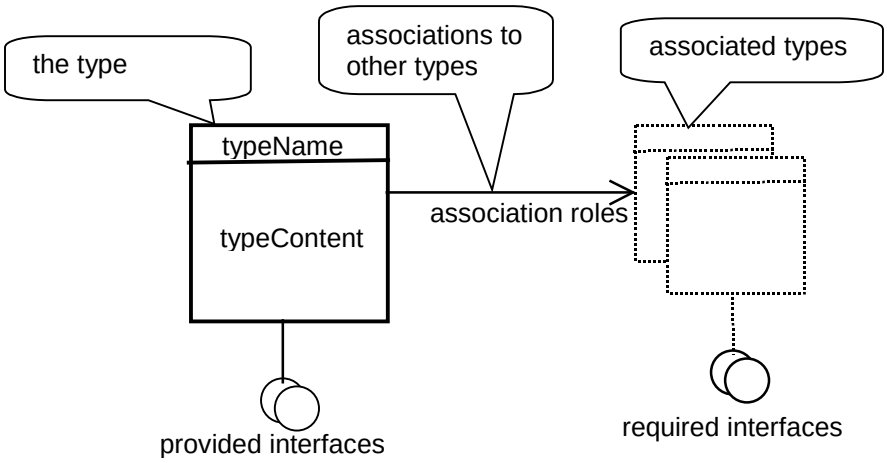

Figure 4. Pattern for a type illustrated using UML notation

This picture is implicit in the UML approach where a type is modeled as a Class. Its static context and content is defined using Class diagrams, while the dynamic behaviour is defined using State Machines. Associations relate a Class to other Classes, and AssociationEnds have roles. An AssociationEnd in UML, represents a set of objects of the target Class (the class it connects to) and implicitly describes properties of the source Class (the Class at the other end of the Association). In addition to this, an AssociationEnd (role) defines requirements on the set of target objects that can validly be linked to a source object. AssociationEnds can identify required interfaces, but there is no notation to describe the behavior in connection with interfaces. An Interface in UML represents a collection of input operations[3].

---

[2] In accordance with UML terminology, associations are relationships between types (classes), while links are relations between objects.

[3] one would expect that signals were part of the interface, but signals are not allowed in UML v1.3. beta.

Note that in UML, the target Class of an AssociationEnd is specified, and this may be overly restrictive. More flexibility can be obtained by specifying only the *required roles* without binding the Class, since this will allow instances of any Class to be linked as long as the required roles are satisfied. In an open evolving system where new types may be included as the system evolves, this is more appropriate.

The purpose of a type is often related to a set of roles (both service roles and association roles) that its instances shall play. Different types may play a given set of roles differently. The content part of the type will define exactly how the roles will be *provided*. In addition to providing roles, a type will *require* that (instances of) associated types play given roles.

This leads up to a relationship between behaviour roles and types as illustrated in Figure 5.

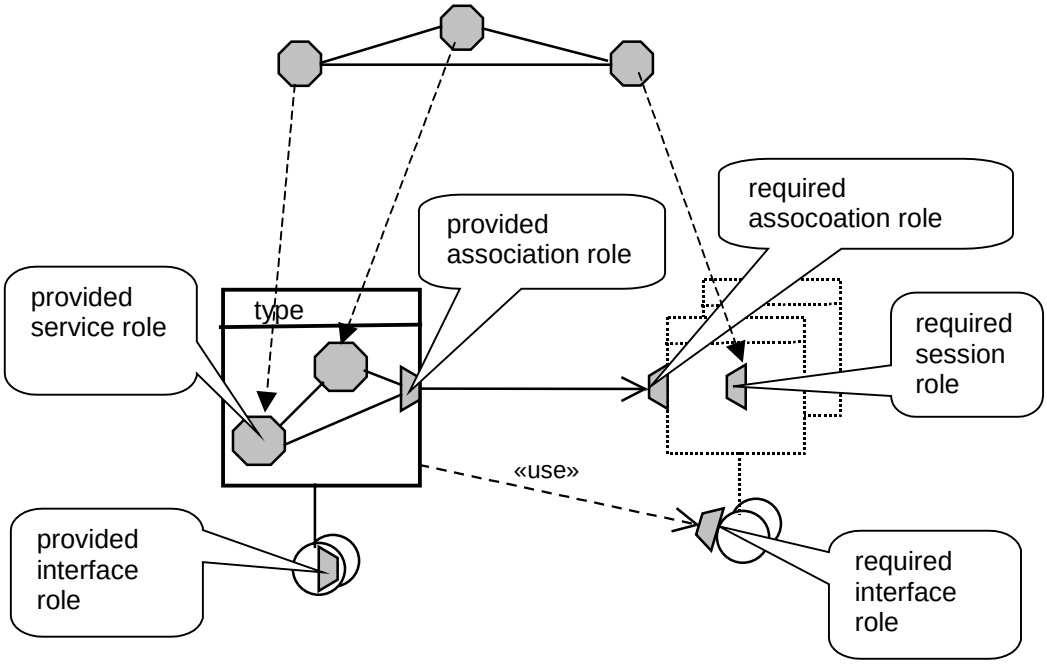

Figure 5. Roles related to a type illustrated using UML.

Note that this view is relative to the type being defined. Provided roles are the roles that objects of the type can play. Required roles are the roles they expect other objects to play.

The set of provided roles (that an object can play) is called its role *repertoire*.

In UML Class diagrams, Associations and Interfaces serve mainly to give properties to the Classes (types) defined by the diagram. Still it makes sense to check consistency in connection with Associations and Interfaces. For an Association between two Classes (types) to be valid, the provided roles must *contain* the required roles on both ends. Similarly for interfaces: the provided interface roles must contain the required interface roles. Statically this means to check that the required sets of signals and operations are contained in the provided sets. Dynamically it means to check that the required role behaviours are contained in the provided role behaviours. (What this means will not be elaborated here, but is similar to checking that a MSC [ITU 96b], can be executed by a design, see e.g. [Kristoffersen 91].)

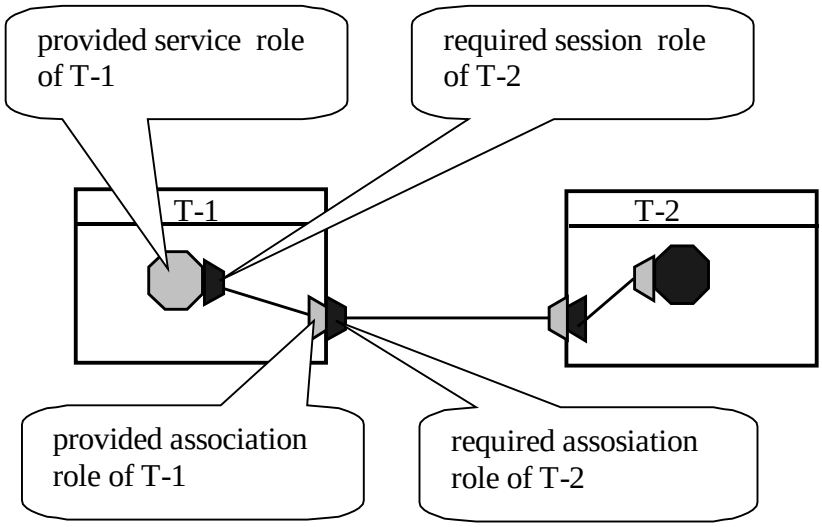

Figure 6 Aligning provided and required roles: required roles shall be contained by provided roles

Although the purpose of a type is to play roles, its content is not composed from roles. The principal reason is that roles by nature, are incomplete and partial views on objects and not construction units. They are properties that shall be satisfied by the construction, and not parts of the construction itself, just as a set of 2D drawings define views on a 3D construction without being part of it.

If roles are to be used actively in behaviour design, like 2D drawings are used in geometrical design, it is essential that the correspondence between roles and designs become well defined. Roles should map to sets of possible design units so that type designs can be defined by selecting and composing such units into a whole that is both internally consistent and compliant with the roles. (Just as in geometrical design.)

This is not common practice today, partly because the concept of behaviour roles is quite new, and partly because the correspondence between behaviour roles and behaviour design units is not well understood. To the extent that roles are used in type design at all, it is normally an ad-hoc approach where the correspondence between roles and design units is lost and not utilized further after the initial design is made. This is a pity because roles seem to have a potential for life-long benefits that could be utilized better if the correspondence between roles and designs were better understood. One possible approach to be elaborated further, is to define *design patterns* corresponding to roles, and define how these patterns can be elaborated and composed into correct designs. This is illustrated in Figure 7.

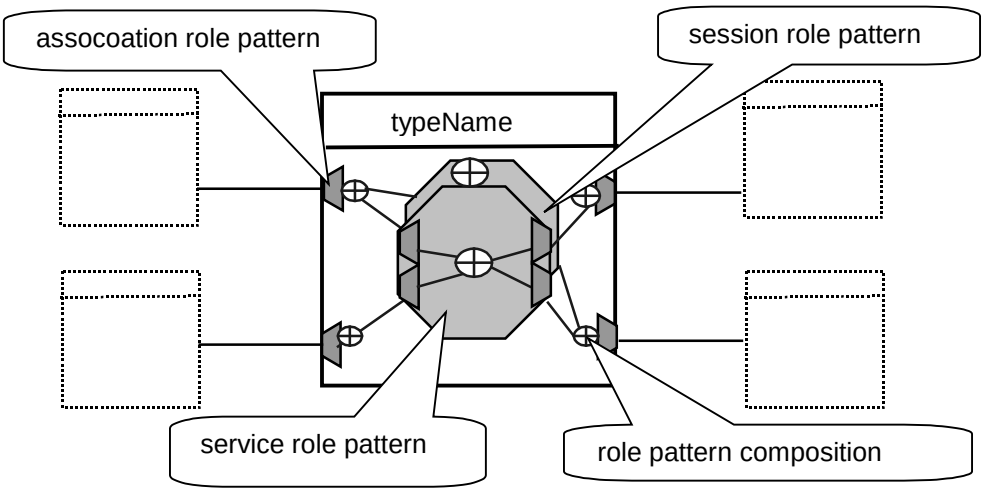

Figure 7. Role pattern composition.

In order to simplify and speed up service development, practicable principles for role design patterns and pattern composition will be of great help, not only to design types, but more importantly to allow roles to be learned dynamically on demand.

The composition of role design patterns can be seen to constitute a *personality*. The personality determines the way that roles are combined and realized. It may add some flavor to the interpretation of roles as long as the resulting behaviour stays within the bounds of the roles. Exactly what this means will not be explained here, but the relationship between roles and personality is similar to the relationship between specifications and realizations.

In traditional designs, objects have a fixed type with a fixed role repertoire. If the type of an object is allowed to change dynamically, the object will be more adaptable to changing requirements. Changing the type means, in most cases, changing the roles, so type changes is one way to adapt objects to changing roles. Another way would be to change the roles without changing the type. This could provide even more flexible adaptations, but it would require a type concept that supported changing roles. In effect it would require *actor* types that were able to “learn” new roles dynamically. This, in turn, would require that roles had “implementations” that could be composed dynamically into a whole, something like manuscripts that can be interpreted along with other manuscripts. (If such manuscripts can be devised, a possible implementation can be Java Applets/Beans.) Such manuscripts may be developed from role design patterns.

Composite types

In order to handle larger systems, it is necessary to support aggregation. This means that objects may be collected into aggregate entities, normally over several levels of aggregation and that such aggregate entities can be defined using types. For this purpose the content part of a type must define a structure of objects.

Although UML supports composition as a special kind of association, composite types are not supported, except for attributes and except for the optional nested graphical presentation of composition. In this respect SDL [ITU 96a], [ITU 93], [Ellsberger 97], [Olsen 94], is better as composite types are well-defined language concepts (systems, blocks, processes).

We will adopt the SDL approach in the following and assume that types can be composed from instances of types over several levels of composition. Moreover that there are two main categories of objects: *active*

objects and *passive objects*. Active objects have their own behaviour while passive objects only have operations and attributes.

Active objects may contain an internal structure of active objects as well as having their own behaviour and a structure of passive objects. There are two important special cases of active objects:

- Active object containers. These are active objects without own behaviour, having only a structure of active objects (and possibly some shared passive objects).

- Simple active objects. These are active objects without an internal structure of active objects, having their own behaviour and possibly a structure of passive objects (i.e. attributes).

The first case corresponds to systems and blocks in SDL-96, and the second to processes. In SDL 2000 these will be generalized and unified into so-called active entities.

The principle of required and provided roles also applies to composite types, but in this case the provided and required roles result from the internal objects of the type and the own behaviour of the type. Where internal objects participate in external associations or interfaces, the corresponding roles will be propagated to the enclosing type. Roles on internal associations and interfaces will be hidden.

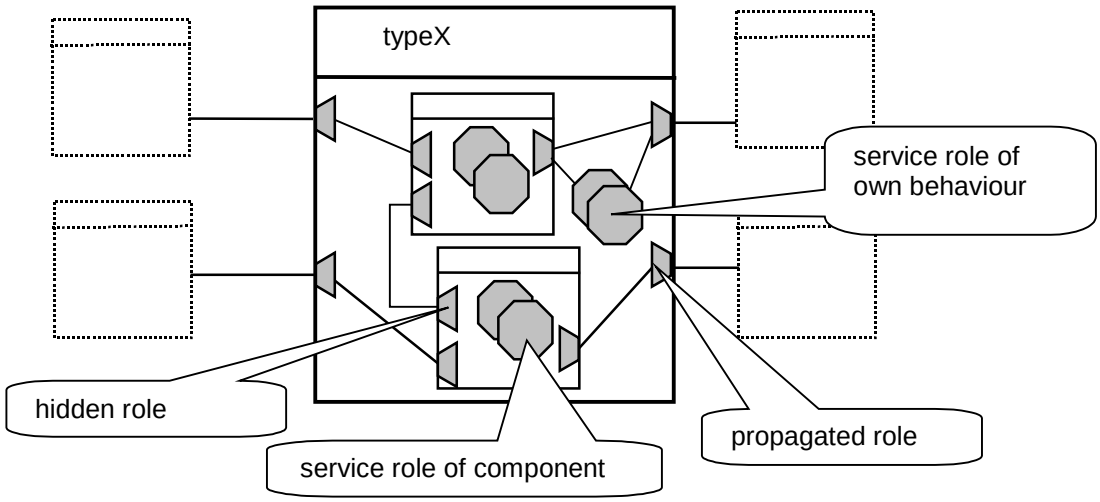

Figure 8. Illustration of a composite type

Objects and object structures

Object structures consist of objects and links between objects where objects are instances of types and links are instances of associations.

When a link is created, it is necessary that the required (association and service) roles are contained in the provided roles. Otherwise the link will be invalid and exceptions may result. Therefore, when a link is created it is necessary for roles to be aligned so that containment is ensured. We consider three levels of *role alignment*:

- *Validation*. Checking that the required roles are contained in the provided roles, and raising exceptions if not.

- *Adaptation*. Negotiating and adapting the roles within bounds given by the object's personality. This is performed by some protocols today.

- *Learning*. Learning to play new roles by receiving role implementations when existing roles are insufficient. Java Applets can be seen as one way of doing this.

Ideally, role alignment should be performed dynamically whenever a link is created, before it is being used. This is not common practice, however. It is more common to rely on off-line design analysis before the system is created combined with error detection and exception handling during use, e.g. by simple input screening.

In the dynamic world of future open services ICT systems, this will be too inflexible and also too error prone, especially if new services and types are to be introduced dynamically. We expect that role learning will become a very important technique in such systems. Role learning can be accomplished even if the general problem of role design patterns and role composition is not fully solved. It is sufficient that a set of (alternative) types exists that can provide the roles and that instances of one of these may be installed on demand in the learning object. In this way installing/downloading objects of predefined types accomplish role learning. Of course, it is necessary that context rules will be satisfied for the downloaded object. The object type to install/download may be selected on basis of this, and in this way the role implementation may be adapted to the receiving actor, e.g. to a terminal with limited capabilities.

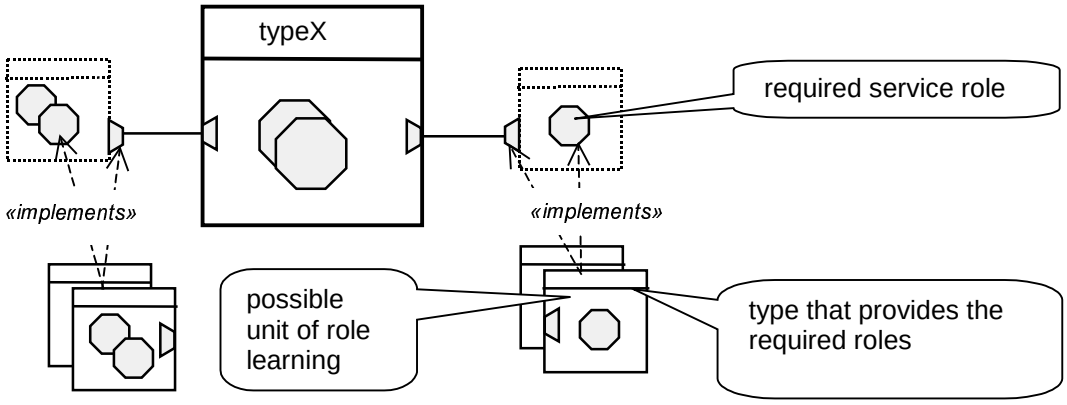

Figure 9. Using types to accomplish role learning

In general we only want object structures that serve their purpose and are well formed in some way. Rules for well formed structures can be classified as *content rules* and *context rules*. Content rules govern the internal composition of (instances of) a type. A (context-free) grammar, for instance, is entirely made up of content rules. Context rules, on the other hand, govern the external use of (instances of) a type. Gate constraints in SDL and interface definitions CORBA style, can be seen as context rules. Traditionally content rules have been more used than context rules. One reason is that content rules are simple to make, another reason is that context rules have not been strong enough to ensure well-formed structures alone. Even CORBA IDL definitions are insufficient because they do not cover dynamic interface behaviour. Our idea of required and provided roles is intended to remedy this.

Content rules restrict the possible structures more than is necessary or desirable in many cases. OMG has realized this, and therefore the cornerstone of CORBA and component-based development in general, is context rules in the form of interface definitions. By only using context rules, it is possible to achieve a higher degree of flexibility, and open-ended-ness. If the only constraints are context rules given for the component types, any object structure is well formed as long as all context rules are satisfied. This is similar to the "Lego principle" where the possibilities are limited only by the "rules of the bricks". Ideally, context

rules should be checked for each composition step and if satisfied they should guarantee that the resulting system is well formed. Unfortunately the static interfaces of CORBA-IDL, UML and SDL does not guarantee correctness. Using provided and required roles combined with role alignment is intended to improve the situation. Given a set of types, any structure of objects will be well formed as long as every object plays the behaviour-roles that other objects in the structure require from them, and this will be a stronger criterion for well formed-ness than possible with static interface definitions like IDL.

In order to serve its overall purpose, a structure must also play the service roles required by its external users. To check that the object system indeed fulfils its purpose, one may compose the object system with models of its environment.

Service development and evolution

Models and object systems

Figure 10 shows the main models that we assume are developed for an (abstract) application system (not considering physical implementation and performance). The first step in the initial development is to define the services using service models with service roles. The next step is to map service roles to types and to design the types. Here the personality of each type is determined and composed with designs for the service roles that the type shall provide. Note that service roles and session roles originate from service models. Association and interface roles originate from types. Finally the object system is created from the types.

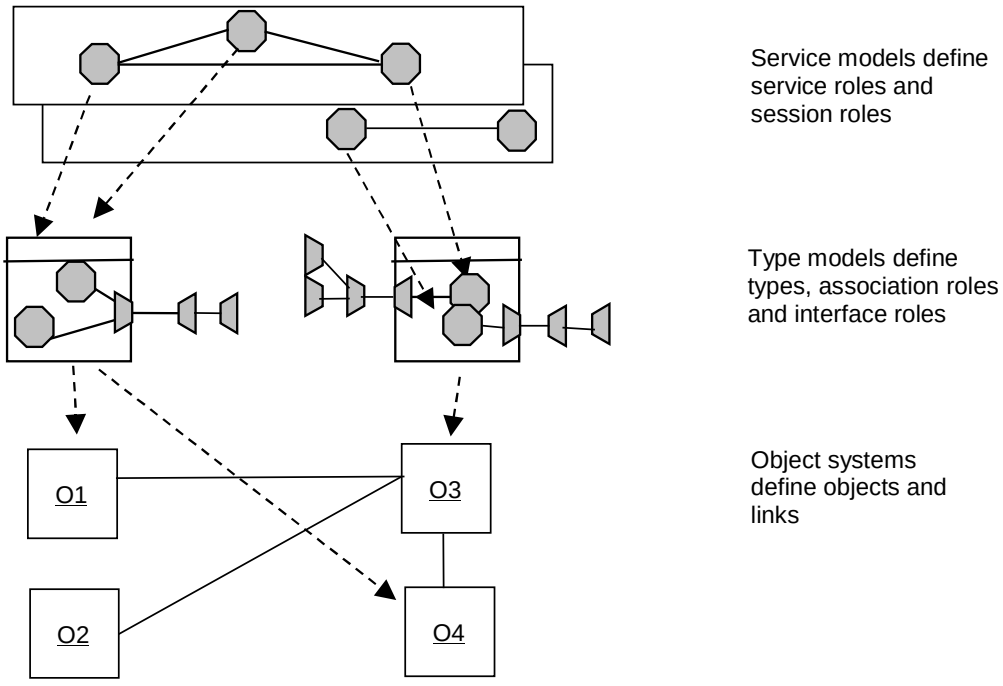

Figure 10. Models and objects in service development

Each service model may consists of:

- UML collaboration diagrams that identify the structure of service roles, but without binding the roles to Classes.

- A set of UML Sequence diagrams or MSC diagrams that (partially) define the overall service behaviour in the form of interactions.

- Possibly some SDL process graphs or UML Statechart diagrams defining the complete behaviour of service roles and session roles. In these diagrams interactions that are not part of a role are hidden, leaving only interactions that are part of the role visible. In order to represent transitions that are triggered by invisible inputs, such inputs are represented by designated input symbols, i.e. *none* in SDL.

The only special here is to use SDL process graphs or UML Statechart diagrams to define role behaviours precisely. Otherwise, similar approaches have been used in the industry for years, e.g.: by Ericsson and Alcatel, although they have not been so explicit about roles.

Making each separate service model is usually neither very difficult, nor very resource demanding. Difficulties start in type design when service models are mapped on types. This is the challenging part that we seek to improve. In order to improve we must understand the fundamental nature of the relationship between services, roles, types and objects:

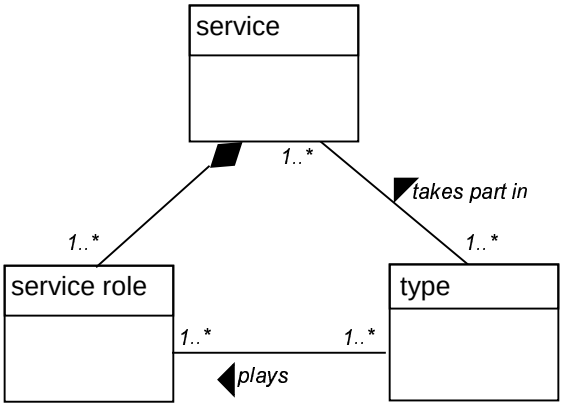

Figure 11. Service-type relationship

- Typical services involve more than one service role, and the roles will be mapped on different types. Therefore a service change typically will impact several types.

- Some types must provide several service roles.

- Many different types may provide a given role.

- Service roles are not independent but interact, both within a service and between different services. The so-called service interaction problem (or feature interaction problem) is related to undesirable interactions between different services (or instances of the same service).

- Service roles may be assigned to objects dynamically when dynamic links are created and may contend for the same objects (actors). Creating dynamic links often involve some kind of resource allocation, and therefore dynamic role assignment is akin to resource allocation. In this light, a resource allocator may be seen as an actor allocator or actor manager/trader. It will respond to requests for actors that can play some given role, perhaps negotiate the role (e.g. CallForward in stead of Busy), and ensure that undesirable interactions are avoided. In [Fritsche 95], a scheme to detect feature interactions by means of service roles was proposed.

Due to the n:m association between services and types, a service change is likely to impact several types, and this is a main reason why service flexibility has been so hard to achieve. In addition:

- Service logic has not been well separated from underlying capabilities and protocols.

- All role implementations have been hard coded.

The solution to these problems must be found primarily in type designs and the principles used to implement roles in types. But some preparations must be made in service models too by describing dependencies between services, how they may be composed and how they shall interact or shall not interact. In order prepare for service role composition, we must seek to identify the dependencies between roles in different services and to include the dependencies into the service models to the extent this is possible. In principle, we need context rules for service roles that can be used when composing role design patterns in type designs.

In order to reduce the impact on type designs, we may partition the set of service roles into *provider-roles* and *client-roles*. The intuition is that provider roles are those that provide the services, while client roles are those that use them. In traditional telecommunication system designs both kinds of roles has been treated the same way, and therefore a service change has impacted types playing provider roles and client roles alike. Typically a new end-user service has required design changes to the user interface as well as to the central service logic, and sometimes even changes to the protocols. Our idea is that client roles can be learned dynamically by client objects as part of role alignment, and thereby reduce the need to redesign the types of the client objects. Much of the success of the www can be attributed to the general ability of client browsers to adapt to server functionality (through the transferred html pages, Java applets, etc.).

The proposed principle is to limit the number of types that must be manually changed to those playing provider roles, using role learning for client roles. This requires a type design that:

- simplify adding or changing provider service roles in its role repertoire;

- can hold descriptions, or references to descriptions, of required client roles in a form that enable client objects to fetch role implementations as part of role alignment;

- can itself learn client roles required from other objects as part of role alignment.

An extension of this is types that have an open role repertoire and can get new roles and/or change roles dynamically.

Evolution and change follow the same overall approach as initial development with the difference that the focus is on additions and changes. Corresponding to the models, evolution and change[4] take place on three main levels:

- *Object change*. These are simply changes to the object structure without changing the types. Reasons may be due to performance (e.g. adding more channels) or usage (e.g. adding new subscribers) requirements. Object changes do not require any changes to the service models or the type models. Note that even in carefully designed systems, object changes may result in invalid links, so role alignment will be useful even if the types are not changed. We may distinguish two cases:

---

[4] in the following we will just use the term “change” for simplicity.

  - Static types. Objects have the same type throughout their lifetime.

  - Dynamic types. Objects may change type dynamically. This is different from deleting an object and then creating a new from a different type, because (some of) the instance attribute values may survive. To some degree software upgrades on a running system may be seen as a case of dynamic types.

- *Type change*. These are changes to the types. They may be motivated by changes in the service roles to be provided (e.g. adding a new service feature), or by other needs for redesign of the types. Type changes must be followed by object changes in order to have effect in a running system. Again two cases may be distinguished:

  - Changes that result in a subtype of the changed type.

  - Changes that result in a completely new type.

- *Service change*. These are changes to the existing services, or additions of new services. The result will be changes and/or additions to the service roles and their behaviour. Before such changes may take effect in a running system, they must be followed by type changes and by object changes, or just by object changes if the type supports dynamic role learning.

Object change is supported, at least to some extent, by most object-oriented systems. Type changes are normally not. Even if type definitions are part of the runtime system, type changes have to be made off-line both for physical and logical objects. Service changes are the most challenging, but they are increasingly important for the industry. When high service flexibility and short time to market is a major competitive factor, support to service changes becomes a prime concern.

Bibliography

[Bræk 93] R. Braek, Ø Haugen; Engineering Real Time Systems - An Object Oriented Methodology using SDL
Prentice Hall, 1993, ISBN 0-13-034448-6.

[OMG 99] Object Management group; OMG Unified Modeling Language Specification (Draft) version 1.3 beta R1, April 1999.

[Rumbaugh 99] J. Rumbaugh, I. Jacobson, G. Booch; The Unified Modeling Language Reference Manual.
Addison-Wesley 1999, ISBN 0-201-30998-X.

[Ellsberger 1997] J. Ellsberger, D. Hogrefe. and A. Sarma; SDL. Formal Object-oriented Language for Communicating Systems.
Prentice Hall, 1997, 0-13-621384-7.

[ITU 93] ITU-T; Recommendation Z.100 ITU Specification and Description Language (SDL).
ITU-T, June 1994,("SDL-92").

[ITU 96a] ITU-T; Addendum to Recommendation Z.100: CCITT Specification and Description Language.
ITU, October 1996, ("SDL-96").

[ITU 96b] ITU-T; Recommendation Z.120 Message Sequence Charts (MSC).
ITU-T, Oct. 1996, ("MSC-96").

[Kristoffersen 91] F. Kristoffersen; Message Sequence Chart and SDL Specification Consistency Check.
SDL´91 Evolving Methods. Proceedings of the Fifth SDL Forum, North Holland: Elsevier 1991.

[Lam 84] S. S. Lam, and A. U. Shankar;. Protocol Verification via Projections.
IEEE Transactions on Software Engineering vSE10(4), 1984.

[Bræk 99] R. Bræk et al; Quality by construction exemplified by TIMe – The Integrated Method
Telektronikk Vol 95 No 1 1999, ISSN 0085-7130.

[Olsen 94] A., Olsen, O.Færgemand, B.Møller-Pedersen, J. R. W. Smith, and R.Reed; Systems Engineering Using SDL-92.
North Holland 1994, ISBN 0 444 89872 7.

[Reenskaug 95] T.Reenskaug, P. Wold, and O. A. Lehne; Working With Objects. Manning: Prentice Hall, 1995.

[Fritsche 95] N. Fritsche; Runtime Resolution of Feature Interactions in Architectures with separate Call and Feature control.
Feature Interactions in Telecommunications Systems III, IOS Press 1995, ISBN 90 5199 238 6.

[Zhao 86] Z. Zhao, and G. v. Bochmann; Reduced reachability analysis of communication protocols: a new approach.
6th International Workshop on Protocol Specification, Testing and Verification, Montreal, Quebec June 1986, North-Holland.