*Kenneth J. Turner. An Architectural Foundation for Relating Features. In Petre Dini, Raouf Boutaba and Luigi Logrippo, editors, Proc. Feature Interactions in Telecommunication Networks IV, pages 226-241, IOS Press Amsterdam, June 1997.*

# An architectural foundation for relating features

**K. J. Turner**
*Computing Science, University of Stirling, Scotland (kjt@cs.stir.ac.uk)*

**Abstract** Consideration of services in the Intelligent Network and telecommunications leads to the definition of ANISE (Architectural Notions In Service Engineering). This is a rigorous language for defining services systematically using a hierarchy of constituent features. The basic telephone call is used as an illustrative example, supplemented by a number of variations that show how ANISE can easily cope with changes to the basic call. An indication is given of how this might be used to detect problem areas that may lead to feature interaction.

**Keywords** Feature Interaction, Formal Description, Intelligent Network, Service Architecture, Telecommunications

## 1 Introduction

The author's belief is that weaknesses in service architecture are an important cause of feature interactions. Of course this is not true of all feature interactions, but it is believed that an improved understanding of service architecture is an important step towards identifying interactions. Architecture is used in this paper to mean the structure of a system specification (here, of an advanced telephone network). More specifically, an architecture is characterised by its components and how they are combined [6]. A service is used to mean the facilities offered to clients (here, telephone users). As the paper unfolds, both terms will be given more precise meanings.

In the present work, the author has concentrated on foundational issues in service architecture and description. The paper concludes with a description of how it is hoped that the foundation will facilitate detection of feature interactions. The thrust of the work is rather different from most attacks on the feature interaction problem. The emphasis on constructing services from well-defined components using well-defined rules is consonant with the architectural approach of ANSA [8]. The building block approach is also similar in spirit to the work of [4], though in their case the building blocks are at a higher level.

Services in the IN (Intelligent Network [3]) are of course directed towards telecommunications. They are relatively low-level since they derive fairly directly from network capabilities. A service in the telecommunications sense generally means some network function that can be separately subscribed to and charged for. A feature might be regarded as one of the constituent parts of a service. However, this is a rather loose distinction since features can be services in their own right. Indeed the IN recommendations to some extent blur the distinction between services and service features.

The immediate aim of the work reported in this paper was to develop an architecture and a language for describing telecommunications services such as those found in the IN. There are technical problems and gaps in the way the IN relates services to features and features to SIBs (Service Independent Building Blocks). For example, SIBs are at rather different levels of granularity and have not been demonstrated to be necessary or sufficient for describing IN services. The way in which Basic Call Processing is treated as a kind of SIB is also unsatisfactory in the author's view. This paper deliberately does not give any special status to services, features, SIBs and the basic call. All are considered to be behaviours of a telecommunications system. Elementary behaviours are taken as the foundation from which more complex behaviours can be built. These behaviours are all termed features. A service is merely a convenient way of labelling or packaging features for marketing purposes.

The major advantage of the approach is architectural consistency, since all the elements of a service have the same status and are described in the same kind of way. Also because the approach is compositional, there is a rigorous foundation on which higher level services can be built. This permits systematic definition, formal description, rapid prototyping and methodical analysis. The architecture is user-oriented in the sense that it concentrates solely on the interactions a user has with services.

Section 2 investigates the architecture of IN and telecommunications services, leading to a systematic modelling approach. Section 3 introduces the language ANISE (Architectural Notions In Service Engineering) for describing features and their combinations in a declarative way. Section 4 uses ANISE to describe the basic call, while section 5 extends this with a number of variations. Finally section 6 discusses the future evolution of ANISE, including its use to identify areas of potential feature interaction.

## 2 Issues in Telecommunications Service Description

### 2.1 OSI and IN Services

The author previously worked on the definition of services in OSI (Open Systems Interconnection). This led to the language and tools called SAGE (Service Attribute Generator [5]). However, since OSI services are much more regular than IN services, SAGE was not immediately applicable to telecommunications. The work has therefore been to adapt the conceptual approach of SAGE. The result is ANISE (Architectural Notions In Service Engineering) – a language for rigorous definition of IN and telecommunications services. The emphasis in ANISE is on behaviour as it might be perceived by a user. ANISE aims to use well-defined patterns of behaviour, to allow flexible definition and modification of services, and to formalise and analyse the resulting services.

The ANISE approach is bottom-up, but from a user's perspective rather than an engineering viewpoint. The idea is to construct features and services as combinations of the signals exchanged between a telephone user and the network (going off-hook, dialling a number, etc.). A feature is characterised by the rules for exchanging these signals. Since higher-level features are built from lower-level ones in a consistent way, everything is just a feature in ANISE. Features in the IN are generally at a high level and are relatively close to services. Features in the ANISE sense start out being rather elementary but grow towards the level of the IN.

It is interesting that the ANISE and IN philosophies are almost diametrically opposed. ANISE focuses on user behaviour and so is implementation-independent. ANISE emphasises high-level architectural issues and intentionally ignores the details of actually building telecommunications networks. ANISE is intended for describing and analysing services without consideration of engineering issues. ANISE is not required to create specifications that are somehow refined into an implementation (though this is a possibility). By way of contrast, the IN focuses on engineering detail and so is rather concrete. Although the IN defines planes of abstraction and purports to show a relationship between these, in practice everything is driven bottom-up by engineering concerns. As a result, the IN service architecture is rather insubstantial and unsatisfying in the author's opinion. ANISE and the IN approach are thus complementary.

The behavioural structure of OSI services is largely independent of the data parameters carried by the service. The behaviour of IN services is, however, rather more dependent on data describing the call state or the customer profile. Many IN services require relationships among calls; in OSI, connections are almost invariably independent. Some of the following terminology and concepts of OSI are borrowed to describe IN services. Service users interact with a service provider at service access points. (Note that throughout most of this paper, an interaction means just an exchange of information and not interference in the sense of feature interaction.) A service user has a unique title, while a service access point has a unique address. Several instances of a service user may execute concurrently. Interactions with a service provider are called service primitives. Each service user engages in connections – separate series of service primitives. Connections terminate at connection endpoints within each service access point.

The suffix of a service primitive name is used to distinguish its role. A request occurs when a service user asks the service provider to carry out some function; an indication occurs when the service provider notifies a service user, usually in response to a request. A response occurs when a service user acknowledges receipt of an indication; a confirm occurs when the service provider notifies the originating service user of a response. The use of indication, response or confirm depends on the particular service; for example a request may be followed by only an indication or a confirm. Service primitives may carry parameters; in particular, a service data unit is information that is conveyed transparently by the service provider. A self-contained capability of a service is called a facility in OSI; this usually employs all primitives with the same name but differing suffix (e.g. a data transfer facility might use *Data.Req*, *Data.Ind* and *Data.Con*). A reasonably close correspondence can be established between OSI and telecommunications terms, for example: an address is the (unique) number of a telephone, a connection is a call, a connection endpoint is the termination of a line, a service facility is a feature, a service primitive is a signal, and a service user is (usually) the subscriber. In what follows, OSI terminology will be preferred where it seems more appropriate.

The OSI diagrammatic notation for services [2] will also be used; see figure 1 later for some examples. The actions of users and the provider are shown in columns, with time running down the page. Service primitives appear as horizontal arrows because they notionally occur instantaneously. Sloping lines are used within the service provider to show the passage of time due to transmission delays. Normally the vertical relationship between primitives shows a time ordering. A tilde ($\sim$) is used where the order is undefined (i.e. the primitives can occur in either relative order).

## 2.2 Relating Endpoints and Calls

The description of a telephone network can focus on the behaviour of individual endpoints that describe the state of users. Calls between endpoints are then relegated to the status of 'glue' that temporarily binds these endpoints. The endpoint-oriented view seems to be the conventional way of regarding telephone networks. The view is essentially one-sided, defining what one user sees of the network behaviour. Certain kinds of coordination are easier in this viewpoint, specifically how the user's actions affect what is observed in future by that user. In Terminating Call Screening, for example, a user may ask not to receive calls from certain numbers or area codes; coordination is needed between the user's request and future incoming calls. The endpoint-oriented view promotes user state and makes call behaviour subservient to this. It also ties in naturally with other user aspects such as billing. The view eases coordination at a single endpoint, but complicates coordination among a number of endpoints. To put this another way, an endpoint-oriented view may have difficulties in managing distributed state.

The description of a telephone network can alternatively focus on the behaviour of calls. Endpoints are then relegated to the status of 'appendages' that exist at each end of a call. The call-oriented view seems to be natural as it chooses the call to be fundamental; the primary role of a telephone network is to establish and support calls. It is therefore perhaps surprising that the call-oriented view is not the conventional one. This may be because the endpoint-oriented view is closer to engineering practice. An interesting question is whether users think of using a telephone network in endpoint-oriented or call-oriented terms. It is likely that some feature interaction problems stem from presenting services in the wrong way to users. The call-oriented view is essentially two-sided, relating the behaviour of both endpoints participating in a call. The entire behaviour of the telephone network is then a collection of all such pairwise calls. Since the basic call in this treatment is unidirectional, it is necessary to instantiate the call description for every permutation of user pairs. It would be possible to model calls bidirectionally, i.e. to recognise explicitly that pairs of users can call each other in a symmetrical way. However, this would require unidirectional call behaviour to be duplicated per bidirectional call.

It is significant that Capability Sets 1 and 2 for the IN are single-ended, single point of control – the so-called Type A service. Services such as Multi-Way Calling appear to break this requirement. Type B services, which may have multiple calls segments or multiple control functions, have not yet been standardised. However, it is expected that such services will be added to the IN in future. The author conjectures that IN Type A services would be more naturally described from a endpoint-oriented perspective, whereas Type B services would be better in a call-oriented style. In what follows, a call-oriented approach is adopted because it is felt to be more appropriate.

## 2.3 Handling Call Instances

Features may be invoked in an isolated fashion. This is the case for a connection-less service in which consecutive invocations are unrelated to others. However, features usually have some relationship to each other. This applies to a connection-oriented service (e.g. a normal telephone call) in which call establishment must precede the voice phase and call clearing. OSI uses the concepts of association and connection to indicate that service facilities are related. However, this is not general enough since there are many possibilities between purely connection-less and connection-oriented.

The more general notion of an interaction group (IG) is therefore introduced. This is a collection of interactions that should be considered as related. Such a group needs to be known to each user by a locally unique reference called an interaction group identifier (IGId). Connections (such as telephone calls) are interaction groups, and connection endpoint identifiers are interaction group identifiers. If there is only one instance of a user (i.e. the user may be engaged in only one call at a time), the interaction group identifier is 1:1 with the user's telephone number. If there are several instances of a user, the interaction group identifier is logically the telephone number qualified by a local identification of the line.

Interaction groups arise in telecommunications when there may be several calls simultaneously on one telephone number. For example, this applies to a PABX with multiple network lines: a call to the single PABX number selects one of the available lines. In the case of Call Distribution or a Hunt Group, it could be said that there are multiple interaction groups for one number. Alternatively it could be said that there is one group per 'real' number, there being a strategy for selecting which number is chosen when a call is made.

IGIds have only local significance for a user, so a set of identifiers (one for each user in a call) is associated with one interaction group. In the context of telephone calls, an IGId identifies one endpoint of a call, and the behaviour of a call is parameterised by the IGIds at each endpoint. There are several possibilities for how these IGIds might be fixed.

Call behaviour could be instantiated for defined sets (usually pairs) of users. This would allow calls only between these endpoints and would thus represent a limited call topology. Although this is conceivable, it is unlikely to apply to a realistic network since normally almost all users can call each other. At the other extreme, both IGIds could be left open. Potential call instances would thus constitute a pool of call resources. The size of this pool would reflect the network capacity though, of course, the pool could be as large as the number of users. As an intermediate possibility the calling IGId could be fixed, with only the called IGId left open. A potential call instance would be rooted at one origin but could terminate at any other endpoint. In this case, the number that was dialled would fix the called IGId. This third approach corresponds most closely to conventional telephony: each user sees a telephone as a means of connecting to any other. Call behaviour would be re-used for successive calls to other users from the same telephone.

Abstractly speaking, there is a relation between telephone numbers and line identifiers (IGIds). This association, *Assoc*, is of type $Num \times IGId$. *Assoc* is truly a relation since it need not be 1:1 in either direction. Of course, *Assoc* is just an abstraction of the (complex) way in which telephone numbers are handled inside a real network. *Assoc* might be a relatively static database. Portions of *Assoc* could also be calculated algorithmically (as in a call plan for Freephone services). The way in which *Assoc* is determined is beyond the scope of the service descriptions considered here.

When the user picks up the handset to make a call, the identifier of the line that is activated must be present in *Assoc*. If it is not (perhaps because the line has been disconnected), the user will not be given dialling tone. Similarly if a user tries to call a number that is not present in *Assoc*, the call cannot be put through and will result in unobtainable. This is trivial for ordinary subscribers but a Freephone call, for example, is more complex. In all cases, the requirement is simply that the (number, line) pair appears in *Assoc*. If a number is associated with several lines it will be necessary to select one. At an abstract level this is treated as a nondeterministic choice, though in practice there will be some algorithm to make the decision. The same approach can also handle PABXs with multiple lines.

## 3 Describing Features

### 3.1 Features as Building Blocks

Features are the components of services. Features may be combined into larger features, so a service is effectively just a top-level feature. Features are characterised by their patterns of interaction among users, these interactions corresponding to the occurrence of service primitives. A particular feature may require only some of a service primitive's roles. Also, a feature might be subdivided into two: a request and indication, followed by a request and indication in acknowledgement rather than a response and confirm. For example, *Call.Req/Ind* might trigger *Answer.Req/Ind* instead of *Call.Res/Con*. In such a case, however, there is really just one feature with different naming of its component service primitives.

A service primitive with name like *Call.Req* belongs with others of the same feature in a group with name *Call*. The group name is a label for the request, indication, response and confirm primitives (whichever are used). If a request and indication rather than response and confirm are used in the acknowledgement then a different group name is used. Thus a call establishment feature might be subdivided into groups *Call* and *Answer*.

The parameters of service primitives in a feature may have a defined relationship to each other. In the simplest case, a parameter of an indication or a confirm is identical to that of a request or response respectively. Similarly, a parameter of a response may be directly related to that of the indication. However, more complicated possibilities exist. For Call Diversion or Freephone calls, for example, the called number will not be the same as that given by the caller. The specification of a feature should thus allow for a relation (that may not be identity) between the parameters of request/indication, indication/response and response/confirm.

OSI service primitives may carry any number of parameters. However, it seems that in telecommunications there is generally at most one explicit parameter. This is not really a necessary restriction of ANISE, but is assumed because it simplifies certain issues in the language design.

### 3.2 Feature Patterns and Properties

A study of typical telecommunications services reveals that there are eight basic patterns in the behaviour of features. These are illustrated in figure 1. As an example, a local-confirmed pattern might occur when a caller goes off-hook (request) and obtains dialling tone (confirm). As a further example, a provider-confirmed pattern might occur during a call attempt with dialling (request), sending ringing current to the called party (indication), and sending ringing tone to the calling party (confirm). More examples of these patterns appear in section 4.1.

Each basic pattern may have one of the properties illustrated in figure 2, arranged in a hierarchy that becomes progressively more degenerate. The properties are **single** (one-off), **consecutive** (sequential), **ordered** (overlapped), **reliable** (overtaking) and **unreliable** (lossy). For example a **single** property might correspond to seizing a line before dialling, whereas a **reliable** property would mean that unordered occurrences are fully

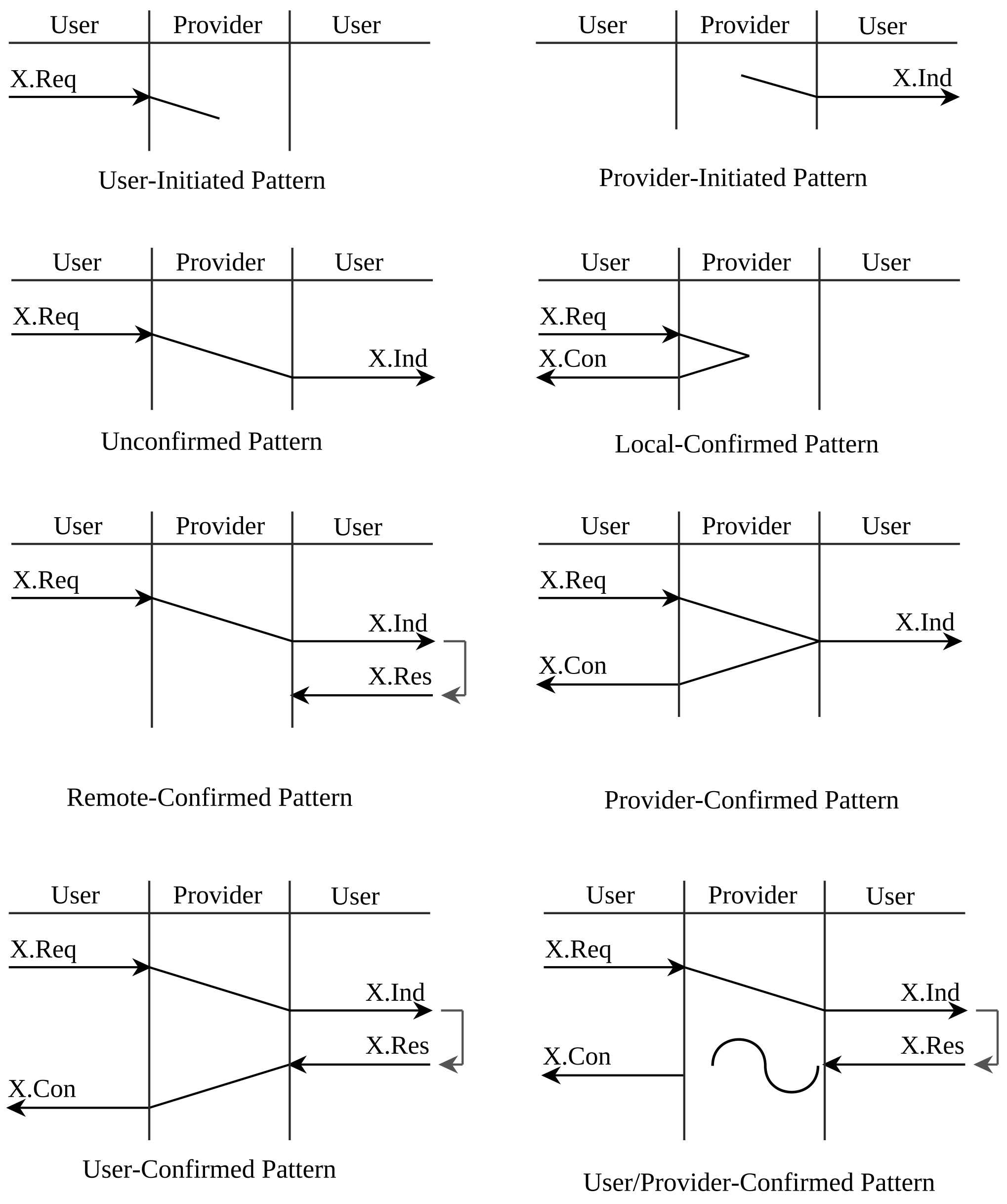

Figure 1: Feature Patterns

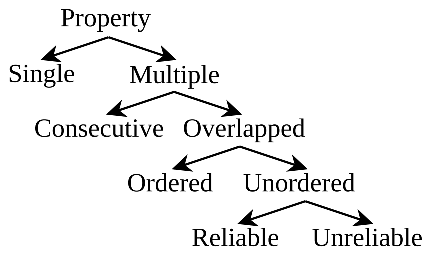

Figure 2: Feature Properties

executed but their relative order may not be preserved (e.g. nearly simultaneous calls to a PABX may be connected in any order). More examples of these properties appear in section 4.1.

Features also have a direction, usually relating a pair of users and therefore interaction group identifiers. Although some services are symmetrical, there can be asymmetries in what users can invoke. For example, some users may be allowed only to initiate calls while others may be allowed only to answer them (i.e. Incoming/Outgoing Call Barring). Within a call, perhaps only the responder may speak or only the originator may be allowed to break the connection (e.g. a recorded announcement in either case). Features are therefore specified unidirectionally; a symmetrical service can simply allow features to be invoked by either user.

## *3.3 Feature Description*

Features and their combinations are declared using the language of ANISE. The language is described informally in this paper, but there has been preliminary work on a denotational semantics using LOTOS (Language Of Temporal Ordering Specification [1]). However, the essence of ANISE is architectural and its semantics could in principle be given in other ways.

Elementary features are declared in one of two ways:

> **feature**(*direction*,*pattern*,*property*,*group*)
> **feature**(*direction*,*pattern*,*property*,*group1*,*group2*)

The meaning of such a declaration is the behaviour specified by the parameters. The direction is **12** or **21**, depending on which user initiates the feature; user 1 is conventionally the left-hand user in a time-sequence diagram, user 2 is the right-hand user. The pattern is one of those given in figure 1 and the property is one of the leaves in the tree of figure 2. Features or combinations are implicitly parameterised by the IGId at each of the endpoints where they operate.

In the case of a user-initiated, provider-initiated or unconfirmed pattern, only one group is given. The group name is the same as that of the service primitives involved (e.g. group *Dial* for primitives *Dial.Req/Ind*). In the case of a confirmed pattern, two groups may be given. This allows for the possibility that the group names differ, e.g. a subdivided call establishment feature may have group names like *Call* and *Answer*. A group may also cite an explicit service primitive parameter (if there is one) in parentheses, e.g. *Call(Num)*. The parameter of a response/confirm may differ from that of a request/indication even if the primitives are all in the same group. In such a case the second group must be given although only the parameter is required, e.g. *Call(Num),(CallingMess)*. All service primitives implicitly carry an IGId as a parameter so this is not declared.

Some basic service primitive parameter types are pre-defined so that they may be used immediately in the declaration of features. The available types are: *CalledMess* (issued to the called party while establishing a call or while it is in progress), *CallingMess* (received by the calling party while establishing a call or while it is in progress), *Num* (a sequence of decimal digits such as a telephone number), and *Voice* (a service data unit representing a segment of speech). Other types may be used freely as service primitive parameters, but their definitions must be added by the specifier. The definitions of the above types may also need to be modified or replaced. For example, a specific telephone number structure may be needed or specific options may be defined.

## *3.4 Feature Combinations*

In principle, features could be combined in a limitless number of ways. However, telecommunications services tend to use a limited range of combinations. These build on one or more behaviours as parameters – those of basic features or their combinations. The meaning of a combinator is the composite behaviour definition. Each declaration takes the form:

> *combinator*(*parameter_1, ... ,parameter_N*)

Such a declaration stands for the behaviour given by its parameters, combined in a particular way. A feature group is given as parameter when the combined behaviour applies only to a particular group within it. Combinators may be built up into larger expressions. Sometimes a single large expression for a service would be unwieldy, or would require repetition of sub-expressions. In such a case a part of the overall behaviour may be defined by:

> **define**(*Name*,*Behaviour*(...))

where *Name* would be used as a parameter to other combinators. Typically this is useful for giving a name to the behaviour of each feature.

ANISE uses the concepts of termination and result because telecommunications features (unlike OSI facilities) are often behaviourally related through values. For example, the user may not dial a number unless the previous step of seizing the line resulted in dialling tone. Separate primitives might be used to distinguish between success and failure (e.g. *Call.Con* and *Call.Err*). However, it is simpler and closer to normal practice to indicate success or failure through a parameter value (e.g. *Call.Con(RingTone)* or *Call.Con(SubsBusy)*).

Table 1: Summary of ANISE Syntax

| Declaration | Meaning |
|---|---|
| *Assoc* | set of associated telephone number and line identifier pairs |
| **associates**(*behaviour*) | service primitive parameter (telephone number) is associated with IGId (line identifier) |
| **alternates**(*behaviour*) | behaviour alternates in each direction |
| *behaviour* | feature declaration or combination |
| **collides**(*behaviour*) | copies of behaviour execute separately in each direction, mutually reinforcing on collision |
| **define**(*name*,*behaviour*) | gives name to behaviour |
| *direction* | **12** (user 1 to user 2), **21** (user 2 to user 1) |
| **disables**(*behaviour1*,*behaviour2*) | first behaviour may terminate second |
| **duplicates**(*behaviour*) | copies of behaviour execute separately in each direction simultaneously |
| **enables**(*behaviour1*,*behaviour2*) | when first behaviour succeeds, second may start |
| **enables_after_ack**(*behaviour1*,*behaviour2*) | when first behaviour executes response or confirm, second may start |
| **enables_after_try**(*behaviour1*,*behaviour2*) | when first behaviour executes request or indication, second may start |
| **enables_on_result**(*behaviour1*,*behaviour2*,*result*) | when first behaviour succeeds with given result, second may start |
| **feature**(*direction*,*pattern*,*property*,*group*(*param*)) | behaviour with given characteristics and group |
| **feature**(*...*,*group1*(*param*),*group2*(*param*)) | behaviour with given characteristics, for request and acknowledgement groups |
| **global**(*name*,*Assoc*,*behaviour*) | gives name to global behaviour for given associations |
| *group* | common part of service primitive names |
| **interleaves**(*behaviour1*,*behaviour2*) | two behaviours execute separately |
| **interrupts**(*behaviour1*,*behaviour2*) | first behaviour may interrupt and restart second |
| **interrupts_after_ack**(*behaviour1*,*behaviour2*) | first behaviour may interrupt and restart second, after second executes response or confirm |
| **interrupts_after_try**(*behaviour1*,*behaviour2*) | first behaviour may interrupt and restart second, after second executes request or indication |
| **loops**(*behaviour*) | behaviour repeats on successful termination |
| *name* | identifier (letters, digits, underscore) |
| **overtakes**(*behaviour1*,*behaviour2*) | execution of first behaviour may begin later than second behaviour but finish earlier |
| *pattern* | **local_confirmed**, **provider_confirmed**, **provider_ initiated**, **remote_confirmed**, **unconfirmed**, **user_ confirmed**, **user_initiated**, **user_provider_confirmed** |
| *property* | **consecutive**, **ordered**, **reliable**, **single**, **unreliable** |
| **reverses**(*behaviour*) | behaviour executes in opposite direction |
| **unique_ids**(*group1*,*group2*,*behaviour*) | service primitives in first group allocate unique IGIds, those in second group deallocate them |
| **withholds**(*group*,*behaviour*) | service primitives of group may be temporarily held off in behaviour |

Behaviour may just stop; this is regarded as unsuccessful termination and produces no result. Normally, however, behaviour will terminate successfully. In this case, the behaviour is defined as producing a result – the parameter value for the last service primitive that occurred. If the last service primitive occurring in a feature does not have a parameter then a void value is assumed. The notion of 'last service primitive' is well-defined for nearly all of the patterns in figure 1. For example, the result of a provider-confirmed feature is that of the confirm. In the case of a user/provider-confirmed feature the values in the response and the confirm are not tied, so the result is taken as that of the confirm. Apart from terminating unsuccessfully or successfully a behaviour may also recurse (i.e. loop). Clearly no result is possible in this case. Considering the properties shown in figure 2 it will be seen that a feature cannot produce a result unless it is described as **single**.

In addition to the explicit result value, the IGIds in use are also produced as part of the result from behaviour. This allows the called IGId to be fixed for the benefit of later behaviour. The functionality of a feature or combination is therefore considered to be:

$$\text{IGId} \times \text{IGId} \nrightarrow \text{IGId} \times \text{IGId} \times \text{Result}$$

There is insufficient space here to describe ANISE in detail. Instead it is summarised in table 1 and explained through examples in what follows.

## 4 Describing The Basic Call

### 4.1 Basic Call Features

This section uses ANISE to describe the basic call; section 5 progressively adds features to this. For the moment only isolated basic calls will be considered, but multiple concurrent calls are covered later. Some major modelling

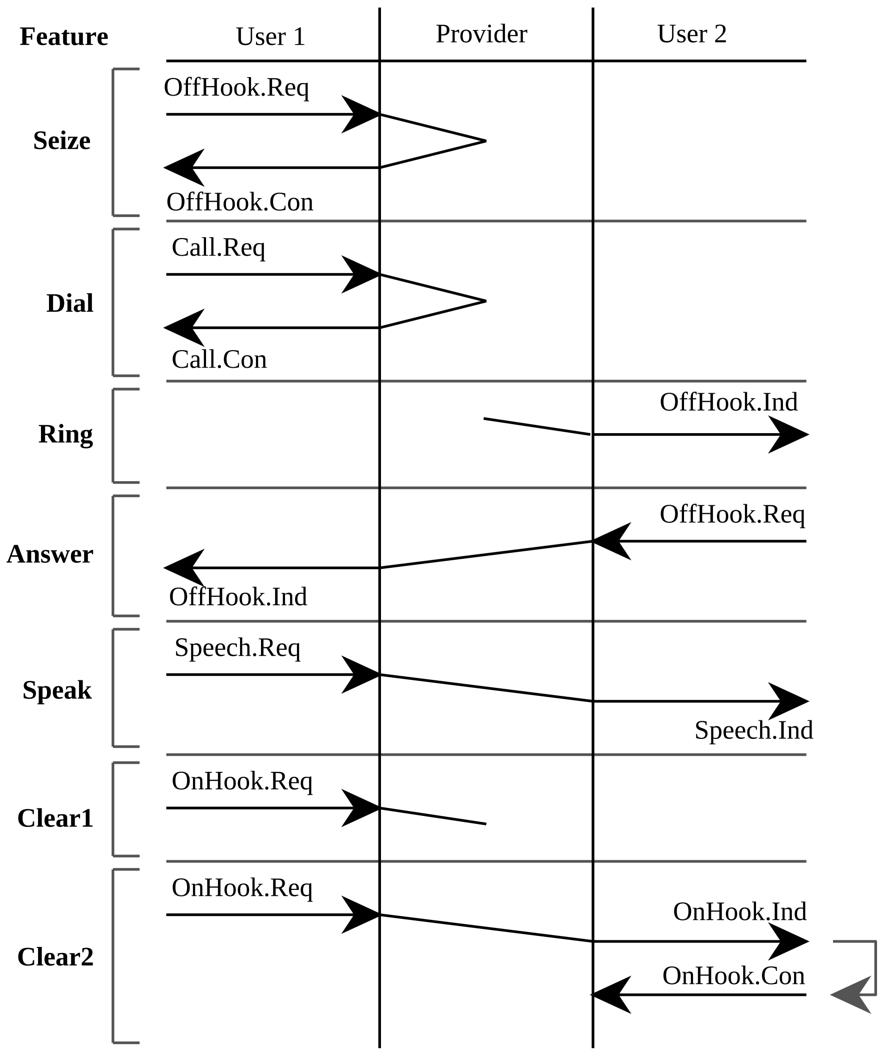

Figure 3: Features of the Basic Call

issues were dealt with in sections 2.2 and 2.3. Many other detailed issues have been addressed in ANISE though they are not discussed here.

The features identified for a basic call are shown in figure 3. This is unadorned with details of features such as their parameters, properties and relationships. The ANISE declarations that follow make these explicit. The meaning of the behavioural building blocks – the features – is now presented, with an explanation following each line in the ANISE description.

**define**(Seize,**feature**(**12**,**local_confirmed**,**single**,OffHook,(CallingMess)))

The feature for seizing the line is called *Seize*. The action of seizing the line takes place from calling to called party, i.e. in the 1→2 direction (even though only user 1 is involved here). The pattern is locally-confirmed since the other user is not contacted; the telephone network returns dialling tone after the telephone is picked up. The line is seized just once for a call, so the pattern has a single instance. Because the pattern is locally-confirmed, it declares a request primitive *OffHook.Req* corresponding to picking up the telephone. Because the pattern is locally-confirmed, it declares a confirm primitive*OffHook.Con* with *CallingMess* as explicit parameter. This will normally indicate dialling tone, but there are other (unlikely) values this type may take such as equipment busy or unobtainable due to a network problem.

**define**(Dial,**feature**(**12**,**local_confirmed**,**single**,Call(Num),(CallingMess)))

The pattern is that the provider confirms what happened after dialling. Normally the remote exchange reports that the called party is being rung so that the local exchange can send ringing tone to the caller. Since the pattern is provider-confirmed, *Call.Req* is declared with an explicit parameter of type *Num* for the telephone number. Note that the lower-level detail of collecting dialled digits into one number is abstracted as a single higher-level parameter for *Call.Req*. Since the pattern is provider-confirmed there is *Call.Con* with parameter of type *CallingMess*, hopefully conveying ringing tone. However there are other possible values such as subscriber busy or unobtainable.

**define**(Ring,**feature**(**12**,**provider_initiated**,**single**,OffHook(CalledMess)))

The provider (i.e. the network) may issue an *OffHook.Ind* when the called party is rung. From the point of view of the called party this indication is spontaneous, though of course it results from the caller dialling. An *OffHook.Ind* is declared with an explicit parameter of type *CalledMess*. Normally this will just be ringing, but other possibilities include Distinctive Ring and Call Waiting tone.

**define**(Answer,**feature**(**21**,**unconfirmed**,**single**,OffHook))

This feature operates from user 2 to 1 without confirmation. *OffHook.Req* and *OffHook.Ind* are declared without an explicit parameter. The *OffHook.Ind* is a little strange in terms of current telephony. It is an explicit signal yet in practice it is the absence of a signal (ringing tone stops). However, in a different kind of telephone there might be some indication that the called party had answered (e.g. a lamp illuminates on the caller's telephone).

**define**(Speak,**feature**(**12**,**unconfirmed**,**ordered**,Speech(Voice)))

This feature is declared in the 1→2 direction only since an identical feature operates in the reverse direction (i.e. both parties may speak to each other). Rather than declare the same feature twice, a single declaration is given. A combinator will be used later to create two interleaved *Speak* features. Since speech is regarded as a sequence of voice samples, speech effectively involves a numbered of overlapped transfers that are kept in the correct relative sequence. *Speech.Req* and *Speech.Ind* both carry speech segments of type *Voice*.

**define**(Clear1,**feature**(**12**,**user_initiated**,**single**,OnHook))

Only the caller initiates a one-sided clear, so there is just an *OnHook.Req*.

**define**(Clear2,**feature**(**12**,**remote_confirmed**,**single**,OnHook))

In two-sided clearing, either party can break the call. This will be taken care of later by a combinator that interleaves a copy of the feature with itself. The combinator will also handle the possibility of collision, i.e. both users simultaneously initiating a clear. Remote confirmation means that the user requests a clear, and this is indicated to the other user who confirms clearing. The declaration means that *OnHook.Req*, *OnHook.Ind* and *OnHook.Con* have no explicit parameter. In practice, if one party goes on-hook the other might be given an unobtainable signal (though this may be network-dependent). Since *OnHook.Ind* is normally associated with just one signal this can be left implicit. However for different possibilities (e.g. a recorded announcement or a tone), *OnHook.Ind* could be given an explicit parameter.

## *4.2 Basic Call Combinations*

The features of a basic call have now been described. The relationships between features – their combinations – are now presented, with an explanation following each line of ANISE.

**define**(Call1,**interrupts_after_try**(Clear1,
**enables_on_result**(Seize,**enables_on_result**(Dial,Ring,RingTone),DialTone)))

Purely to break the description up into smaller pieces, the initial behaviour of the call (seizing the line dialling, ringing) is separately named as *Call1*. A one-sided clear by the caller is permitted once an attempt to make a call has begun. Specifically, *OnHook.Req* may occur following *OffHook.Req*. Once the call has been interrupted by the clear, the behaviour of the call repeats. If the plain **interrupts** combinator had been used here, an *OnHook.Req* would have been permitted before *OffHook.Req*! The **interrupts_after_try** variant can interrupt only after the first request (or indication, though this is irrelevant here). Seizing the line permits dialling only if dial tone is received as a result of seizing the line. Any other result, e.g. unobtainable, prevents dialling. Similarly, receipt of ringing tone permits ringing to occur. Any other result, e.g. engaged, prevents further progress on the call. In either case, the **interrupts_after_try** combinator will only permit the caller to go on-hook and try again. The whole behaviour of *Call1* exits when ringing occurs. The choice of result from seizing or dialling is non-deterministic; from the user's perspective only the possible results and not how they arise are relevant.

**define**(Call2,**disables**(**collides**(Clear2),**enables**(Answer,**duplicates**(Speak))))

Again for convenience, the main behaviour of the call (answering and speaking followed by clearing) is separately named as *Call2*. Clearing may prevent speaking at any time. Since disabling is immediately possible, there may be no answer or speech before the call is cleared. By itself speaking continues indefinitely, so this behaviour cannot exit and prevent the disabling action. *Clear2* refers to two-sided clearing. The **collides** combinator makes a copy of this behaviour, defined for the 1→2 direction only, and instantiates it for 2→1 as well. Both copies may execute independently, i.e. either user may clear the call. But in the particular case that both users hang up almost simultaneously, the **collides** combinator ensures that only the *OnHook.Req* at each end occurs (i.e. the indications do not appear). When the call is answered, the **enables** permits speaking. If answering does not occur, no progress through enabling is possible. The next action will therefore be clearing by the caller. The **duplicates** combinator interleaves independent copies of the *Speak* feature in each direction. (*Speak* was defined for the 1→2 direction only.) The two copies run independently and without interference: both users may thus speak without restriction.

**define**(Call,**loops**(**enables**(Call1,Call2)))

This names the overall behaviour of calls between a single, unspecified pair of users. Once the call (i.e. *Clear2*) has finished, it begins again. Note that there are two loops inside the definition of a call. The 'outer loop' here deals with repetition of the whole call. The 'inner loop' (using **interrupts_ after_try** in *Call1*) repeats if dialling or ringing is unsuccessful. The first part of the call enables the remainder of the call provided it results in ringing. If ringing does not occur because the call cannot be put through, no further behaviour is possible from this part of the description. *Clear1* will therefore occur, i.e. the caller will hang up. Ringing permits the second part of the call to proceed. Note the way that the call has been broken up here: *Call1* (seizing, dialling, ringing) and *Call2* (answering, speaking, clearing). There is nothing special about this decomposition, and indeed the call could be described as a single behaviour.

**define**(Assoc,{(*number1*,*line1*), ...})

This declares the *Assoc* relation to be some set of telephone number/line identifier pairs.

**global**(bc,Assoc,Call)

This declares the global behaviour of the basic call service *bc* as generic calls with the given association between telephone numbers and line identifiers.

The description of the basic call just given requires 12 lines of ANISE. Revisit the ANISE declarations and observe the structure of the description: the behavioural building blocks are the features, and these are combined progressively to yield the overall behaviour of the basic call. ANISE descriptions can be fairly compact because the built-in capabilities of the language are quite expressive.

## 5 Describing Variations on The Basic Call

This section mainly deals with isolated basic calls, but multiple concurrent calls are dealt with in the final variation. The variations can be seen as simple features. It was felt better to evaluate the approach in this way first before tackling more realistic features such as Call Waiting or Three-Way Calling. Work is in progress towards full-blown services such as UPT (Universal Personal Telecommunications).

A benefit claimed for ANISE is that the descriptions are modular. This allows selected aspects of a description to be changed fairly readily, and declarations to be re-used in new descriptions. To underline this point, the following shows how variations on the basic call description in section 4 can be accommodated. Interestingly, the changes required to the basic call description are small in all cases so only the differences are given. In the following, the explanation comes after the changes required to the basic call description.

**define**(Call2, ... **reverses**(Speak)) *Recorded Announcement*

A recorded announcement service such as the Speaking Clock effectively permits just the called party to speak. To model this situation requires only replacement of the **duplicates** combinator with **reverses** inside *Call2*. The use of **reverses** is necessary here because *Speak* is defined only for the 1→2 direction.

**define**(Call2, ... **alternates**(Speak)) *Echo Suppression*

Telephone connections sometimes permit only half-duplex speech (e.g. for echo suppression). This requires the **duplicates** combinator to be replaced with **alternates** inside *Call2*.

**define**(Speak,**feature**(**12**,**unconfirmed**,**ordered**,Speech(Voice))) *Leased Line*
**define**(Call,**duplicates**(Speak))

A leased line has no call establishment or call clearing; at least, this is invisible to the user even though it may occur under the control of network management. Although the leased line is defined as carrying speech, in practice it is most likely to be used to link modems (which of course generate signals in the speech band). This allows the whole description to be drastically simplified to the two lines above.

**define**(Call2, ... **enables_ after_try**(...)) *Speech before Answer*

The basic call description requires the caller to be told the called party has answered before receiving any voice signals. If the called party speaks immediately on answering, the initial segment of speech may arrive before the caller is notified that the call has been answered. This is a very fine point that is hardly worth considering, but it shows how ANISE can make such distinctions where necessary. The change needed is to turn the **enables** combinator in *Call2* into **enables_ after_try**. This allows the called party to speak immediately, i.e. *Speech.Req* after *OffHook.Req* but before the corresponding *OffHook.Ind*. The speech segment, i.e. *Speech.Ind*, may be delivered before the caller knows the call has been answered.

**define**(Call2, ... **interleaves**(**duplicates**(**loops**(Dial)),**duplicates**(Speak))) *Extra Dialling*

Dialling after a connection has been set up should have no effect on a normal call. To achieve this, *Call2* must be changed to allow duplicated (1$\rightarrow$2, 2$\rightarrow$1) repeated instances of *Dial*. Implicitly this assumes that speech and signalling do not interfere. The duplicated instances are interleaved with *Speak* so that they are independent.

**define**(Call1,**interrupts_ after_try**(Clear1,Seize)) *Hot-Line*

A hot-line service is a simplified basic call: picking up the handset calls a predefined number, normally resulting in ringing tone. The *Dial* feature of the basic call is thus not needed and can be dropped. This simplifies *Call1* as above.

**define**(Call1, ... **withholds**(OffHook,Seize)) *Exchange Busy*

If the local exchange is busy, going off-hook may not (immediately) seize the line and return dialling tone. In other words, the occurrence of an *OffHook.Req* (i.e. off-hook being noticed by the exchange) may be temporarily withheld for certain lines. The change required in *Call1* is to qualify *Seize* by **withholds**.

**define**(Call1, ... **associates**(Dial) ...) *One-Number*
**define**(Call, ... **associates**(Ring) ...)

Services of the one-number type include Operator Services and Freephone. They are called with a single number irrespective of location. The call is then routed to an appropriate destination (of which there may be more than one); the routing decision may be made by call plan software. ANISE simply assumes the availability of the *Assoc* relationship discussed in subsection 2.3. The change needed to the basic call description is to introduce the **associates** combinator. This was not used for the basic call since there are only two users. When multiple calls are considered in a moment, **associates** will be used to handle this multiplicity. Both *Dial* in *Call1* and *Ring* in *Call* need to be qualified by **associates**. The first of these ensures that the dialled number selects the required destination line. The second ensures that Calling Number Delivery (if in use) supplies the caller's number. In addition, the *Assoc* relation needs to be enriched for the one-number case, e.g. to include Freephone numbers.

**global**(mbc,Assoc,**unique_ids**(OffHook,OnHook,Call)) *Multiple Users*

The basic call description in section 4 defines calls between a single pair of users. This can be generalised to deal with many users who can call each other, but with only pairwise connections. The **associates** combinator is therefore needed. In addition a richer set of number/line associations will be needed. The effect of **unique_ids**(*OffHook,OnHook,Call*) is to allocate an IGId (mark it busy) when an *OffHook* (request or indication) occurs. An IGId will be deallocated (marked as free) when an *OnHook* (request or confirm) occurs. The book-keeping of which lines are busy and which are free is handled internally by **unique_ids**. It is ensured that a busy line cannot be rung (using *OffHook.Ind*) because its IGId will have already been allocated (by *OffHook.Req*). The multiple basic call service, *mbc*, uses **unique_ ids** at the top level to constrain the operation of basic calls in this way. A multi-user service is rather more likely to suffer from call congestion; the solution using **withholds** given earlier could be relevant here. A multi-user service is also more likely to make use of one-number services such as Freephone, and can adopt the approach described previously.

## 6 A Look to The Future

Work so far on ANISE has concentrated on architectural and language issues. However it is the foundation for describing realistic IN-type services and contributing to the detection of feature interactions. The key idea is that new services are generally modifications of the basic call (or of existing services). These modifications may be additions, deletions or changes. Orthogonal additions should not be any problem; 'additions' that affect basic call behaviour should be considered as changes. Removal of basic call behaviour may be problematic if other parts of the basic call rely on this. However, it might be possible to omit certain parts without problems (e.g. to permit only outgoing calls or one-way speech). Changes in basic call behaviour are possible causes of feature interaction and so should be spotlighted for investigation.

ANISE should allow potential areas for feature interaction to be highlighted. The assumption is that all services can be seen as modifications of the basic call. In a very literal sense, a new service will modify ANISE declarations such as those in section 4. A service may thus be defined by its 'deltas' – the textual changes it causes to the basic call description in ANISE. This would allow services to be described compactly in terms of their changes to the basic call. More importantly, the approach would highlight where services overlap. If services modify the same parts of an ANISE description, this is an indication of interdependence and hence of potential feature interaction. The modifications could conceivably be combined compatibly, but the specifier should look carefully at how such integration could be achieved – if at all.

The semantics of ANISE have not yet been defined, though preliminary work has been undertaken on giving denotations in LOTOS. LOTOS is particularly suitable for this in view of its flexible synchronisation mechanisms. In fact, some of the ANISE operators are just thinly disguised LOTOS. The use of LOTOS also opens up interesting possibilities such as visual animation of the translated specifications [7]. The aim would be to translate ANISE descriptions into LOTOS and then animate them on-screen without the user having to know (much) LOTOS. It is also not hard to imagine how the time-sequence diagrams underlying ANISE might be brought to life graphically using the ideas of [7].

Although ANISE is mainly aimed at understanding the construction of services, a formal semantics would permit rigorous analysis through standard verification techniques. The ability to generate a formal description from ANISE could also be useful in other ways. For example, it could serve as a contract with developers, could be used as part of a formal method, and could be used to derive conformance tests in a rigorous way.

ANISE copes comfortably with the description of the basic call in section 4. The most pleasing results are in section 5, where a large number of modifications to the basic call are accommodated by almost one-line changes to the description. In a sense, section 5 extends the basic call with new features. However, these are rather small by telecommunications standards. Future work will produce descriptions of typical IN services.

An article of faith underlying the development of ANISE has been that it would ultimately support the detection of feature interactions. The principle has been to develop a rigorous, user-oriented, architectural method for describing services. It is believed that a sounder understanding of how to construct services will lead to a more structured approach, thus making the detection of interactions easier.

## Acknowledgements

The work reported in this paper was conducted during a short-term fellowship at the research laboratories of British Telecommunications PLC (BT). The author is grateful for the generous support of BT that made this possible. The author was also fortunate to have the constructive advice of BT staff during the work, particularly B. Strulo, R. Masson, M. J. Crowther, R. Kett and R. E. Booth.

## References

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

[2] ISO/IEC. *Information Processing Systems – Open Systems Interconnection – Conventions for the Definition of OSI Services*. ISO/IEC TR 10731. International Organization for Standardization, Geneva, Switzerland, 1992.

[3] ITU. *Intelligent Network – Q.120x Series Intelligent Network Recommendation Structure*. ITU-T Q.1200. International Telecommunications Union, Geneva, Switzerland, 1996.

[4] F. Joe Lin and Yow-Jian Lin. A building block approach to detecting and resolving feature interactions. In L. G. Bouma and Hugo Velthuijsen, editors, *Proc. Second International Workshop on Feature Interactions in Telecommunications Systems*, pages 86–119. IOS Press, Amsterdam, Netherlands, 1994.

[5] Kenneth J. Turner. An engineering approach to formal methods. In André A. S. Danthine, Guy Leduc, and Pierre Wolper, editors, *Proc. Protocol Specification, Testing and Verification XIII*, pages 357–380. North-Holland, Amsterdam, Netherlands, June 1993. Invited paper.

[6] Kenneth J. Turner. Specification architecture illustrated in a communications context. *Computer Networks and ISDN Systems*, March 1997.

[7] Kenneth J. Turner, Ashley McClenaghan, and Colin Chan. Specification and animation of reactive systems. In Volkan Atalay, Uğur Halici, Kemal Inan, Neşe Yalabik, and Adnan Yazici, editors, *Proc. International Symposium on Computer and Information Systems XI*, pages 355–364. Middle-East Technical University, Ankara, Turkey, November 1996. ISBN 975-429-103-9.

[8] Rob van der Linden. Using an architecture to help beat feature interaction. In L. G. Bouma and Hugo Velthuijsen, editors, *Proc. Second International Workshop on Feature Interactions in Telecommunications Systems*, pages 24–35. IOS Press, Amsterdam, Netherlands, 1994.