Kenneth J. Turner. An Architectural Description of Intelligent Network Features and Their Interactions (pre-publication version). Computer Networks and ISDN Systems, Special Issue on Feature Interaction, September 1998.

An Architectural Description of Intelligent Network Features and Their Interactions

Kenneth J. Turner

Department of Computing Science and Mathematics
University of Stirling, Stirling FK9 4LA, Scotland

Email: kjt@cs.stir.ac.uk

1st June 1999

Abstract

A brief explanation is given of the language ANISE (Architectural Notions In Service Engineering) that can be used to describe generic services as well as telecommunications services. The ANISE approach embodies a feature calculus that can be used to structure and analyse services. A description is given of the Plain Old Telephone Service using ANISE. This is extended for a number of typical features drawn from the Intelligent Network. It is shown how the description of features permits static and dynamic analysis of interactions. A partial semantics is given for ANISE using LOTOS (Language of Temporal Ordering Specification). This allows feature descriptions to be translated into LOTOS and analysed formally.

Keywords Feature Calculus, Feature Interaction, Formal Method, Intelligent Network Service, LOTOS (Language of Temporal Ordering Specification), Service Architecture, Service Feature, Telecommunications

1 Introduction

1.1 Modelling Services using ANISE

An architectural foundation for describing (service) features in telecommunications is given in [25]. This presents an architecture and a language for defining features in a rigorous, declarative and consistent manner. The approach deliberately does not give any special status to services, features, Service Independent Building Blocks or the basic call. All are considered to be behaviours – features – of a telecommunications system. Elementary behaviours are taken as the basis on which more complex behaviours can be built.

The goals of this work are:

- to gain a better understanding of telecommunications services, particularly in the sense of the IN (Intelligent Network, Q.1200 series of recommendations [16])
- to develop an abstract architecture for structuring telecommunications services in terms of their constituent features
- to treat services from a user point of view
- to offer a uniform service modelling approach
- to develop a formally-based language for describing telecommunications features and their combinations
- to gain an insight into feature interactions through static and dynamic analysis of service descriptions.

A service in the telecommunications sense is some set of network capabilities that is separately subscribed to and charged for. A service is thus a way of packaging and marketing a collection of features. However, this is a rather loose distinction since some features are services in their own right. The examples used in this paper are based on services in the IN. The IN recommendations to some extent blur the distinction between services and features.

The author's belief is that weaknesses in service architecture contribute towards the feature interaction problem. Of course this is not true of all feature interactions, but it is felt 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, an IN service). More specifically, an architecture is characterised by its components and how they are combined [28]. A service is used to mean the facilities offered to clients (here, telephone users).

The base document for this paper is [25] that introduces the language ANISE (Architectural Notions In Service Engineering). ANISE is used to describe features and their combinations in a declarative way. The present paper is complementary to [25] since it shows how to build higher-level features on top of an ANISE model for POTS (the Plain Old Telephone Service). The version of ANISE used here has evolved to allow description of IN-like services. To make the paper readable in isolation, a summary is given of ANISE and how it can be used to describe POTS.

1.2 Related Work

The work is rather different in style from most approaches to describing features and their interactions. The emphasis on constructing services from well-defined components using well-defined rules is reminiscent of the Advanced Network Systems Architecture [30]. The building block approach is also similar in spirit to the work of Lin and Lin [19], though in their case the building blocks are at a higher level.

The emphasis in ANISE is on structuring specifications of IN-like services. TINA (Telecommunications Intelligent Network Architecture [6]) has taken a somewhat different approach to defining telecommunications services, including concepts from ODP (Open Distributed Processing [13]) and modelling using OMT (Object Modelling Technique [20]). The TINA work is complementary to ANISE.

As will be seen, the semantics of ANISE is given by LOTOS (Language Of Temporal Ordering Specification [10]). There are therefore interesting comparisons with other work that uses LOTOS directly. Faci and co-workers have successfully used LOTOS in a constraint-oriented style to describe telephone systems and their features [7, 8]. In [9] they have shown how the feature interaction problem can be analysed in LOTOS. Thomas [22] also uses LOTOS, focussing like the present paper on the user interface to a telephone system, but detecting interactions by checking temporal properties against a LOTOS specification.

So why not write LOTOS specifications directly? It is the author's experience [21, 23, 27, 29] that LOTOS offers rather general features for specifying behaviour. In a particular application domain such as telecommunications services the specifier must 'bootstrap' him/herself to the architectural level at which systems should be described. Telecommunications services, for example, are expressed using concepts like dialling, call, diversion, clearing and busy. A direct specification has to express such concepts using only the constructs of LOTOS. The author believes that it is preferable to have such concepts available in the specification language, even if they are represented at a lower level in LOTOS. This allows the specifier to work at the level of the application domain, and results in smaller and more comprehensible descriptions. It also allows formal specifications to be produced (through translation from the service description language) without requiring the user to know LOTOS.

Several researchers specify the relationship between user and network actions using Message Sequence Charts (e.g. [3, 5]). These are reminiscent of the time-sequence diagrams that inspired the present paper. Among the approaches using Message Sequence Charts, that of Bergstra and Bouma [2] is similar in style to the work reported here although a different algebraic formalism is used.

1.3 Structure of the Paper

Section 2 summarises the ANISE approach to describing services. Since POTS is the foundation for all higher-level services, its description in ANISE is presented briefly in section 3. Features are broadly categorised as affecting an isolated call or as relating separate calls. Examples of both categories are described in sections 4 and 5, mainly using IN features. Section 6 discusses how ANISE helps to identify problem areas where feature interaction might occur. ANISE needs a formal meaning, and needs to be extensible for description of new kinds of services. Appendix A therefore outlines how ANISE constructs can be given meaning using LOTOS.

2 The ANISE Language

ANISE consists of constructs for defining elementary behaviours (features) and combinators for building these into more complex behaviours (also called features). It might thus be regarded as a feature calculus. Appendix A summarises the ANISE language in tables 2 and 3. [25] discusses the foundations of ANISE, in particular table 2, so the details are not repeated in the present paper. However, the constructs of ANISE will be illustrated through examples.

2.1 Modelling Approach

The ANISE approach is bottom-up, but from a user's perspective rather than an engineering viewpoint. Features and services are 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 signals. Following the terminology of OSI (Open Systems Interconnection [12]), signals are called service primitives. Since higher-level features are built from lower-level ones in a uniform way, all behaviour is just a feature in ANISE.

The major advantage of the approach is architectural consistency since a uniform notation and set of concepts is used. The elements of a service have the same status and are described in the same kind of way. There is no special treatment, for example, of the basic call or of services vs. features. Because the approach is compositional, there is a rigorous foundation on which higher-level features 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. (Note that the paper uses 'interaction' in the general sense of exchanging information, and also in the sense of features interfering with each other.)

The generic constructs in ANISE are useful for describing a wide range of services such as those in OSI, the IN or ODP. OSI services tend to be very regular in nature and need little more descriptive power than basic ANISE. IN services tend to be much more individual. The result is that ANISE needs to be extended with specific constructs to reflect particular services. Experience so far with ANISE suggests that a new service can normally be described at the call level with the generic ANISE constructs. However, a special-purpose combinator is often needed to reflect the service's functionality at a global level. Section 5.2 will illustrate this observation for Call Waiting. The price that is paid for specification convenience is that the semantics of special-purpose combinators has to be defined (through their translation to LOTOS).

Features may be invoked in an isolated fashion, but usually have some relationship to each other. For example, a normal telephone call requires ordered execution of call establishment, speech and clearing. The general notion of an interaction group is introduced for a collection of interactions that should be considered as related. Such a group is known to each user by a local reference called an interaction group identifier (*Id* for short). Connections such as telephone calls are interaction groups, and interaction group identifiers indicate the telephone lines that are connected. A set of *Id*s (one for each user in a call) is associated with an interaction group. Interaction group identifiers are part of the modelling approach, and are not explicitly written in an ANISE description.

As an example, a call starts out with undefined identifiers for each end. The act of seizing the line fixes the identifier of the calling end, while dialling fixes the identifier at the called end. Both identifiers are now known and are used during the remainder of the call. When the call is cleared, the identifiers are no longer needed and can be re-bound in a new call.

2.2 Feature Characteristics

Features 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 may 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). Features are therefore specified unidirectionally; a symmetrical service simply allows a feature to be invoked by either user.

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 the patterns of interaction among users; interactions correspond to the occurrence of OSI-like service primitives. OSI defines four roles for primitives:

req(uest): a service user asks the service provider to carry out some function

ind(ication): the service provider notifies a service user, usually in response to a request; the service provider may also give an indication autonomously *res(ponse):* a responding service user acknowledges receipt of an indication

con(firm): the service provider confirms execution of a request back to the originating service user

A study of typical telecommunications services reveals that there are a number of basic patterns in the behaviour of features, illustrated in figure 1 using the OSI diagrammatic notation for services [11]. The actions of users and the service provider are shown in columns, with time running down the page. User 1 is conventionally on the left of these diagrams, and User 2 is on the right. 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). Some possible telephony examples of the patterns in figure 1 are as follows:

user_initiated: a user clears a call by going on-hook (request)

provider_initiated: the network rings the called party (indication)

unconfirmed: speech by one party (request) is delivered to the other (indication)

local_confirmed: a caller goes off-hook (request) and obtains dialling tone (confirm)

remote_confirmed: the network asks the caller to hang up (indication) who does so (response)

asymmetric_confirmed: one party hangs up (request), causing the other to be notified of disconnection (indication) and to hang up as well (response)

provider_confirmed: a number is dialled (request), causing ringing at the called party (indication) and at the caller (confirm)

user_confirmed: a user asks for the cost of the last call (request), and this is forwarded to the billing centre (indication); the cost is announced by the billing centre (response) and is notified to the user (confirm)

user_provider_confirmed: a credit card number is supplied (request) and is notified to the billing centre (indication); in either order, the network is told that the charge will be honoured (response) and the user is told that the card will be charged (confirm)

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). Possible telephony examples of these properties are as follows:

single: seizing a line before dialling

consecutive: a call is dealt with before another is permitted

ordered: voice samples are delivered in the correct order even if their end-to-end transmission is overlapped

reliable: nearly simultaneous calls to a PABX (Private Automatic Branch Exchange) may be connected in any order

unreliable: call requests may be lost due to problems in the service provider such as congestion

Patterns and properties are essential in defining the elements of call behaviour. Examples of their use will appear in figures 5 and 7. The semantics of patterns and properties are illustrated with some sample translations to LOTOS in appendix A.

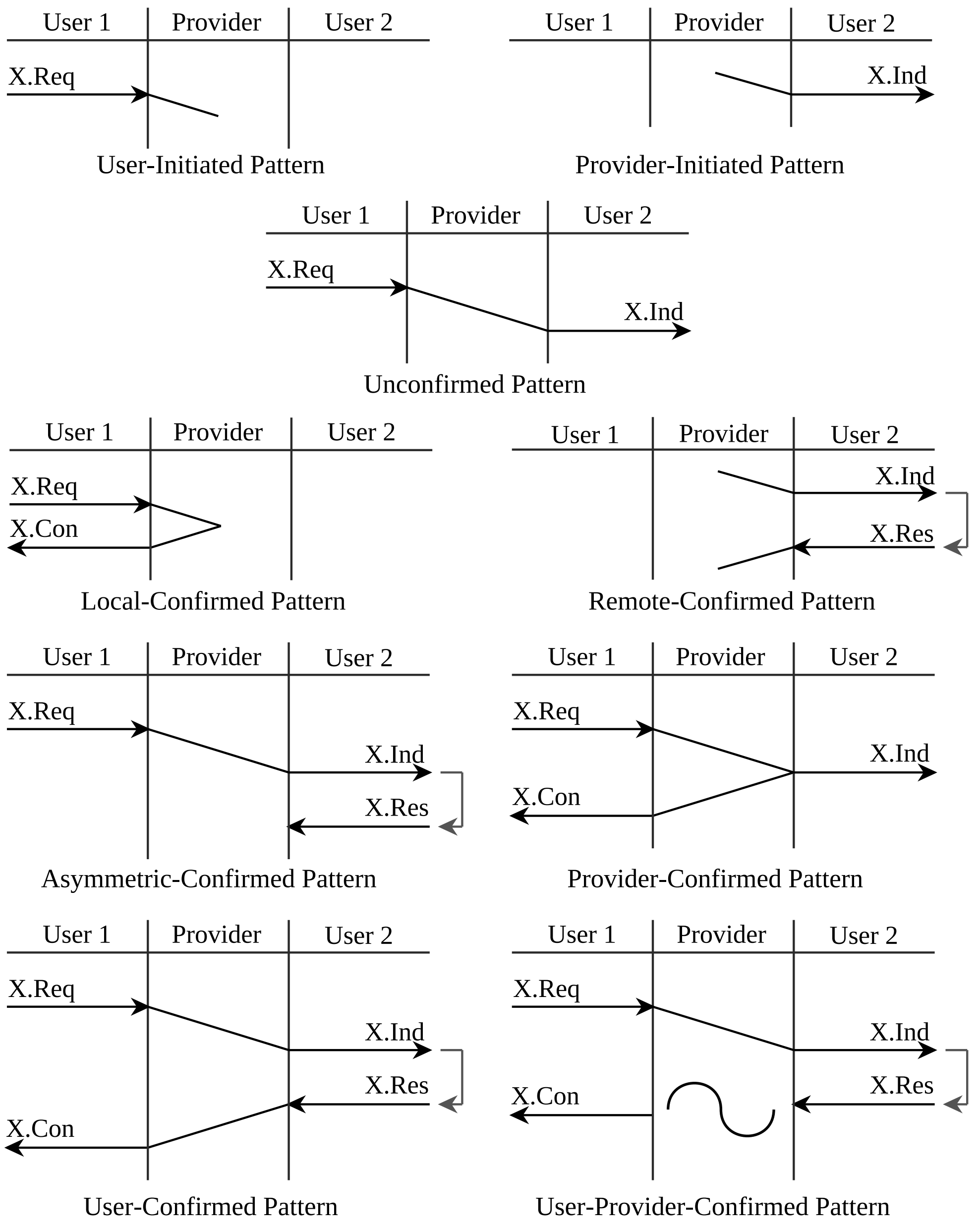

Figure 1: Feature Patterns

$$\begin{tikzcd}
 & \text{Property} \arrow[dl] \arrow[dr] & & & & \\
\text{Single} & & \text{Multiple} \arrow[dl] \arrow[dr] & & & \\
 & \text{Consecutive} & & \text{Overlapped} \arrow[dl] \arrow[dr] & & \\
 & & \text{Ordered} & & \text{Unordered} \arrow[dl] \arrow[dr] & \\
 & & & \text{Reliable} & & \text{Unreliable}
\end{tikzcd}$$

Figure 2: Feature Properties

2.3 Feature Combinations

Features and their combinations are declared using the language of ANISE. 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 implied by the parameters. The direction is 12 or 21, depending on which user initiates the feature (logical endpoint 1 or 2). The pattern is one of those given in figure 1, and the property is one of the leaf nodes in figure 2. The group name is usually the same as that of the service primitives involved (e.g. group *OffHook* for primitives *OffHook.Req/Ind*). In the case of a confirmed pattern, two groups may be given. This allows for the possibility that the group names differ. For example call establishment might be considered to have separate calling and answer behaviour, and thus to have two group names: *Call* and *Answer*. A group may also cite an explicit parameter type in parentheses, e.g. *Call(Num)*. If both group names are the same but their parameters differ then only the second parameter may be given, e.g. *Call(Num),(CallingMess)*. In this case the primitives might be *Dial.Req(Num)* and *Dial.Con(CallingMess)*.

The following basic data types are pre-defined in ANISE. Each type has named constant as undefined value (*NoDig*, *NoNum*, etc.). In addition a number of data types are used internally such as *Id* (a line identifier) and *Result* (generated by a feature). If special-purpose data types are needed to describe a service, they must be incorporated in the translation to LOTOS.

CalledMess: issued to the called party while a call is being established or is in progress (e.g. ringing current, calling number)

CallingMess: received by the calling party while a call is being established or is in progress (e.g. ringing tone, subscriber busy)

Dig: a 'digit' (including * and #)

Num: a sequence of digits such as a telephone number

Voice: a segment of speech

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 elementary features or their combinations:

> *Combinator*(*Parameter_1, ... ,Parameter_N*)

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. A single expression for a high-level feature might be unwieldy, or might require repetition of sub-expressions. In such a case a part of the overall behaviour may be named by:

> **declare**(*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.

2.4 Feature Parameters

The behaviour of a telecommunications feature is often dependent on characteristics chosen by the subscriber. For example, the subscriber might set up abbreviated dialling or might request call diversion to another number. This collection of characteristics is stored in a telephone number's profile – a record-like set of values. The profiles are globally available to all features in case their behaviour depends on it.

Since a feature or combinator relates endpoints, its behaviour is parameterised by their numbers. However, a feature may establish or change a number as it executes. For example, the calling number is set by the feature that seizes the line, and the called number is established when the dial feature determines the destination line. The called number may differ from the dialled number due to the use of features like call forwarding. A feature therefore exports the numbers in use when it terminates; the numbers may be used by features that execute later.

Features are often behaviourally related through result values. For example, unless seizing the line yields dialling tone it will not be possible to dial a number. Similarly, dialling a number must result in ringing tone before the call can be answered. The behaviour of a feature may just stop; this is regarded as unsuccessful termination and produces no result. If a behaviour terminates successfully it is considered to produce a result – the parameter value of the confirm service primitive (if any). Apart from terminating unsuccessfully or successfully, a behaviour may also recurse (i.e. loop). Clearly no result is possible in this case.

Summarising the points above, the functionality of a terminating feature is considered to be:

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

where the pair of numbers refers to the local/remote ends, and *Result* is some type like a called or calling message. The numbers of a feature instance start out undefined (*NoNum*). When a feature instance first interacts at an endpoint it accepts any number. In future only that number is permitted at the endpoint, and the numbers in use are exported on termination.

To make this idea concrete, consider seizing the line by picking up the telephone. Initially the number of neither endpoint is known, but the act of seizing the line determines the caller's number. In addition, seizing normally yields dialling tone as result. In this case the functionality of the *Seize* feature is:

$$\text{NoNum} \times \text{NoNum} \mapsto \textit{SomeNum} \times \text{NoNum} \times \text{DialTone}$$

Normally a feature executes in the direction specified for it; for example, feature *Feat* might be defined for direction 12. It is sometimes desirable to override this by specifying different endpoints; for example, *Feat(21)* invokes the feature in the reverse direction. It is essential to specify endpoints in this way for features that involve more than one call such as Three-Way Calling; for example, *Feat(32)* invokes the feature between logical endpoints 3 and 2. Endpoint 0 corresponds to no number; for example, *Feat(00)* starts with neither calling nor called number known. Endpoint S means a surrogate user (as used with Call Waiting and Three-Way Calling in sections 5.2 and 5.3).

3 Call Description

This section uses ANISE to describe POTS (the Plain Old Telephone Service). A fuller explanation appears in [25], so a fairly brief exposition is given here. Note that the basic call is *defined* in ANISE and is not intrinsic to it. The description gives an architecture for the basic call in the sense of defining its component behaviours and how these are combined. Since the ANISE description is linear, figure 3 may be helpful in understanding the overall structure. The innermost ovals in the figure show the features. These are progressively combined into more complex behaviours using the combinators shown. The basic call behaviour is a repeated sequence of the three phases.

3.1 Basic Call Features

The elementary features identified for a basic call are shown in figure 4. This is unadorned with details of features such as their parameters, properties and relationships; the ANISE declarations in the first part of figure 5 make these explicit.

Seize: This feature is invoked by the caller (i.e. in the 1→2 direction), and the pattern has a single instance. *OffHook.Req* corresponds to picking up the telephone. *OffHook.Con* is the response, with parameter *CallingMess* normally indicating dialling tone. However, other responses like unobtainable are possible.

Dial: *Dial.Req* conveys the called number *Num*. The response is *Dial.Con*, hopefully carrying ringing tone as the value of parameter *CallingMess*. However other possibilities include subscriber busy and unobtainable.

Ring: *StartRing.Ind* is issued when the called party is rung.

Silence: *StopRing.Ind* is issued to stop the called party ringing.

Answer: This feature operates from called to caller (i.e. 2→1). *Answer.Req* corresponds to answering the call. *Answer.Ind* is an explicit signal that the called party has answered, though in current telephony the caller is not directly aware of this.

Speak: 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). Speech allows overlapped voice transmissions in the same direction that are kept in the correct relative order. *Speech.Req* and *Speech.Ind* both carry speech segments of type *Voice*.

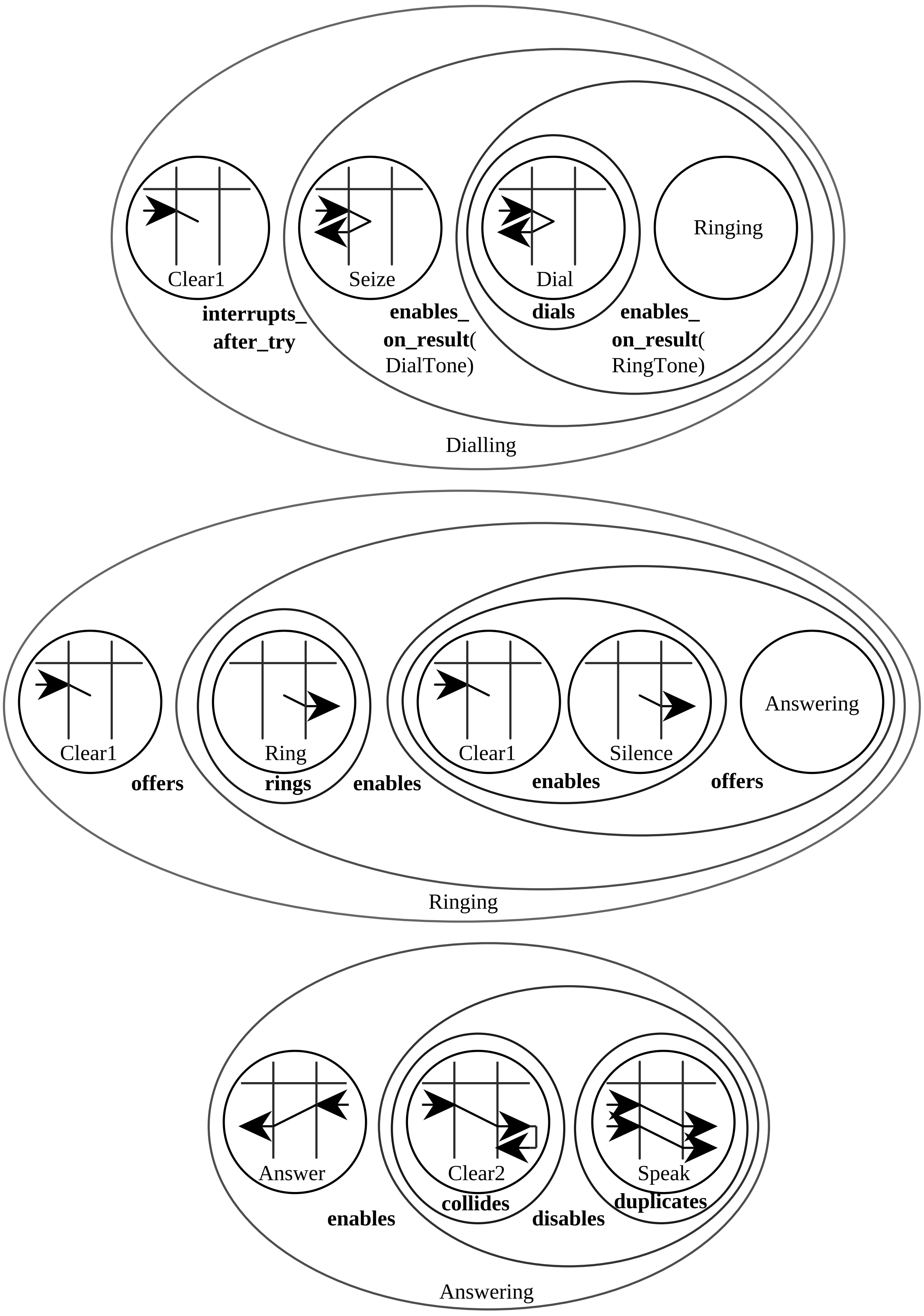

Figure 3: Structure of the Basic Call Description in ANISE

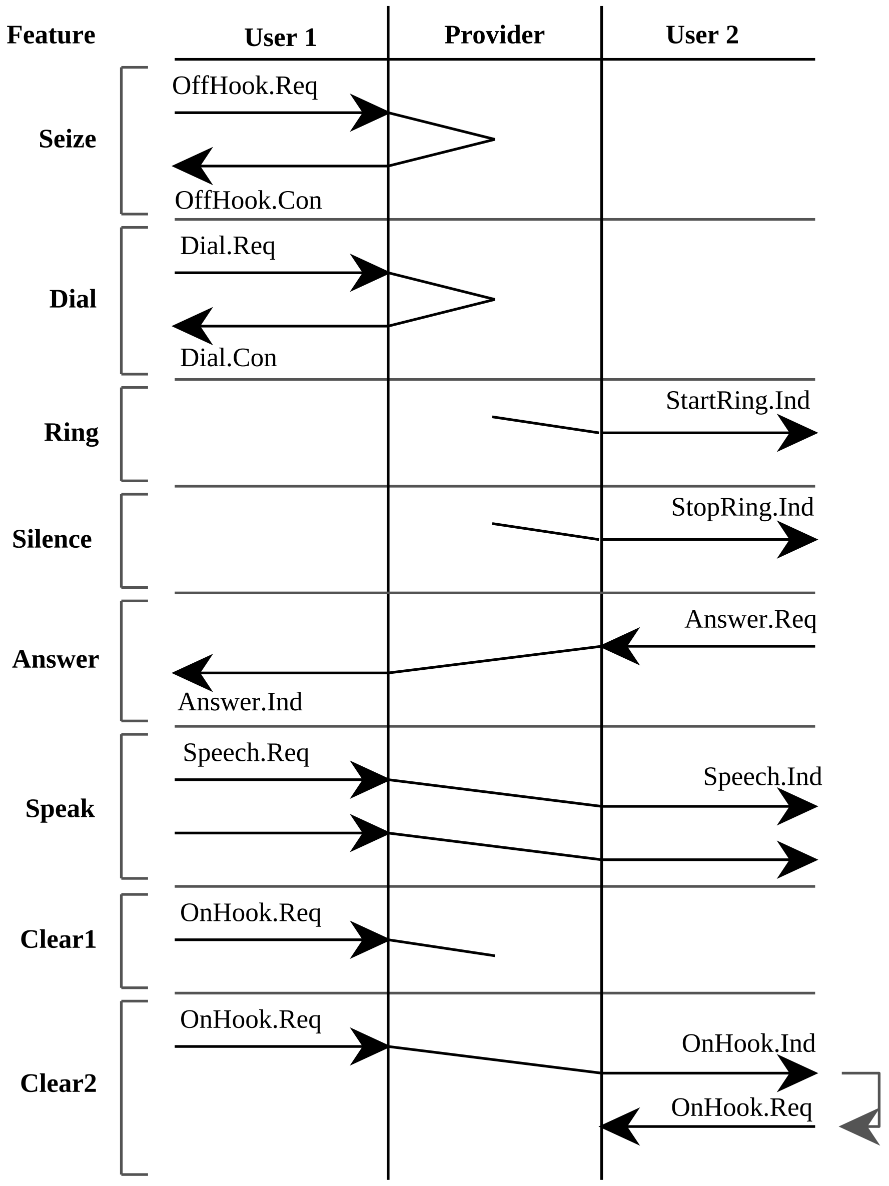

Figure 4: Elementary Features of the Basic Call

```
% Basic Call Elementary Features:

declare(Seize,feature(12,local_confirmed,single,OffHook,(CallingMess)))       % seize the line
declare(Dial,feature(12,local_confirmed,single,Dial(Num),(CallingMess)))      % dial number and connect
declare(Ring,feature(12,provider_initiated,single,StartRing))                 % start ringing number
declare(Silence,feature(12,provider_initiated,single,StopRing))               % stop ringing number
declare(Answer,feature(21,unconfirmed,single,Answer))                         % answer call
declare(Speak,feature(12,unconfirmed,ordered,Speech(Voice)))                  % one-way speech
declare(Clear1,feature(12,user_initiated,single,OnHook))                      % clear by caller
declare(Clear2,feature(12,asymmetric_confirmed,single,OnHook,OnHook))         % clear by caller or called

% Basic Call Behaviour:

declare(Dialling,                                                             % dial phase
  interrupts_after_try(                                                       % clear interupts seize, dial
    Clear1,                                                                   % caller clear
    enables_on_result(                                                        % seize, dial if dial tone
      Seize(00),                                                              % seize line in fresh call
      DialTone,                                                               % if seize got dial tone then ...
        enables_on_result(                                                    % dial, exit if ring tone
          dials(Dial),                                                        % dial selects destination
          RingTone,exit))))                                                   % if ring tone, exit dial phase

declare(Ringing,                                                              % ring phase
  offers(                                                                     % clear or ringing
    Clear1,                                                                   % caller clear
    enables(                                                                  % ring, clear or answer
      rings(Ring),                                                            % ring destination appropriately
      offers(                                                                 % clear or answering
        enables(Clear1,Silence),                                              % caller clear, stop ringing
        Answering))))                                                         % answer, speak

declare(Answering,                                                            % answer phase
  enables(                                                                    % answer, stop ring, speak, clear
    Answer,                                                                   % answer call
    disables(                                                                 % clear stops speech
      collides(Clear2),                                                       % caller/called clear
      duplicates(Speak))))                                                    % speech in both directions

declare(Call,                                                                 % overall basic call
  loops(                                                                      % loops indefinitely doing ...
    enables(Dialling,Ringing)))                                               % dial phase, ring/answer phase

% Global (POTS) Behaviour:

profile(124,53) ...                                                           % number 124/id 53, etc.

declare(MaxCalls,100)                                                         % number of concurrent calls

global(tel,                                                                   % name for telephone service
  checks_busy(                                                                % monitor line free/busy
    instances(MaxCalls,Call)))                                                % instantiate calls
```

Figure 5: ANISE Declarations for the Plain Old Telephone Service

Clear1: This is one-sided clearing, where the caller breaks the call with *OnHook.Req* before the other party has been rung.

Clear2: This is two-sided clearing, where either party can break the call by hanging up (i.e. doing *OnHook.Req*). *OnHook.Ind* signals the other party that the call has been broken, though in current telephony there may be no explicit indication of this. The call is fully cleared only when the second party hangs up, i.e. *OnHook.Req* occurs.

3.2 Basic Call Behaviour

The relationships between basic call features – their combinations – are shown in the second part of figure 5. Although a call could be defined by a single ANISE declaration, the description is broken down into smaller pieces for convenience. The features and combinators used in the description are mentioned below.

Dialling: This is the initial behaviour of the call (seizing the line, dialling). A one-sided clear by the caller (*Clear1*) is permitted once an attempt to make a call has begun (**interrupts_after_try**). If a call is interrupted by this clear, its initial behaviour repeats; this is the 'inner loop' of the call. Seizing the line (*Seize*) permits dialling only if dial tone is received (**enables_on_result**). Any other result such as unobtainable prevents dialling. Receipt of ringing tone permits the dialling phase to finish (*Dial* obtains *RingTone*). Any other result like subscriber busy prevents further progress on the call. The **dials** combinator ensures that the called telephone is the one determined by the dialled number (*Dial*).

Ringing: The ringing phase is surprisingly complex because the caller may hang up at a number of points before answering occurs. The caller may clear before the called party is actually rung (*Clear1*). Normally, the called party is rung (*Ring*); the **rings** combinator chooses the appropriate ring pattern. At this point the caller may hang up, causing ringing to stop (*Clear1, Silence*). Hopefully, however, the call will progress to the answering phase (*Answering*).

Answering: This is the main behaviour of the call (answering and speaking followed by clearing). When the call is answered (*Answer*), speech is enabled. The **duplicates** combinator interleaves independent copies of the *Speak* feature in each direction. Since interruption through clearing is immediately possible, there may be no speech before the call is cleared. Speech continues indefinitely, so only clearing can stop this. The **collides** combinator allows either party to initiate clearing (*Clear2*), and handles the case of simultaneous clearing. The whole behaviour of *Answering* terminates when clearing has finished.

Call: This defines the behaviour of calls between a single, unspecified pair of users. Once clearing has been completed, a new call can begin. If ringing does not occur because the call cannot be put through, *Clear1* in *Dialling* will occur (i.e. the caller will hang up). Ringing tone permits the ringing phase to start (*Ringing*). The 'outer loop' in the definition (**loops**) deals with repetition of the whole call when it clears.

3.3 Plain Old Telephone Service

At the global level, the third part of figure 5 shows how basic calls are assembled into the standard service between any pair of users.

profile: For POTS, the only profile information needed for each number is its associated line identifier. A number/identifier pair is defined by each call of **profile**.

global: The highest level in the ANISE description names the telephone service as *t*. The **checks_busy** combinator keeps track of which lines are busy and which are free. In particular it ensures that calling an engaged number results in the caller getting subscriber busy. The **instances** combinator creates the required number of independent basic call instances.

3.4 Call Feature Classification

Features have static and dynamic aspects. The static aspects reflect the functionality and parameterisation of a feature. Generally speaking, if a feature is present it is potentially available to all users. Subscription to a feature is regarded as static in ANISE, that is the parameters required to invoke a feature are included in a description –

specifically, in the profiles. In practice the subscriber may supply feature parameters to the network operator just once on subscription or may request changes infrequently. It may be possible for the user to change feature parameters dynamically (e.g. to set up call diversion). ANISE could in principle allow such changes (including cancellation of a subscription). However, management of subscriber profiles is a separate issue and is not currently considered in ANISE.

The dynamic aspects of a feature reflect its invocation and execution. A feature may access the profile for the corresponding number to decide whether and how it should execute. The existence of the profile is largely hidden from the ANISE specifier, i.e. its use is implicit in the combinators. Use of the profile is actually defined in the semantics for ANISE. The addition of a new feature may require the profile to be extended. See the treatment of abbreviated dialling in appendix A.2 as an example of the how the profile is handled 'behind the scenes'.

Features can be categorised as single-call or multiple-call. Single-call features apply to a call in isolation, e.g. Abbreviated Dialling and Call Screening. Such features modify the behaviour of the basic call. In ANISE terms, they appear as changes to the first or second parts of figure 5 (i.e. the elementary features or basic call behaviour). Multiple-call features relate the behaviour of several calls, e.g. Call Waiting and Three-Way Calling. Such features principally operate at a higher level than the basic call, so they mainly change the third part of figure 5 (i.e. the global behaviour). Single-call and multiple-call features are treated separately in sections 4 and 5. The naming of features varies among network operators, so the terms and abbreviations in Capability Sets 1 and 2 [15, 17] are used where possible. However alternative names are mentioned where known to the author; some of these are trademarked by their owners.

In the following two sections, a brief description of each feature is given and then a discussion of how it is modelled in ANISE. The alterations caused by a feature are relative to the POTS description in figure 5. Unlike the generic combinators in table 2, the telephony combinators in table 3 assume the naming conventions for primitives in figure 4. This is not a necessary restriction, however, since the names could in principle be supplied as parameters to the combinators.

4 Single-Call Features

4.1 Abbreviated Dialling (ABD)

Description: Abbreviated Dialling, also named (Short) Code Dialling, allows the subscriber to define short dialling sequences (e.g. **23) that expand to full telephone numbers.

ANISE: The profile may contain a set of short code/full number pairs. The **dials_code** combinator translates the short code given to a behaviour into the full number. It modifies dialling in the basic call:

dials_code(Dial)

4.2 One Number (ONE)

Description: One Number is used in various services to allow the called party to have a single number that may be called from anywhere. Commercial examples include free calls (FreePhone) and national calls at a local rate. Other uses for One Number include calls to emergency services, the telephone operator and network customer services. This feature resembles Abbreviated Dialling, but the short codes are defined by the *called* party on a network-wide basis. An important difference is that One Number need not be a simple 1:1 translation. For example, a call to the single number may be routed to a company's nearest branch. A call plan may even carry out the translation based on other factors such as the time of day.

ANISE: The required translation is similar to that of the **dials** combinator except that it may be more than a static look-up. By default, ANISE supports only a simple look-up of the profile; an algorithmic number translation would have to be built into the LOTOS definition. A call plan that depended on time of day would present a problem since standard LOTOS does not specify metric time. A clock process could be introduced (though this would not model the passage of real time). A better solution would be to use E-LOTOS (Enhancements to LOTOS [14]) that does allow timed specification. The **dials_one** combinator modifies dialling in the basic call:

dials_one(Dial)

4.3 Originating Call Screening (OCS)

Description: Originating Call Screening, also named Outgoing Call Barring, prevents calls to certain telephone numbers or area codes. Parents, for example, might wish to stop their children from calling premium rate numbers. In the extreme case, all outgoing calls (except perhaps to the emergency services) may be disabled; the network operator might do this if a telephone bill remains unpaid.

ANISE: The profile may contain a set of called numbers to be forbidden, or the value *All*. A call to any such number is not allowed to proceed beyond dialling. The **screens_out** combinator prevents behaviour that would otherwise call a banned number. It modifies dialling in the basic call:

screens_ out(Dial)

4.4 Terminating Call Screening (TCS)

Description: Terminating Call Screening, also named Incoming Call Barring, prevents the reception of calls from certain telephone numbers or area codes. In the extreme case, all incoming calls may be disabled; for example, a call-box may not be allowed to receive calls.

ANISE: The profile may contain a set of calling numbers to be forbidden, or the value *All*. A call from any such number is not allowed to proceed to ringing. The **screens_ in** combinator prevents behaviour that would otherwise accept a call from a banned number. It modifies ringing behaviour in the basic call:

screens_ in(Ring)

4.5 Calling Number Delivery (CND)

Description: Calling Number Delivery, also named Caller Display, requires a special telephone that can show the caller's number before a call is answered. Based on the calling number, the user might decide not to answer when the telephone rings.

ANISE: The **rings_display** combinator sets the *CalledMess* parameter in a behaviour to the caller's number. It modifies ringing behaviour in the basic call:

rings_display(Ring)

4.6 Distinctive Ring (DR)

Description: Distinctive Ring, also named Multiple Directory Number Line with Distinctive Ringing, does not yet seem to have been introduced into the IN Capability Sets. It allows the subscriber to have multiple numbers associated with one telephone. For example a family might choose a number and ring pattern for calls to parents, and another for calls to children. A separate number and ring pattern might also be used for calls to a fax machine that shares the same line as the telephone.

ANISE: There are several telephone numbers for the same line identifier, each associated with a particular ring pattern in its profile. The **rings_preference** combinator selects the *CalledMess* parameter in a behaviour according to the ring preference in the profile. It modifies ringing behaviour in the basic call:

rings_preference(Ring)

4.7 Call Forwarding Unconditional/Don’t Answer (CFU/CFDA)

Description: Call Forwarding has a variety of forms and names such as Call Diversion, Call Forwarding on Busy Line, Call Forwarding Don’t Answer, Call Rerouting Distribution and Follow-Me Diversion. Their common characteristic is that the user may decide to have incoming calls diverted to an alternative telephone. At the user’s discretion calls may be diverted unconditionally, if the number is busy, or if there is no answer after some period. All forms of call forwarding essentially perform a translation, changing the called number into the actual destination line. Call Forwarding Unconditional and Call Forwarding Don’t Answer are considered here since they affect just an isolated call. Call Forwarding on Busy Line appears in section 5.1.

Answer Call, also named Call Minder, has not yet been fully introduced into the IN Capability Sets. It resembles Call Forwarding in that a call to a busy line or an unanswered call is diverted to an answering service. This will record a message that the user can later retrieve.

ANISE: Call diversion information may appear in the profile. Various **diverts** combinators modify the called *Id* in a behaviour according to the selected diversion condition. Although Call Forwarding is often offered as a single service to the user, ANISE correctly treats the various forms of Call Forwarding separately. Call Forwarding Unconditional/Don’t Answer uses the **diverts_always**/**unanswered** combinators that modify dialling/ringing in the basic call:

diverts_always(Dial)
diverts_unanswered(Ring)

Diversion on no answer requires metric time so E-LOTOS may be a better choice for the semantics. Without time, ANISE can only non-deterministically decide whether to continue ringing the telephone or to divert the call.

At the level of abstraction chosen for ANISE, Answer Call looks just like Call Forwarding on Busy Line/Don’t Answer. ANISE treats the answering number just like another subscriber. Only a **diverts** combinator would therefore be used.

5 Multiple-Call Features

5.1 Call Forwarding on Busy Line (CFBL)

Description: Single-call forms of Call Forwarding appeared in section 4.7. Call Forwarding on Busy Line needs to be treated as a multi-call feature because it relies on knowledge of whether the destination line is busy.

ANISE: Coordination at the multiple call level is achieved by the **diverts_busy** combinator that surrounds **checks_busy** at the global level:

diverts_busy(**checks_busy**(**instances**(MaxCalls,Call)))

If the called number is busy, the forwarding number (if any) in the profile is used to determine the new destination number.

5.2 Call Waiting (CW)

Description: Since realisations of Call Waiting may vary, a typical service will be described. Normally a call to a busy line results in engaged tone for the caller. However if the called party has subscribed to Call Waiting, the caller can be held. The called party is alerted by a special tone to say that an incoming call is waiting. Call waiting can be controlled by pressing Recall (flash-hook) and then a digit. The called party can reject waiting call(s) with Recall + 0. Hanging up clears the current call, causing the waiting call to be connected; the same result can also be achieved by Recall + 1. It is possible to switch between the current and waiting calls by using Recall + 2.

ANISE: Call Waiting modifies the endpoints of the basic call. Since the callers cannot both be connected to the called party, connections are made to a surrogate user that manages waiting calls. This is a modelling concept that may not correspond to a real network function. The surrogate deals with requests by the called party to manage Call Waiting. The transitions between different call waiting situations are shown figure 6.

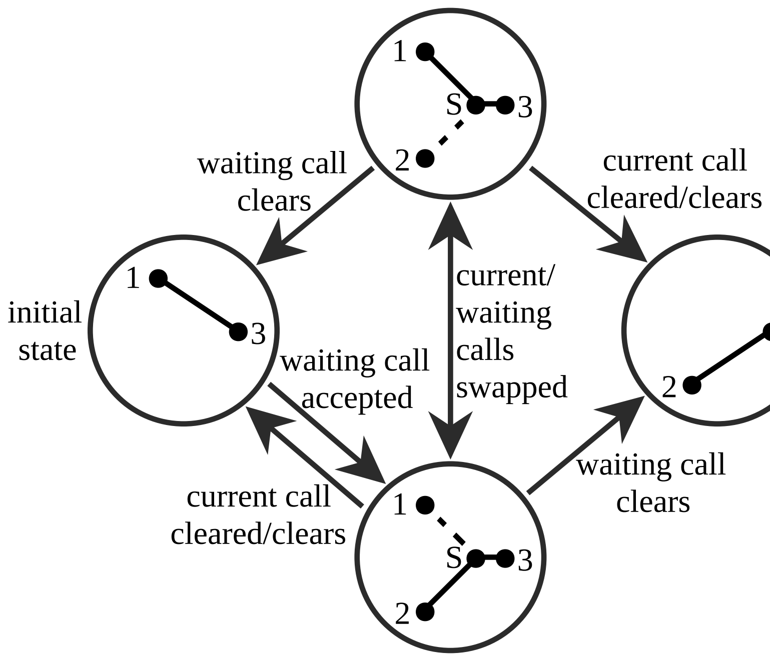
1
S
3
2
waiting call clears
current call cleared/clears
initial state
1
3
current/ waiting calls swapped
3
2
waiting call accepted
current call cleared/clears
waiting call clears
1
S
3
2

Figure 6: Transitions between Call Waiting Situations

Here, user 1 is the original caller, user 2 is the second caller, user 3 is the called party, and 'user' S is the surrogate.

ANISE models the current or waiting call very much like the normal basic call. However, feature definitions have to be introduced for recall (*Recall* or Flash-Hook), dialling a digit (*Select*), and responding to an exchange ring (*Respond*). *Recall.Req* suspends an active call, or resumes a suspended call. *Recall* could be declared as **unconfirmed** to model a network announcement with *Recall.Ind* that the call has been suspended or resumed. *Select.Req* chooses a call waiting action according to the dialled digit. The *Ring* feature of the basic call is extended to allow call waiting tone as a value of *CalledMess*.

The surrogate deals with call waiting signals, hence they do not appear directly in the basic call. However, answering and speech in the basic call of figure 5 have to be put under call waiting control. The **holds** combinator handles the effect of suspension and resumption, including the possibility of either party being put on hold:

```
  holds(Answering)                                  % answer/speech may be held
```

From the point of view of the waiting caller, the destination is rung and may answer. Interactions with the surrogate re-use many features of the basic call. The behaviour of the call waiting surrogate is shown in the second part of figure 7. It is rather striking that the description of Call Waiting is more complicated than that of the basic call! This reflects the inherent complexity of Call Waiting rather than a lack of expressiveness in ANISE. Even so, figure 7 defines only user control of Call Waiting; global coordination is embedded in the **waits_call** combinator described below.

The initial behaviour of call waiting is different for two reasons: it is possible to cancel the waiting call and any future ones; and the waiting call must be answered the first time, not just taken off hold. If the waiting call is accepted, future behaviour is that of the first or second caller waiting. Call Waiting involves three logical endpoints: the first or current caller (1), the second or waiting caller (2), and the called party (3). Features must be invoked with an explicit pair of endpoints so as to identify the relevant call.

CallPending: This behaviour applies after call waiting has begun and one party is waiting. The current caller or the called party may hang up, causing the waiting call to be resumed; after this there is just

% Actual Waiting: 1 Current Caller, 2 Waiting Caller, 3 Called, S Surrogate

```
declare(CallWaiting,                                  % actual call waiting
  disables(                                           % clear finishes call waiting
    offers(                                           % choice of clears
      Clear2(1S),                                     % current caller clears to surrogate
      Clear2(2S),                                     % waiting caller clears to surrogate
      enables(                                        % called clears, picks up
        Clear1(3),                                    % called hangs up
        collides(Clear2(S1)),                         % surrogate clears current caller
        rings(Ring(S3)),                              % surrogate rings called again
        Respond(3)))),                                % called responds to ringing
    enables(                                          % recall, select
      Recall(3),                                      % suspend current caller
      offers(                                         % select or recall
        enables_on_result(                            % select decides action
          Select(3),                                  % dial digit
            1,collides(Clear2(S1)),                   % if 1, clear current caller
            2,CallWaiting(21),                        % if 2, swap current/waiting
            CallWaiting),                             % else same current/waiting
        enables(                                      % recall
          Recall(3),                                  % resume current caller
          CallWaiting)))))                            % same current/waiting callers
```

% Potential Waiting: 1 First Caller, 2 Second Caller, 3 Called, S Surrogate

```
declare(CallPending,                                  % potential call waiting
  disables(                                           % clear finishes call waiting
    offers(                                           % choice of clears
      enables(                                        % first clears, second answered
        Clear2(1S),                                   % first caller clears to surrogate
        Answer(S2)),                                  % surrogate answers second caller
      enables(                                        % second clears, stops wait tone
        Clear1(2),                                    % second caller hangs up
        Silence(3))),                                 % call wait tone stops at caller
      enables(                                        % called clears, second answered
        Clear1(3),                                    % called hangs up
        collides(Clear2(S1)),                         % surrogate clears first caller
        rings(Ring(S3)),                              % surrogate rings caller again
        Answer(32))),                                 % called answers second caller
    enables(                                          % recall, select
      Recall(3),                                      % suspend first caller
      offers(                                         % select or recall
        enables_on_result(                            % select decides action
          Select(3),                                  % dial digit
            0,collides(Clear2(S2)),                   % if 0, clear second caller
            1,enables(collides(Clear2(S1)),Answer(S2)),  % if 1, clear first, answer second
            2,enables(Answer(S2),CallWaiting(21)),    % if 2, answer second, swap first
            CallPending),                             % else second caller still waiting
        enables(                                      % recall
          Recall(3),                                  % resume first caller
          CallPending))))                             % second caller still waiting
```

Figure 7: ANISE Coordination of Call Waiting

a normal call. Alternatively the called party may use recall and dial a digit to control waiting. The current call may be cleared and replaced by the waiting call (1 dialled). The current call may also be swapped with the current call (2 dialled). If any other digit is dialled or the called party presses recall again, the current call is resumed. The **rings** and **collides** combinators are used as in the basic call to control ringing and clearing.

CallPending: This is the initial behaviour when a potential waiting call arrives. It differs from the later behaviour of *CallWaiting* in a few ways. The called party may reject the waiting call by selecting 0. If the waiting call is accepted, it must first go through the answering procedure. Thereafter, call waiting is as described by *CallWaiting*.

Coordination at the multiple call level is achieved by the **waits_call** combinator that surrounds the global call instances:

waits_call(CallPending,**instances**(MaxCalls,Call))

If the called number is free, it is rung in the ordinary way. If the called number is busy and does not have Call Waiting, the caller receives subscriber busy and the call progresses no further. However if the called number is busy and is allowing Call Waiting, the calls are connected to an instance of *CallPending* as surrogate. The surrogate terminates when it is no longer needed because only one call remains.

5.3 Three-Way Calling (TWC)

Description: Three-Way Calling as such does not currently appear in the IN Capability Sets. Instead the IN refers to Conference Calling (CONF) which appears to allow any number of parties. For concreteness a typical Three-Way Calling service will be described. One party in a normal call can add a third party to the call. Three-Way Calling can be controlled by pressing Recall (flash-hook) and then a digit or full telephone number. The controlling caller dials the third party with Recall + Number. The controlling caller can clear the original call but retain the new call with Recall + 1. It is possible to switch between the original and new calls by using Recall + 2. A three-way call is set up with Recall + 3. After a three-way call is established, the original caller can be disconnected by using Recall + 5. Alternatively, the third party can be disconnected by using Recall + 7. Hanging up a three-way call clears both call legs. Three-Way Calling shares some aspects with Call Waiting, e.g. the ability to switch between calls.

ANISE: Three-Way Calling modifies the behaviour of the basic call. Establishing the second leg of the call is very much like a normal call, except that a surrogate is involved. Since the three parties are not initially all connected, the caller on hold is connected to a surrogate much as for Call Waiting. The surrogate deals with requests by the controlling party to manage Three-Way Calling. When the three-way call begins, the surrogate also relays speech between the callers much as a bridge does in a real exchange.

The ANISE description of Three-Way Calling resembles that for Call Waiting in section 5.2 but extends it for the new facilities. Figure 7 has shown how complex Call Waiting is, so for brevity the ANISE declarations for Three-Way Calling are not given here. Three-Way Calling modifies the basic call in the same kind of way, requiring the introduction of *Recall* and *Select*. The behaviour of the Three-Way Calling surrogate has a definition *CallThree* that is not given here. Coordination at the multiple call level is achieved by the **calls_three** combinator that surrounds the global call instances:

calls_ three(CallThree,**instances**(MaxCalls,Call))

In a normal call, recall creates an instance of *CallThree*. This oversees establishment of a call to the third party. Initial swapping between the original caller and the third party is managed just as for Call Waiting. When a three-way call starts, the surrogate remains to relay speech among the three parties. The surrogate is also involved when the whole three-way call or just one of its legs is cleared.

5.4 Completion of Calls to Busy Subscriber (CCBS)

Description: Completion of Calls to Busy Subscriber, also named Automatic Call-Back, Caller Return and Ring-Back, can be invoked when the caller encounters an engaged number. By dialling a digit (5 in one realisation), the caller can ask to be rung back when the destination is free. At this point the call is connected without having to redial the number.

```
declare(RingBack,                         % ring-back call
  enables(                                % ring caller, seize, ring called
    rings(Ring(21)),                      % ring original caller
    Respond,                              % respond to exchange call
    Ringing))                             % ring phase
```

Figure 8: ANISE Coordination of Completion of Calls to Busy Subscriber

ANISE: The basic call is modified by allowing a digit to be dialled on subscriber busy:

```
enables_on_result(                        % dial, check tone
  dials(Dial),                            % dial selects destination
    SubsBusyTone,finishes(Select),        % if busy, dial extra digit
    RingTone,exit)                        % if ring tone, exit dial phase
```

The **finishes** combinator stops further action after the extra digit has been dialled. At this point, the caller must hang up (i.e. *Clear1* must occur).

A ring-back call is similar to the basic call. The main difference is that the original caller is rung and then responds, instead of seizing the line and then dialling. The *RingBack* declaration in figure 8 can re-use much of the basic call definition in figure 5 (*Ringing*, *Answering*). *Select* and *Respond* features are used to control ring-back. *Select* allows a manual request to return a call to a busy number. Coordination at the multiple call level is achieved by the **returns_manual** combinator that surrounds the global call instances:

returns_manual(RingBack,**instances**(MaxCalls,Call))

It is necessary for **returns_manual** to wait until the calling and called parties are both free. At this point, an instance of *RingBack* is created to re-establish the call. Automatic Call-Back is similar, but needs no manual request since it automatically returns a call to a busy number. The **returns_automatic** combinator is provided for this.

6 Feature Interaction Analysis

6.1 Architectural Analysis

One way in which ANISE helps to detect feature interaction is through static analysis of how the features modify the behaviour of POTS. This is a structural and intuitive examination of how the features are described in ANISE. Features are considered in pairs, looking for the following situations:

- One feature may remove some aspect of the basic call that another depends on. For example, Completion of Calls to Busy Subscriber removes the need for the caller to re-dial a number when ring-back occurs. It is not clear whether features that affect dialling (such as Outgoing Call Screening) should apply in this case.
- One feature may add new behaviour that is normally modified by another. For example, Call Waiting requires the waiting call to ring if the called party hangs up. It is not clear whether features that affect ringing (such as Distinctive Ring) should apply in this case.
- Both features may modify the same behaviour. For example, One Number and Outgoing Call Screening both affect dialling. It is then necessary to decide the relative priorities or execution order of these features.

These kinds of situation merit investigation. Of course, they do not necessarily imply an unwanted interaction but they pinpoint areas for further analysis. One of the practical problems in detecting interactions is that with n features it may be necessary to look at $\frac{n(n-1)}{2}$ pairs of features. Discovering whether two features interact can be time-consuming, so it is useful to have an approach such as the one described here to narrow the focus of investigation.

The architectural approach clearly has limitations. It is intuitive, though some degree of automation could be provided to identify where features relate in their ANISE descriptions. However, the intention is to identify pairs of

ABD	CCBS	CF	CND	CW	DR	OCS	ONE	TCS	TWC	
						1	2		3	ABD
		4	5	6	7	8		9	10	CCBS
			11	12	13	14	15	16	17	CF
				18	19			20		CND
					21			22	23	CW
							24	25		DR
							26		27	OCS
									28	ONE
									29	TCS
										TWC

ABD	Abbreviated Dialling	CCBS	Completion of Calls to Busy Subscriber
CF	Call Forwarding (all forms)	CND	Calling Number Delivery
CW	Call Waiting	DR	Distinctive Ring
OCS	Outgoing Call Screening	ONE	One Number
TCS	Terminating Call Screening	TWC	Three-Way Call

The numbered notes appear in the text

Table 1: Potential Interactions Among ANISE Features

features whose combined behaviour should be studied in more detail. A focused analysis such as in section 6.2 can then be applied to pairs of features that are identified as potentially interacting. It is also possible for more than two features to interact in a unique way, though these situations are less common than strictly pairwise interactions. *N*-way interactions would not be detected by architectural analysis since, by supposition, the features would not be considered to interact when taken in pairs. However, the formal analysis in section 6.2 could in principle detect *N*-way interactions since the overall behaviour of the service can be studied.

It is striking that so many features affect dialling and ringing; these seem to be the major hot-spots for features to alter. Table 1 shows a pairwise comparison of the features in sections 4 and 5. An entry in the table is blank if there is no structural overlap between the features; of course there may still be an undesirable interaction. Where one of the three situations above applies, the possible problems of having both features have been considered. The numbered notes in the table refer to the following discussion, which shows that there are a number of unusual cases to be considered.

1. Abbreviated Dialling must be handled before Originating Call Screening, since short dialling codes might expand to numbers that should be screened.

2. Must Abbreviated Dialling be handled before One Number look-up? In principle a short code could be published as a universal number.

3. Presumably Three-Way Calling must also allow Abbreviated Dialling when establishing the second call leg.

4. If the called party is busy but has Call Forwarding on Busy Line, presumably the call is forwarded rather than giving the caller the option to be rung back. If a call is forwarded to a busy line, should it be the called or forwarding number that is used on ring-back? This question is particularly relevant if Call Forwarding is cancelled before ring-back occurs.

5. Presumably a ring-back call must still identify the caller's number to the called party.

6. If the called party is busy but has Call Waiting, presumably this is invoked rather than reporting busy and allowing Completion of Calls to Busy Subscriber.

7. Presumably ring-back must respect Distinctive Ring so that the called subscriber can recognise which number is being rung.

8. If Completion of Calls to Busy Subscriber is requested for a number that is later banned through Outgoing Call Screening, should ring-back occur?

9. Should Terminating Call Screening be applied on ring-back in case the screening list has changed?
10. If trying to connect the second leg in Three-Way Calling call gets busy, presumably it is not permissible to request ring-back.
11. Presumably it is the number of the real caller that is delivered to the destination, and not that of the telephone at which Call Forwarding was activated.
12. How is the conflict handled between Call Forwarding on Busy Line and Call Waiting?
13. Is Distinctive Ring given according to the called number or the forwarding number? The forwarding number may have its own distinctive ring.
14. Presumably Outgoing Call Screening is applied if Call Forwarding is invoked.
15. May a One Number call be forwarded?
16. Should Terminating Call Screening apply to the called number or to the number that receives the forwarded call?
17. Presumably a call to establish the second leg in Three-Way Calling may be forwarded.
18. Is the waiting caller's numbered delivered if the waiting call is connected by hanging up?
19. Can the calling number be delivered as well as Distinctive Ring?
20. Presumably Terminating Call Screening is applied to the calling number before its delivery is considered.
21. Is Distinctive Ring given if the waiting call is connected through hanging up the current call?
22. Presumably Terminating Call Screening is applied before Call Waiting is considered.
23. When a call is waiting, can Three-Way Calling be initiated? When the second leg in Three-Way Calling is being established, can it be subject to Call Waiting at the third party?
24. Can a Distinctive Ring be allocated to the One Number associated with a telephone?
25. Terminating Call Screening must be applied before Distinctive Ring is considered.
26. Should Outgoing Call Screening be applied before One Number is translated to an actual destination?
27. Presumably a call to establish the second leg in Three-Way Calling is subject to Outgoing Call Screening at the controlling party.
28. Presumably One Number can be used to establish the second leg in Three-Way Calling.
29. Presumably a call to establish the second leg in Three-Way Calling is subject to Terminating Call Screening at the third party.

Of course, the analysis above is retrospective since it describes features that are well known. It does, however, suggest the approach that would be taken when describing and investigating a new feature in ANISE. It is interesting to compare the discoveries of table 1 with current knowledge, such as the feature interaction benchmark of [4]:

- some situations are well-known interactions (cases 10, 12, 14, 16 and 23)
- some situations require feature priorities that are presumably familiar to telecommunications engineers (cases 1, 6, 20, 22, 25 and 26)
- some situations require features to be extended to others (cases 3, 15, 17, 27, 28 and 29).

This nonetheless leaves a number of observations in the table (2, 5, 7, 8, 9, 11, 13, 18, 19, 21, 24) that deserve further study. Relative to [4], table 1 does not appear to identify certain well-known problems. Ambiguity with Call Waiting + Call Waiting does not arise in ANISE because cancelling the current call always returns to the waiting call even if it is on hold. Call Forwarding + Call Forwarding is beyond the scope of ANISE because it assumes sensible profiles that avoid forwarding loops.

6.2 Formal Analysis

ANISE is also intended to allow dynamic analysis of feature interaction through formal means. Appendix A briefly suggests how ANISE can be turned into LOTOS (though the mapping and supporting translator are currently only in prototype form). The static analysis described in section 6.1 would be useful in directing attention to particular cases for formal investigation.

The resulting LOTOS specification can be analysed using standard tools. For example deadlocks, unspecified receptions and non-determinism can be investigated. A common interpretation of feature interaction is that a feature's behaviour is different in isolation compared to when it is embedded in the total system. In LOTOS terms, this means checking some notion of equivalence (e.g. observation) between the feature and system specifications. Another common tell-tale for interaction is non-deterministic behaviour, that can also be discovered formally.

A LOTOS specification can be regarded as a rapid prototype. The behaviour of features in isolation and in conjunction can be studied through simulation of the LOTOS specification. Of course this is just validation (i.e. testing). The theory of canonical testers [1] could be used to derive test suites for services in a rigorous way.

7 Conclusion

A brief explanation has been given of the basis for ANISE. It has been seen that ANISE offers general-purpose features and combinators for describing services, as well as telecommunications-specific combinators that allow typical IN features to be described. Architectural analysis of ANISE descriptions can highlight areas where feature interaction should be investigated in depth. Translation of ANISE into LOTOS permits formal analysis of problems. New constructs can also be added to ANISE by providing their definitions in LOTOS.

The goals listed in section 1.1 have been achieved as follows:

- insights have been gained into how IN-like services can be structured
- an abstract approach has been developed for structuring telecommunications services from features and their combinations
- only user interactions with services are described
- services are built from features in a uniform manner
- features and their combinations are described from the telecommunications point of view, but can be translated to a formal representation in LOTOS
- architectural analysis is possible for static detection of potential feature interactions; formal analysis can be used for more thorough investigation of dynamic feature behaviour and interaction

In the main, ANISE allows IN features to be described fairly compactly. However, ANISE descriptions of some IN features like Call Waiting and Three-Way Calling are intricate, reflecting their inherent complexity. ANISE achieves its expressive power partly through generic combinators and partly through ones that are introduced expressly to describe certain features. To some extent this moves the burden of effort from defining features to defining combinators. However this is a deliberate trade-off, making life easier for the telecommunications engineer but more complex for the ANISE developer.

A static analysis has been given of potential interactions among the features considered in the paper. As expected some well-known interactions appear, and some feature priorities need to be established. However, some new insights have been obtained. A deeper, dynamic analysis awaits completion of the semantics for ANISE and a tool for translating ANISE into LOTOS. As new features are studied, it is expected that the library of ANISE combinators will be extended. However, it is hoped that the present level of ANISE definition will be enough to help in the development of new features.

Acknowledgements

The work reported in this paper is an extension of research conducted during a short-term fellowship funded by British Telecommunications PLC (BT). The author is grateful for the support by BT that made this original work possible. Malte Plath (University of Birmingham) kindly provided some inputs to the work in the present paper. The comments of the anonymous referees were helpful in improving the presentation.

References

[1] R. Alderden. COOPER – the compositional construction of a canonical tester. In S. T. Vuong, editor, *Proc. Formal Description Techniques II*, pages 13–18. North-Holland, Amsterdam, Netherlands, 1990.

[2] Jan Bergstra and Wiet Bouma. Models for feature descriptions and interactions. In Petre Dini, Raouf Boutaba, and Luigi M. S. Logrippo, editors, *Proc. 4th. International Workshop on Feature Interactions in Telecommunication Networks*, pages 31–45. IOS Press, Amsterdam, Netherlands, June 1997. ISBN 90-5199-3471.

[3] Johan Blom. Formalisation of requirements with emphasis on feature interaction detection. In Petre Dini, Raouf Boutaba, and Luigi M. S. Logrippo, editors, *Proc. 4th. International Workshop on Feature Interactions in Telecommunication Networks*, pages 61–77. IOS Press, Amsterdam, Netherlands, June 1997. ISBN 90-5199-3471.

[4] E. Jane Cameron, Nancy D. Griffeth, Yow-Jian Lin, Margaret E. Nilson, William K. Schnure, and Hugo Velthuijsen. A feature-interaction benchmark for IN and beyond. *IEEE Communications Magazine*, pages 64–69, March 1993.

[5] Rachida Dssouli, Stephane Somé, J.-W. Guillery, and Nathalie Rico. Detection of feature interactions with REST. In Petre Dini, Raouf Boutaba, and Luigi M. S. Logrippo, editors, *Proc. 4th. International Workshop on Feature Interactions in Telecommunication Networks*, pages 271–283. IOS Press, Amsterdam, Netherlands, June 1997.

[6] Fabrice Dupuy, Gunnar Nilsson, and Yuji Inoue. The TINA consortium: Towards networking telecommunications information services. *IEEE Communications Magazine*, pages 78–83, November 1995.

[7] Mohammed Faci, Luigi Logrippo, and Bernard Stepien. Formal specification of telephone systems in LOTOS: The constraint-oriented style approach. *Computer Networks and ISDN Systems*, 21(1):53–67, March 1991.

[8] Mohammed Faci, Luigi Logrippo, and Bernard Stepien. Structural models for specifying telephone systems. *Computer Networks and ISDN Systems*, 29(4):501–528, March 1997.

[9] Mohammed Faci and Luigi M. S. Logrippo. Specifying features and analysing their interactions in a LOTOS environment. In L. G. Bouma and Hugo Velthuijsen, editors, *Proc. 2nd. International Workshop on Feature Interactions in Telecommunications Systems*, pages 136–151. IOS Press, Amsterdam, Netherlands, May 1994.

[10] 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.

[11] 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.

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

[13] ISO/IEC. *Information Processing Systems – Open Distributed Processing – Basic Reference Model*. ISO/IEC 10746. International Organization for Standardization, Geneva, Switzerland, 1995.

[14] ISO/IEC. *Information Processing Systems – Open Systems Interconnection – Enhancements to LOTOS – A Formal Description Technique based on the Temporal Ordering of Observational Behaviour.* ISO/IEC CD. International Organization for Standardization, Geneva, Switzerland, February 1998.

[15] ITU. *Intelligent Network – Introduction to Intelligent Network Capability Set 1*. ITU-T Q.1211. International Telecommunications Union, Geneva, Switzerland, 1993.

[16] ITU. *Intelligent Network – Q.120x Series Intelligent Network Recommendation Structure.* ITU-T Q.1200. International Telecommunications Union, Geneva, Switzerland, 1993.

[17] ITU. *Intelligent Network – Introduction to Intelligent Network Capability Set 2*. ITU-T Q.1221. International Telecommunications Union, Geneva, Switzerland, 1997.

[18] Brian W. Kernighan and Dennis M. Ritchie. The *m4* macro processor. Technical report, Bell Laboratories, Murray Hill, New Jersey, USA, 1977.

[19] 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. 2nd. International Workshop on Feature Interactions in Telecommunications Systems*, pages 86–119. IOS Press, Amsterdam, Netherlands, 1994.

[20] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modelling and Design*. Prentice-Hall, Englewood Cliffs, New Jersey, USA, 1991.

[21] Richard O. Sinnott and Kenneth J. Turner. Applying formal methods to standard development: The Open Distributed Processing experience. *Computer Standards and Interfaces*, 17:615–630, October 1995.

[22] Muffy H. Thomas. Modelling user views of telecommunications services for feature interaction detection and resolution. In Petre Dini, Raouf Boutaba, and Luigi M. S. Logrippo, editors, *Proc. 4th. International Workshop on Feature Interactions in Telecommunication Networks*, pages 168–182. IOS Press, Amsterdam, Netherlands, 1997.

[23] 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.

[24] Kenneth J. Turner. Exploiting the *m4* macro language. Technical Report CSM-126, Department of Computing Science and Mathematics, University of Stirling, UK, September 1994.

[25] Kenneth J. Turner. An architectural foundation for relating features. In Petre Dini, Raouf Boutaba, and Luigi M. S. Logrippo, editors, *Proc. 4th. International Workshop on Feature Interactions in Telecommunication Networks*, pages 226–241. IOS Press, Amsterdam, Netherlands, 1997.

[26] Kenneth J. Turner. Incremental requirements specification with LOTOS. *Requirements Engineering Journal*, 2:132–151, November 1997.

[27] Kenneth J. Turner. Relating architecture and specification. *Computer Networks and ISDN Systems*, 29(4):437–456, March 1997.

[28] Kenneth J. Turner. Specification architecture illustrated in a communications context. *Computer Networks and ISDN Systems*, 29(4):397–411, March 1997.

[29] Kenneth J. Turner and Richard O. Sinnott. DILL: Specifying digital logic in LOTOS. In Richard L. Tenney, Paul D. Amer, and M. Ümit Uyar, editors, *Proc. Formal Description Techniques VI*, pages 71–86. North-Holland, Amsterdam, Netherlands, 1994.

[30] Rob van der Linden. Using an architecture to help beat feature interaction. In L. G. Bouma and Hugo Velthuijsen, editors, *Proc. 2nd. International Workshop on Feature Interactions in Telecommunications Systems*, pages 24–35. IOS Press, Amsterdam, Netherlands, 1994.

[31] Chris A. Vissers, Giuseppe Scollo, and Marten van Sinderen. Architecture and specification style in formal descriptions of distributed systems. *Theoretical Computer Science*, 89:179–206, 1991.

A Defining and Extending ANISE

A.1 Generic Features and Combinators

Table 2 summarises the generic ANISE constructs used in this paper. This table gives general-purpose combinators that are not specific to telephony. A rather similar set of constructs is used in [23] to describe services in OSI (Open Systems Interconnection [12]). Only some of the generic ANISE constructs are needed for the kinds of features

Declaration	Meaning
% *comment text*	introduces informal explanation
behaviour	feature declaration or combination
collides(*behaviour*)	copies of behaviour execute separately in each direction, mutually reinforcing on collision
declare(*name*,*behaviour*)	gives name to behaviour
direction	local/remote logical *endpoints*, such as 13 or S1
disables(*behaviour1*,*behaviour2*)	execution of first behaviour stops the second one
duplicates(*behaviour*)	copies of behaviour execute independently in each direction simultaneously
enables(*behaviour1*,*behaviour2*,...)	when each behaviour terminates, the next may start
enables_on_result(*behaviour1*,*result2*,*behaviour2*, *result3*,*behaviour3*,...,*behaviourN*)	result of first behaviour decides which later one starts: if *result2* then *behaviour2*, ..., *behaviourN* by default
endpoint	logical identification of an endpoint, such as 0 (undefined), 1 (first caller) or S (surrogate)
exit	current behaviour terminates successfully
feature(*direction*,*pattern*,*property*,*group*(*param*))	elementary behaviour with given characteristics, for one group
feature(...,*group1*(*param*),*group2*(*param*))	elementary behaviour with given characteristics, for request and acknowledgement groups
finishes(*behaviour*)	no further action after behaviour terminates
global(*name*,*behaviour*)	global specification behaviour with given service name
group	common part of service primitive names
holds(*behaviour1*,*behaviour2*)	first behaviour interrupts second behaviour, then resumes it when repeated
instances(*count*,*behaviour*)	give number of independent behaviour instances
interrupts_after_try(*behaviour1*,*behaviour2*)	first behaviour may interrupt then restart second one, after second executes request or indication
loops(*behaviour*)	behaviour repeats on successful termination
name	identifier (letters, digits, underscores)
param	service primitive parameter (*CalledMess*, *CallingMess*, *Dig*, *Num*, *Voice*)
pattern	**asymmetric_confirmed**, **local_confirmed**, **provider_confirmed**, **provider_initiated**, **remote_confirmed**, **unconfirmed**, **user_confirmed**, **user_initiated**, **user_provider_confirmed**
profile(*number*,*identifier*,...)	subscriber number, line identifier and profile
property	**consecutive**, **ordered**, **reliable**, **single**, **unreliable**

Table 2: Referenced ANISE Generic Constructs

Declaration	Meaning
calls_three(*behaviour1*,*behaviour2*)	three-way calls follow first behaviour, normal calls follow second one
checks_busy(*behaviour*)	monitors and respects the busy state of lines in behaviour
dials(*behaviour*)	dialled number determines called number in behaviour
dials_code(*behaviour*)	abbreviated dialling code is expanded for behaviour
dials_one(*behaviour*)	called universal number is translated to called identifier
diverts_always(*behaviour*)	called identifier for behaviour is changed unconditionally
diverts_busy(*behaviour*)	called identifier for behaviour is changed if called is busy
diverts_unanswered(*behaviour*)	called identifier for behaviour is changed if call is unanswered
returns_ automatic(*behaviour1*,*behaviour2*)	automatic ring-back calls follow first behaviour, normal calls follow second one
returns_ manual(*behaviour1*,*behaviour2*)	manual ring-back calls follow first behaviour, normal calls follow second one
rings(*behaviour*)	called message in behaviour has appropriate ring pattern
rings_display(*behaviour*)	called message in behaviour supplies calling number
rings_preference(*behaviour*)	called message in behaviour supplies called ring preference
screens_ in(*behaviour*)	behaviour for banned calling numbers will not terminate
screens_out(*behaviour*)	behaviour for banned called numbers will not terminate
waits_call(*behaviour1*,*behaviour2*)	waiting calls follow first behaviour, normal calls follow second one

Table 3: ANISE Telecommunications Service Constructs

described in the paper. Table 3 summarises the ANISE constructs for describing telecommunications services such as those in sections 4 and 5. [25] discusses the foundations of ANISE, in particular table 2.

The intention is that ANISE should offer a library of combinators that are used in practice for telecommunications services. It is clearly not possible to anticipate every need. ANISE is, however, meant to facilitate the introduction of new services. Simple services do not require new combinators (as shown in [25]), but realistic services such as those in sections 4 and 5 do require additions. ANISE is therefore open-ended in the sense that it is possible to define new combinators. To do so requires a language for defining combinators and some idea of how the existing combinators are defined.

ANISE derives from a similar language called SAGE (Service Attribute Generator [23]) for describing services in OSI. The semantics of SAGE was defined denotationally, giving each language construct a representation in LOTOS (Language Of Temporal Ordering Specification [10]). In fact, the semantics of SAGE was given implicitly by building a SAGE to LOTOS translator written using the *m4* macro language [18]. Although *m4* is just a macro processor, it is flexible and appropriate to the task [24].

A semantics for ANISE has been largely worked out using the same approach as for SAGE. The semantics will be defined in a forthcoming paper, so only some examples will be given here. The main reasons for using LOTOS as the underlying language are its built-in data typing and flexible synchronisation rules. Some features (such as a call plan or diversion on no answer) may have time-dependent behaviour. E-LOTOS (Enhancements to LOTOS [14]) might therefore be a better choice than current LOTOS. In any case, E-LOTOS offers many improvements including clearer data typing and more powerful operators such as suspend-resume that would simplify the translation from ANISE.

A number of the ANISE combinators can be defined in the constraint-oriented style [26, 31] that allows existing behaviour to be restricted in some way. A constraint process is put in parallel with existing behaviour to synchronise on its events, except for certain undesired ones.

The occurrence of a service primitive in ANISE corresponds to a LOTOS event with structure:

tel ! num ! prim

where the gate name *tel* is the service name given by the **global** combinator, *num* identifies the number where the primitive is occurring, and *prim* is a data value for a service primitive. The numbers at both endpoints start out undefined, and become instantiated on the first event at each side of a call. For internal call control there are also events at the hidden gate *ctl*.

ANISE features and nearly all combinators are mapped onto LOTOS process definitions. Processes are named after the feature primitive group or combinator, and numbered sequentially in case there are several examples of the same kind. The LOTOS examples that follow were generated by the prototype translator.

The data type for primitives *Prim* and its associated types are automatically generated by ANISE from the elementary features. The data types are relatively complicated to allow for the large variety of feature patterns and properties in figures 1 and 2. The operations defined by ANISE for primitives include:

IsGroup: primitive is of a given group (e.g. *OnHookGroup* for all *OnHook* primitives)

IsKind: primitive is of a given kind (e.g. *OnHookReqKind* for *OnHook.Req*)

IsReq/*IsInd*/*IsRes*/*IsCon:* primitive is a request/indication/response/confirm

IsNextPrim: checks that a pair of primitives follow from each other, e.g. *Speech.Ind* follows *Speech.Req* and must carry the same voice segment.

The type *PrimQueue* is used when primitives have to be delivered in order; such a queue can have delivery priorities.

The behaviour of a feature is defined by its direction, pattern and property as discussed in section 2.3. As an example, the *Clear1* feature in figure 5:

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

is translated to the following LOTOS:

```
process OnHookFeat [g1, g2] (n1, n2 : Num) : exit (Num, Num, Result) :=
  g1 ? id1 : Id ? prim1 : Prim                                (* allow primitive ... *)
    [IsKind (prim1, OnHookReqKind) and IsId (id1, n1)];       (* if OnHook.Req and valid Id *)
  exit (IdNum (id1, n1), n2, ResultOf (prim1))                (* exit with numbers and result *)
endproc
```

The current gates and telephone numbers are supplied as parameters. Both gates are usually set to the normal communication gate *tel*, but either may be set to the hidden control gate *ctl*. This is necessary when an endpoint is connected to a surrogate user (as for Call Waiting or Three-Way Calling). *OnHookFeat* allows *OnHook.Req* at endpoint 1 with a valid *Id*. The values of the numbers are exported for future use along with the result of *OnHook.Req*. The relationship between *Id*s and numbers is stored in the profiles. *IsId* checks that an *Id* corresponds to a number, while *IdNum* determines the number for an *Id* and a number. *IsId* and *IdNum* handle the case of an undefined number on call start-up.

As a more complex example of a feature, consider *Seize* in figure 5:

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

This is translated into LOTOS as follows:

```
process OffHookFeat [g1, g2] (n1, n2 : Num) : exit (Num, Num, Result) :=
  g1 ? id1 : Id ? prim1 : Prim                                (* allow primitive ... *)
    [IsKind (prim1, OffHookReqKind) and IsId (id1, n1)];      (* if OffHook.Req and valid Id *)
  g1 ! id1 ? prim2 : Prim                                     (* allow next primitive ... *)
    [IsNextPrim (prim2, prim1)];                              (* if OffHook.Con *)
  exit (IdNum (id1, n1), n2, ResultOf (prim2))                (* exit with numbers and result *)
endproc
```

This is rather similar to *OnHookFeat* except that it allows a second primitive (confirm being next after request). The result of the feature is the calling message carried by *OffHook.Con* (normally dialling tone).

A combinator takes a feature or other combinators as argument. The meaning of a combinator is given by a new LOTOS process that uses the given one. As a simple example of a combinator, the declaration of two-way speech in figure 5:

duplicates(Speak)

is translated to the following LOTOS:

```
process Duplicates [tel, ctl] (n1, n2 : Num) : noexit :=
  SpeechFeat [tel, tel] (n1, n2)                        (* speech from user 1 to 2 *)
|||
  SpeechFeat [tel, tel] (n2, n1)                        (* speech from user 2 to 1 *)
endproc
```

Duplicates simply interleaves an instance of speech *SpeechFeat* in each direction. It has no exit values since speech is non-terminating. The *Duplicates* process may in turn be used by other, higher-level combinators.

As a more complex example of a combinator, consider the use of **interrupts_after_try** in figure 5:

interrupts_ after_ try(Clear1,**enables_ on_ result**(...))

The **interrupts** combinator allows the first (interrupting) behaviour to stop the second (main) behaviour. After the interrupting behaviour has completed, the whole behaviour is restarted. The **interrupts_after_try** variant allows interruption only after the main behaviour has started (i.e. there has been a request or indication). This combinator is translated to the following LOTOS:

```
process InterruptsAfterTry [tel, ctl] (n1, n2 : Num) : exit (Num, Num, Result) :=
  (
    EnablesOnResult [tel, ctl] (n1, n2)                 (* OffHook/Dial *)
  [>
    (
      tel ? id1 : Id ? prim : Prim                      (* allow primitive ... *)
        [IsGroup (prim, OnHookGroup)];                  (* if a clear *)
      InterruptsAfterTryA [tel, ctl] (IdNum (id1, n1), n2)  (* allow other primitives *)
    )
  )
||
  (
    tel ? id : Id ? prim : Prim                         (* allow primitive ... *)
      [not (IsGroup (prim, OnHookGroup))];              (* if normal Req/Ind *)
    InterruptsAfterTryB [tel, ctl] (IdNum (id), n2)     (* allow other primitives *)
  )
where
  process InterruptsAfterTryA [tel, ctl] (n1, n2 : Num) : exit (Num, Num, Result) :=
    tel ? id : Id ? prim : Prim                         (* allow any primitive ... *)
      [IsId (id, n1) or IsId (id, n2)];                 (* if valid Id *)
    InterruptsAfterTryA [tel, ctl] (n1, n2)             (* allow other primitives *)
  []
    exit (any Num, any Num, any Result)                 (* exit when required *)
  endproc
  process InterruptsAfterTryB [tel, ctl] (n1, n2 : Num) : exit (Num, Num, Result) :=
    (
      OnHookFeat [tel, tel] (n1, n2)                    (* allow clear *)
    >> accept n1, n2 : Num, res : Result in
      InterruptsAfterTry [tel, ctl] (n1, n2)            (* repeat whole behaviour *)
    )
  []
    tel ? id : Id ? prim : Prim                         (* allow primitive ... *)
      [not (IsGroup (prim, OnHookGroup))];              (* if not a clear *)
    InterruptsAfterTryB [tel, ctl] (n1, n2)             (* allow other primitives *)
  []
    exit (any Num, any Num, any Result)                 (* exit when required *)
  endproc
endproc
```

EnablesOnResult is the process defined by the **enables_on_result** combinator for seizing the line and dialling. *OnHookFeat* is the process defined by the one-sided clear feature. The **interrupts_after_try** combinator introduces two synchronised behaviours. *EnablesOnResult* may be disabled by a clear, followed by *InterruptsAfterTryA* that allows any primitive or can exit. In parallel with this, a normal request or indication must precede *InterruptsAfterTryB*. This then allows a clear followed by repetition of the whole behaviour. Other primitives from *EnablesOnResult* may occur without constraint, and *EnablesOnResult* is allowed to exit.

A.2 Telephony Combinators

The **dials_code** combinator used in section 4.1 to modify **dials** for abbreviated dialling:

dials(**dials_code**(Dial))

is translated to the following LOTOS that maps a short dialling code to the destination line identifier:

```
process Dials [tel, ctl] (n1, n2 : Num) : exit (Num, Num, Result) :=
  (
    DialFeat [tel, tel] (n1, n2)                         (* allow dial primitives *)
  >> accept n1, n2 : Num, res : Result in
    exit (n1, any Num, res)                              (* calling number, dial result *)
  )
|[tel]|
  DialsA [tel, ctl] (n1, n2)                             (* constrain called number *)
where
  process DialsA [tel, ctl] (n1, n2 : Num) : exit (Num, Num, Result) :=
    tel ? id1 : Id ? prim : Prim                         (* allow primitive ... *)
      [(IsCon (prim) implies                             (* if Dial.Con ... *)
        IsNumCallingMess (n2, CallingMessOf (prim)))];   (* has consistent message *)
    (
      [IsReq (prim)] ->                                  (* Dial.Req? *)
        ctl ! CtlDial ! n1                               (* request association with ... *)
          ! ProfCallAbbr (n1, NumOf (prim), Profiles);   (* unabbreviated number *)
        ctl ! CtlDial ? n1, n2 : Num;                    (* accept associated number *)
        DialsA [tel, ctl] (IdNum (id1, n1), n2)          (* allow other primitives *)
    []
      [not (IsReq (prim))] ->                            (* not Dial.Req? *)
        DialsA [tel, ctl] (IdNum (id1, n1), n2)          (* allow other primitives *)
    )
  []
    exit (any Num, n2, any Result)                       (* called number, dial result *)
  endproc
endproc
```

In general, defining a new feature may require the structure of the subscriber profile record to be extended with a new field plus some operations to manipulate it. The **dials_code** combinator requires a set of short code/full number pairs to be stored in a subscriber's profile. The *ProfCallAbbr* operation looks up the dialled number in the profile for the calling number. If the called number is present in the short code list it is expanded to the full number, otherwise the number is used as given.

DialFeat is the process defined by the dialling feature. When it exits, it fixes the calling number and dialling result. *DialsA* runs in parallel with this to constrain dialling primitives. If *Dial.Con* occurs, the calling message it contains must be consistent with the status of the dialled number (e.g. calling a busy number gets subscriber busy). If *Dial.Req* occurs, the dialled number is used to generate the called number via the internal control gate *ctl*. As will be seen, the global process that monitors line status also synchronises on *ctl*. It returns the actual destination number, which may differ due to call forwarding for example. *DialsA* constrains only *Dial.Req* and *Dial.Con*. It is prepared to exit when *DialFeat* does, setting the called number to that determined by dialling.

As a final example, the **checks_busy** combinator used in figure 5 to monitor line busy/free:

checks_busy(**instances**(MaxCalls,Call))

is translated to the following process:

```
process ChecksBusy [tel, ctl] : noexit :=
  Instances [tel, ctl]                                   (* all call instances *)
||
    ChecksBusyA [tel, ctl] (<>)                                  (* busy/free check *)
  where
    process ChecksBusyA [tel, ctl] (Calls : IdAssocs) : noexit :=
      tel ? id : Id ? prim : Prim                                (* allow primitive ... *)
        [IsKind (prim, OffHookReqKind) implies                   (* if OffHook.Req ... *)
          IdAssocAvail (id, Calls)];                             (* is for available line *)
      (
        [IsKind (prim, OffHookReqKind) or                        (* picking up or ... *)
         IsKind (prim, StartRingIndKind)] ->                     (* starting ringing ... *)
          ChecksBusyA [tel, ctl] (IdAssocAdd (id,Calls))         (* marks line as busy *)
      []
        [IsKind (prim, OnHookReqKind) or                         (* hanging up or ... *)
         IsKind (prim, StopRingIndKind)] ->                      (* stopping ringing ... *)
          ChecksBusyA [tel, ctl] (IdAssocRem (id, Calls))        (* marks line as free *)
      []
        [not (IsKind (prim, OffHookReqKind) or IsKind (prim, StartRingIndKind) or
         IsKind (prim, OnHookReqKind) or IsKind (prim, StopRingIndKind))] ->
          ChecksBusyA [tel, ctl] (Calls)                         (* busy/free unaltered *)
      )
    []
      ctl ! CtlDial ? n1, n2 : Num;                              (* accept association request *)
      (
        let n2A : Num = IdAssocNum (n2, Calls) in                (* set associated number *)
          ctl ! CtlDial ! n1 ! n2a;                              (* return associated number *)
          (
            [n2A ne BusyNum] ->                                  (* associate number if free *)
              ChecksBusyA [tel, ctl] (IdAssocLink (NumId (n1), NumId (n2a), Calls),)
          []
            [n2A eq BusyNum] ->                                  (* no association if busy *)
              ChecksBusyA [tel, ctl] (Calls)
          )
      )
    endproc
  endproc
```

The interleaved calls generated by **instances** are synchronised with *ChecksBusyA*. This process maintains the current list of associations: pairs of *Id*s that are currently connected by a call. *ChecksBusyA* starts with an empty list of associations (<>). A number of auxiliary functions are used to manage *Id* associations:

IdAssocAvail: checks if a line is available

IdAssocAdd: adds a line to an association

IdAssocRem: removes a line from an association

IdAssocLink: creates an association between a pair of lines

IdAssocNum: determines the actual called number from the dialled one.

ChecksBusyA permits any primitive to occur except in one case: going off-hook requires a number to be available. Picking up the telephone or starting it ringing (i.e. *OffHook.Req* or *StartRing.Ind*) mark a line as busy. Hanging up the telephone or stopping it ringing (i.e. *OnHook.Req* or *StopRing.Ind*) mark a line as free. Other actions do not affect the set of busy lines. *ChecksBusyA* accepts an enquiry at the *ctl* gate, asking to create an association between a pair of numbers. The third value in such events is set to the destination number determined by this pair. *ChecksBusyA* then updates its list of associations.

Biography

Ken Turner graduated in electrical engineering from the University of Glasgow in 1970. He was awarded a Ph.D. from the University of Edinburgh in 1974 for his research on pattern recognition. Until 1986 he was employed by International Computers Ltd. as a data communications consultant. During this period he specialised in systems architecture, data communications and formal methods. This led to his appointment as Professor of Computing Science at the University of Stirling in 1987. His research interests include formalisation of systems architecture, analysis of telecommunications services, quality of service, and formal methods for hardware-software co-design.