Kenneth J. Turner. Validating Architectural Feature Descriptions using LOTOS. In Kristofer Kimbler and Wiet Bouma, editors, Proc. Feature Interactions in Telecommunication Networks V, IOS Press, Amsterdam, September 1998.

Validating Architectural Feature Descriptions using LOTOS

Kenneth J. Turner

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

Email: kjt@cs.stir.ac.uk

Abstract.
The phases of the ANISE project (Architectural Notions In Service Engineering) are briefly explained with reference to the work reported here. An outline strategy is given for translating ANISE descriptions to LOTOS (Language of Temporal Ordering Specification), thus providing a formal basis. It is shown how modular ANISE descriptions of features can be defined and then merged. Potential feature interactions can be identified statically through structural overlaps. A scenario language is introduced to express validation tests for features in a modular fashion, and a number of examples are given. Scenarios are automatically translated to LOTOS and analysed through LOTOS simulation. This allows features to be validated in isolation, and dynamically in combination with other features. The design of the translation and validation tools is discussed, showing typical results when investigating feature descriptions. The paper concludes with a guide to extending the approach for new features.

1 Introduction

1.1 Context of the Work

ANISE (Architectural Notions In Service Engineering) is an approach, a language and a toolset for defining and analysing telecommunications services and features in a rigorous fashion. For the work reported in this paper, the specific goals were: to define a largely complete denotational semantics for ANISE; to devise a strategy and tool for translation to LOTOS; to develop a language and tool for combining modular descriptions of features; and to develop a language and tool for validating the resulting specifications.

The approach deliberately does not give any special status to services, features, Service Independent Building Blocks or the basic call. All are considered behaviours – features – of a telecommunications system. Elementary behaviours are used to build more complex behaviours. Note that ANISE uses the term ‘feature’ to mean any self-contained behaviour of a telecommunications service. An ANISE feature may be anything from basic communication between the user and a service (e.g. dialling, speech) to higher-level combinations (e.g. Abbreviated Dialling, Three-Way Calling). Normally only the latter would be called services or features. However all behaviours are technically the same in nature, so ANISE talks about features even if the behaviour would not normally be marketed by itself.

Work on ANISE has been proceeding in a number of well-defined phases. Initially an architectural foundation for describing telecommunications features was defined [10], allowing the basic call and simple modifications of it to be described. Extensions then permitted typical telecommunications features to be described [12], and static (structural) analysis to

highlight potential interaction areas. In the recent phase reported in this paper, the semantics of ANISE have been defined through translation to LOTOS (Language Of Temporal Ordering Specification [3]). Apart from ascribing formal meaning to ANISE descriptions, this permits rigorous analysis and validation. Tool support has also been implemented, allowing services to be defined and validated in a modular manner. Language support has been introduced for merging ANISE descriptions and expressing tests of them. In a future phase, verification (proof) and formally-derived tests will be employed.

1.2 Philosophy

An important aim of ANISE is to understand the construction of services and features. It is hoped that a more consistent architecture will help to avoid interactions, so in this respect ANISE is an avoidance method. However, ANISE also aims to support rapid prototyping and analysis. In this respect ANISE is an off-line detection method. Interactions are found at a high-level of feature specification, without reference to implementation details. Since ANISE obliges precise formulation of features, the very act of specification often identifies interactions. ANISE also aims to validate the intrinsic logic of a feature, though of course further interactions may arise at the implementation level. As will be seen, ANISE can statically highlight possible feature interactions. This is not strictly detection since further analysis is needed to find which interactions are genuine. ANISE can also detect dynamic interactions among arbitrary features during validation; such interactions appear as deadlock or non-determinism. Any interactions identified by ANISE must be resolved manually, i.e. by modifying the specification. This helps to expose the essential relationships between features.

Although ANISE uses LOTOS as its formal basis, the approach intentionally hides its use. As will be seen, features can be described, combined and validated without the user seeing any LOTOS. This is partly because LOTOS is a specialised formal language that a service engineer is unlikely to be familiar with. ANISE obtains the benefits of formality without forcing LOTOS upon the user. The other reason for hiding LOTOS is that ANISE is independent of its underlying formalism. Although LOTOS is the target language for the current tools (their 'machine code'), it would be possible to support ANISE using other formal languages like SDL (Specification and Description Language [6]) or using programming languages like Java. LOTOS is thus merely a means to an end, although it is convenient and powerful.

ANISE tool support was considered to be important for a number of reasons. For ANISE to have a usable semantic basis, automatic translation to LOTOS is essential. Telecommunications features (e.g. Call Waiting, Three Way Calling) can be complex and need automated analysis. Since the intended users of ANISE are telecommunications engineers, tool support is required if the approach is to be acceptable. Figure 1 shows the relationship between tools in the current suite. The functions of these tools will be explained in later sections. The entire toolset has 14 modules and 6,300 lines of code (including comments). In fact the TOPO suite of LOTOS tools and the LOLA simulator [7] are a separate and substantial part of the ANISE validator, though the existence of these is hidden from the user. Further tools are planned, for example to create and animate ANISE descriptions graphically.

1.3 Related Work

ANISE supports detailed and realistic descriptions of telecommunications services. Comparing ANISE descriptions with those written using other approaches, it is interesting to note that ANISE often provides a finer level of detail. As a simple example, it is commonly assumed that going off-hook leads to dialling tone. ANISE correctly allows for other outcomes, including

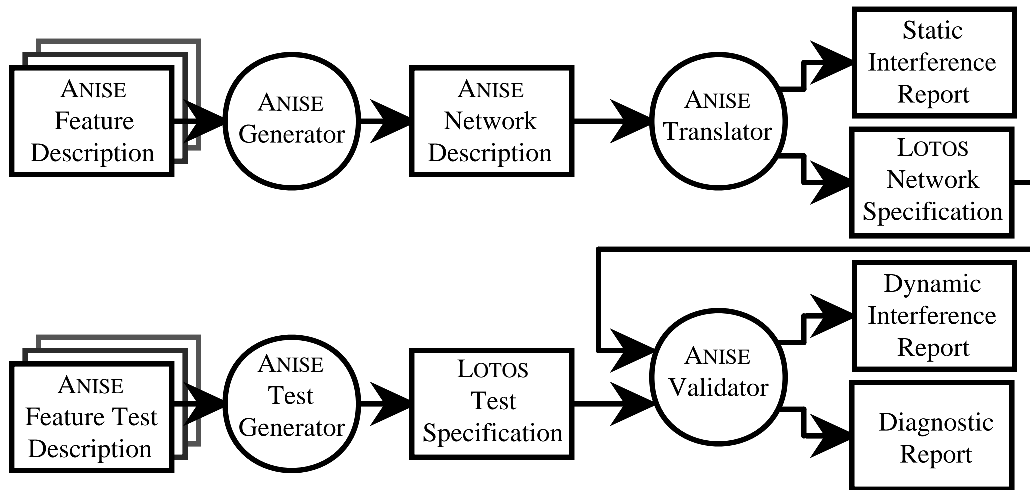

Figure 1: Tool Support for ANISE

going on-hook before dialling tone is received. Although this is largely a modelling issue (i.e. a single off-hook/dialling tone event could be considered), ANISE nonetheless encourages and supports a detailed description of how services work. Although this complicates the model to some extent, it helps to give a more accurate specification of behaviour. More importantly, a finer level of granularity in the events allows more subtle interactions to be discovered. Telecommunications networks are inherently concurrent and distributed, so fine-grained concurrency in the models may uncover more problems.

Since ANISE is supported by LOTOS, it is useful to compare it 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 [2], and have shown how the feature interaction problem can be analysed in LOTOS. Thomas [8] also uses LOTOS but with particular interest in formulating a theory of features, and in detecting interactions by checking temporal logic properties against a LOTOS specification.

The author's experience of using LOTOS for architectural specification [11] is that LOTOS is rather general. Although LOTOS is fairly abstract, it is nonetheless too low-level in terms of a typical application domain. For this reason, the author has found it more productive to specify problems in a given domain using a meta-language that directly supports domain concepts. For example, when describing telecommunications services it is useful to have direct access to specification concepts like telephone number, dialling, feature, call, diversion and subscriber profile. Using only LOTOS it is necessary to express such concepts from scratch. However, a meta-language can embody these concepts and yet be translated automatically to LOTOS. This allows the specifier to work at the level of the application domain, resulting in smaller descriptions that relate better to the problem area. As will be seen later, ANISE descriptions are much more compact than their automatically generated LOTOS counterparts. The author has found that descriptions in a problem-specific meta-language are often only a few percent in size of their LOTOS counterparts.

2 Feature Description

2.1 Language

ANISE offers constructs for defining elementary behaviours and combinators for building these into more complex behaviours. [10] discusses the generic capabilities of ANISE, while [12]

introduces its capabilities for specific telecommunications features. Only a basic indication is given in this section of how ANISE is translated into LOTOS. The details of the translation are intricate, and not much more interesting than the internals of a compiler.

2.2 Translation Strategy

2.2.1 Elementary Features

Elementary features describe the simplest communication behaviours between a user and the network. Such features are modelled following the OSI (Open Systems Interconnection) concepts of service and service primitive. Service primitives play one of four roles: request/indication (notification user↔network) and response/confirm (acknowledgement user↔network). Consider the elementary *Seize* feature described in ANISE by:

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

This allows the user to go off-hook, and the network to confirm this. The first primitive is an off-hook request (corresponding to going picking up the telephone). This is followed by an off-hook confirm (normally supplying dial tone).

Features have a defined direction (12 means from user 1, the originator, to user 2, the responder). Features also have a pattern (**local_confirmed** means that a user request gets a network confirm). An ordering property defines how multiple invocations of a feature are related (**single** means that only one instance of a feature may be invoked at a time). The request/indication and response/confirm primitives are declared in the form *Name(Parameters)*. The name for response/confirm is usually the same as for request/indication and can be omitted. If there are no parameters they are omitted. For *Seize* the primitives are all called *OffHook*. There is no parameter for request/indication, but for response/confirm it is of type *CallingMess*. This is used for network messages to the calling party and includes dial tone.

Communication between a user and the network corresponds to a LOTOS event like *tel !42 !OffHookReq*. All events between users and the network take place at a common gate chosen by the specifier, here *tel* (telephone). Events are associated with an *Id* like 42 that specifies the line on which they occur. It is necessary to use a line identifier since several telephone numbers may be associated with the same line (e.g. for Distinctive Ring). The other information in an event is a service primitive value like *OffHookReq*. Auxiliary operations are defined, like *IsKind* (primitive is of given kind) and *IsId* (line identifier matches telephone number). *IsNextPrim* checks if one primitive logically follows another (in the sense that a confirm with dial tone may follow an off-hook request).

The parameters of an elementary feature determine its LOTOS translation. An infinite number of parameter combinations may be used; patterns and properties alone define 40 different variations. This aspect of the translation is therefore moderately complex, though it is entirely automated. Each declaration of an elementary feature is translated to a LOTOS process definition. This is necessary because some elementary features may be invoked repeatedly, so recursion must be possible. In the case of *Seize*, the generated LOTOS is:

```
process OffHookFeat [g1, g2] (n1, n2 : Num) : exit (Num, Num, Result) :=
  g1 ?id1 : Id ?prim1 : Prim                                (* allow primitive if ... *)
    [IsKind (prim1, OffHookReqKind) and IsId (id1, n1)];  (* off-hook request for caller *)
  g1 !id1 ?prim2 : Prim                                     (* allow primitive if ... *)
    [IsNextPrim (prim2, prim1)];                            (* off-hook confirm *)
  exit (IdNum (id1, n1), n2, ResultOf (prim2))              (* exit with numbers, tone *)
endproc (* OffHookFeat *)
```

An elementary feature process is named according to service primitive (here, *OffHook*). The event gates used for communication with the originating and terminating users are *g1* and *g2*; normally these are both set to the global gate *tel*. The process is parameterised by the originating/terminating user numbers (*n1* and *n2*). In fact these are not fixed when a call starts, and are determined by the calling and called numbers. The numbers associated with a call may change as the call evolves (e.g. due to call diversion), so the numbers in use are exported when a process exits. *IdNum* in the above sets the originating number according to the line that went off-hook. Features also have a result value – the parameter of their last service primitive. For going off-hook, *ResultOf* returns the calling message (usually dial tone); this result may be tested by later features.

ANISE provides common types such as *Num* (telephone number) and *Result* (value generated by a feature) that are defined by specific LOTOS data types. The trickiest part of the translation concerns the type definitions for service primitives since the specifier may choose any names and parameters for these. For the *Seize* feature, the translation defines the following operations for service primitive values of type *Prim*, as well as some auxiliary operations:

```
OffHookReq :                 -> Prim        (* go off-hook)
OffHookCon : CallingMess     -> Prim        (* get dial tone, etc. *)
```

2.2.2 Combinators

More complex call behaviour is built by combining the elementary features. Some combinators are simple, for example **enables***(Clear,Silence)* describes the sequence of caller on-hook causing the called party to stop ringing. Its LOTOS translation is trivial:

```
process Enables [tel, ctl] (n1, n2 : Num) : exit (Num, Num, Result) :=
  OnHookFeat [tel] (n1, n2)                    (* on-hook behaviour *)
>> accept n1, n2 : Num, res : Result in        (* use current numbers *)
  StopRingFeat [tel] (n1, n2)                  (* stop ringing behaviour *)
endproc (* Enables *)
```

In functionality this resembles the translation of an elementary feature, but the process is named after the combinator. The gates are set to *tel* (the global gate) and *ctl* (a hidden control gate). It will be seen that *ctl* is used for global coordination of related calls, though for **enables** it is not actually needed. The translation of **enables** is its behavioural parameters in sequence (*Clear* corresponding to going on-hook, and *Silence* to stopping ringing).

The more useful combinators in ANISE are rather more complex than **enables**. For example **interrupts_after_try** is used to ensure that a caller goes on-hook only after a call has begun. More advanced combinators make full use of the flexible synchronisation operators in LOTOS. Several of these are written in the constraint-oriented style that allows restrictions on behaviour to be combined. The **interrupts_after_try** combinator, for example, is translated to two synchronised behaviours and 38 lines of LOTOS. Although LOTOS is good at describing complex combinations of behaviour, the intricacy of such a translation reflects the fact that the more advanced combinators are relatively powerful. They are also designed to work with arbitrary behaviour parameters since these may themselves be combinations. ANISE supports 20 general-purpose combinators, although only 10 of these are regularly used. There are also 13 telecommunications-specific combinators.

2.2.3 Profiles

Combinators are used to build up the behaviour of a single call (including features that work at the level of one call). A subscriber profile may be defined for each telephone number. At a minimum these must give each telephone number and its corresponding line identifier. Several

profiles may exist for the same line if it has several directory numbers (e.g. for Distinctive Ring). In this case, the first number for a line is considered to be the primary number. A profile may optionally define features for a number:

profile(684,27,
call_divert(207,BusyLine),**dial_code**(**3 ⇒296,**4 ⇒532),
ring_display,**ring_ preference**(2),**screen_ in**(415,532))

This states that number 684 is line 27. It has call diversion to number 207 on busy, abbreviated dialling codes **3 and **4, calling number display, distinctive ring pattern 2, and terminating call screening for calling numbers 415 and 532. The names used to define profile preferences are similar to the corresponding combinators (e.g. profile **screen_in** and combinator **screens_ in**). ANISE checks for parameter errors like two line identifiers for the same number and call forwarding loops. Although the latter is sometimes regarded as a feature interaction, it is detected statically in ANISE.

2.2.4 Global Level

Individual call behaviour is instantiated at the global level. Like any feature or combinator, each call process is parameterised by the pair of telephone numbers it connects. These are determined as the call progresses, and then erased when the call terminates and the process recurses. Calls operate independently of each other, but subject to global coordination. In LOTOS terms the overall structure is:

hide ctl **in** (Call [tel, ctl] (n1a, n2a) ||| Call [tel, ctl] (n1b, n2b) ||| ...) || Global [tel, ctl]

Although call coordination is called global, this does not mean that it is centralised. The global process represents the control distributed throughout the network. The global process synchronises with ordinary events in calls at the *tel* gate; for example it has to monitor lines becoming busy or free. User interactions with the network also occur at *tel*.

The global process also synchronises at the *ctl* gate when it is necessary to coordinate groups of calls. This applies particularly to features like Call Waiting and Three Way Calling. In fact the way LOTOS processes are combined embodies a rather fundamental decision about the nature of ANISE. The majority of present IN services are classified as Type A (single-ended, single point of control); they are endpoint-oriented. However it is anticipated that future IN services will include Type B and thus involve coordination of multiple endpoints or calls. ANISE was intentionally made call-oriented because it seems more natural to emphasise the call rather than the endpoint. ANISE is thus more attuned to Type B services, though Type A services have been successfully described.

Most calls follow the standard behaviour. However features like Alarm Call, Call Waiting and Call Completion to Busy Subscriber require an additional form of call. In such cases, a separate description of the special call is given using ANISE. An instance of a special call is created alongside normal calls, and is controlled like any other by the global process. Other LOTOS-based approaches such as [2] have a broadly similar specification structure at the outer level, but are endpoint-oriented rather than call-oriented.

2.3 Tool Support

The ANISE translator is written using the GNU *m4* macro processor. Although *m4* is just a macro processor, it is surprisingly flexible [9]. The advantage of *m4* relative to a conventional *lex*/*yacc* translator is that it has good facilities for text processing and translation. The translator is fairly robust and confirms the static correctness of an ANISE description. For example, it checks the functionality, pattern or property of combinator arguments to make sure

Feature		*m4*		ANISE Lines		LOTOS Lines	
abbrev.	meaning	macros	lines	call	profile	data	process
ABD	Abbreviated Dialling	4	17	1	1	43	1
ACB	Automatic Call Back	2	4	12	1	10	57
CCBS[a]	Call Completion to Busy Subscriber	0	0	1	0	0	14
CFBL	Call Forwarding Busy Line	2	3	1	1	58	1
CFU[b]	Call Forwarding Unconditional	2	5	1	1	9	1
CND	Calling Number Delivery	2	11	1	1	24	1
CW	Call Waiting	3	7	64	1	9	470
DR	Distinctive Ring	1	9	1	1	24	1
ONE	One Number	3	12	1	1	2	1
OCS	Outgoing Call Screening	3	12	1	1	24	1
POTS	Plain Old Telephone Service	200	2300	38	1	900	310
TCS	Terminating Call Screening	3	12	1	1	24	1

[a] The figures are the extra beyond ACB
[b] The figures are the extra beyond CFBL

Figure 2: ANISE Translator Statistics

that they make sense. Consider the behaviour combination **enable***(Speech,Clear)*. Since the *Speech* feature executes indefinitely, it is statically incorrect to expect this to enable clearing.

Figure 2 summarises some statistics about the translator. The notion of 'number of lines' excludes comments (which are generated automatically during translation so as to make the resulting LOTOS specification human-readable). A 'line' also conforms to the author's layout conventions, but is consistent across different features so that some comparison can be made.

Each feature is supported by a collection of *m4* macros that form part of the whole translator. For each feature, figure 2 gives the corresponding number of macros and their size in lines (excluding literal embedded LOTOS). This gives some idea of the size of the translator and its complexity in macro terms. The number of ANISE lines required to define a feature is also given. Two kinds of ANISE declarations are needed: to define a feature at the call or global level, and to equip a particular number with a feature in its profile. In most cases one declaration is needed for each. The ANISE statistics give some idea of how compact ANISE descriptions are. The figure also gives the size of the resulting LOTOS translation as lines of type definition and lines of process definition.

It may be noted that the translator contains a large number of relatively small macros. In fact the translator is very modular, dealing with different aspects of the translation (elementary features, general-purpose combinators, profile entries, etc.). Within modules, macros are self-contained and were developed largely in isolation. The translator is extensible by adding new modules and macros. The translator makes extensive use of macro processing: for example it makes over 24000 calls to user-defined macros when translating the POTS description into LOTOS. Although *m4* is a remarkably simple language, it can be used to achieve powerful effects. Translator development using *m4* is not, however, for the faint-hearted.

POTS is the largest element in figure 2. Since POTS is not built into ANISE and is just another described behaviour, it is convenient to show it as if it were a separate feature. For simplicity all the *m4* infrastructure is shown against POTS in the figure; of course this is available to all features. The data types generated for POTS provide a framework for other features, though these usually require some specific type definitions. The processes for POTS support the basic call behaviour that is then modified or extended by other features. As may be seen, the size of a generated LOTOS specification is manageable (around 2000 lines for the features in figure 2). LOTOS tools can cope with this size of specification, and so can human readers. The size expansion from ANISE to LOTOS is about 15 times, so it can be seen that

ANISE does indeed provide compact descriptions.

One of the problems with current LOTOS is that its data types are rather verbose. A new version of LOTOS is currently being standardised: E-LOTOS (Enhancements to LOTOS [5]). This offers significant improvements, including more convenient types. The author's experience is that E-LOTOS data types are more compact and readable than LOTOS data types. E-LOTOS also offers modularity, new operators and timing specification, all of which are likely to be useful in a future version of the translator.

3 Service Generation

3.1 Feature Addition

It is possible to write a single ANISE description of POTS and all its high-level features. However this is undesirable since it leads to monolithic specification and does not allow automatic identification of feature overlaps. ANISE describes each feature by its 'deltas' from POTS. That is, the additions, deletions and changes should be made explicit. This allows each feature to be described in isolation, and thus encourages modularity. More importantly, the alterations made to POTS by features can be checked for static (structural) compatibility. Alterations are given in the ANGEN (ANISE Generation) language. A feature is automatically integrated with POTS based on the directives in this language.

The **header** directive gives the name, date, etc. of the feature and is used for version control. The header is preserved by later transformations on the ANISE description so that an 'audit trail' can be maintained. The main directives state how POTS is to be changed. **change***(old,new,context)* gives the current ANISE text and its replacement; an optional context establishes where to make the change. Other kinds of alteration can be made with **append** (add new after old), **prefix** (add new before old), **delete** (remove old), **fill** (insert new into old) and **wrap** (place new round old). The **fill** directive is used to insert text as a parameter of an existing call. The **wrap** directive is used to treat existing text as a parameter of a new call.

In fact the editing directives are relatively smart. Because of the applicative (functional) style of ANISE, alterations take account of whether old or new text is a call or parameter. For example, **append** will notice if the old text is a combinator parameter and thus needs the new text as a parameter too. As a simple example, Distinctive Ring may be described as follows:

header(Distinctive Ring, Author, Date, ...)	% version header
fill(rings,rings_preference,Ringing)	% apply Distinctive Ring
prefix(% Global,profile(125,54))	% insert number 125/line 54
append(54,ring_preference(1),124)	% line 54, ring 1, number 124
append(54,ring_preference(3),125)	% line 54, ring 3, number 125

Distinctive Ring modifies ringing (combinator **rings**) by inserting selection of the ring pattern according to the called number profile (combinator **rings_preference**). This change is made in the context of the *Ringing* declaration. For test purposes, number 125 (line 54) is included as an alternative to number 124 for the same line; '*% Global*' defines a suitable point in the ANISE description of POTS to which the profile of a number may be prefixed. The ring pattern is then included in the profiles for relevant lines. The profile parameter **ring_preference** is appended after the line identifier (54) in the context of each number's profile (124, 125).

3.2 Static Interference Checks

ANGEN begins with some basic checks on the ANISE description. It warns the user if several features try to add the same declarations. For example, Automatic Call Back and Call Forwarding Busy Line both need the same definition of a ring-back call. Also, Call Completion

to Busy Subscriber and Call Waiting both need the elementary feature *Select* that accepts a digit to control the call. In these cases, ANGEN simply ignores the duplicate declarations.

ANGEN automatically checks for overlapping alterations to the same part of the description by different features. For example, two **change** directives for the same part of a description are clearly incompatible. Similarly **delete** and **change** for the same part are also incompatible. Most warnings concern interrelated insertions that may need to be applied in a specific order. When applying the features in figure 2, ANGEN produces warnings like:

```
Warning – Allowed overlapping changes of 'dials':
  Fill in ABD, CFU, OCS, ONE
```

It is up to the service engineer to decide whether such a warning needs action; ANISE merely reports an overlap. Although the detection of interaction is not automatic, it is helpful to have potential interactions highlighted out of the many possible feature combinations. As it happens, the warning above is significant since it is important to apply the features in the correct order when they modify dialling. Abbreviated Dialling must be applied first to the number dialled by the user; this is in case the user employs a short code like **3. One Number expansion must now be used for cases like FreePhone or an emergency number. Outgoing Call Screening must then be applied; using it first in the number translation might allow the user to bypass screening by using a short code. Call Forwarding Unconditional must be applied last so that Outgoing Call Screening is not bypassed. Not respecting this order of application would indeed lead to feature interactions. Interestingly, the possibility for interaction is identified *statically* by ANGEN from the nature of the structural alterations to POTS. Of course, human involvement is needed to determine that interactions *are* present, and to apply the features in the correct order.

ANGEN also reports the following warning:

```
Warning – Allowed overlapping changes of 'rings':
  Fill in CND, DR, TCS
```

This raises the question of whether Terminating Call Screening should be applied before Calling Number Delivery, i.e. is a banned number displayed on the called party's telephone? This depends on the interpretation of these features; in ANISE (as in most networks) a forbidden caller will not cause the called party to be rung, so the caller's number will not be displayed. There is also the question of whether Calling Number Delivery is compatible with giving a Distinctive Ring. Again this depends on the network implementation, but in ANISE (as in most networks) the two may be combined. Although it can be concluded that the above warnings are harmless, they nonetheless give pause for thought.

A further warning from ANGEN also deserves consideration:

```
Warning – Allowed overlapping changes of 'checks_busy':
  Fill in ACB, CCBS, CFBL, CW
```

The problem here is that all four features modify how line busy is handled (combinator **checks_ busy**). The overlap between Call Completion to Busy Subscriber and Automatic Call Back may not be serious. For the former, the user has the option to ask for a ring-back call if the called number is busy; for the latter, the ring-back is automatic. There is no problem if the network allows manual selection of ring-back even if it is automatic anyway; ANISE is tolerant in this respect. However, Call Forwarding Busy Line interacts with Automatic Call Back; ANISE presumes that CFBL should take precedence over ACB. However, suppose that the ultimate number after forwarding is also busy. Should the call be returned from the originally dialled number or from the forwarding number? Apart from the ambiguity, this could give rise to conflict with TCS. Finally, Call Waiting causes difficulties because it changes the meaning of line busy. For example, if a caller with ACB calls a busy line with CW, should the caller be held or should the call be returned? For the same situation, the called party may have rejected waiting on the current call; in this case a different action may be desirable. The warnings above therefore require the service engineer to think carefully about how to specify and combine these features.

3.3 Tool Support

Alterations defined in ANGEN are effected by a script that uses *perl*, since this has very flexible pattern matching and text substitution facilities. The ANGEN directives in a feature description are converted to *perl* using a translator written in *m4*. In fact the alterations given by all the feature descriptions are merged and applied as a single edit to the POTS base file. The *perl* and *m4* programs cooperate to detect the kinds of feature overlaps discussed in section 3.2.

4 Validation

4.1 Test Scenarios

The next stage using ANISE is to check the correctness of a feature description. Features can be validated on their own with POTS, or checked in conjunction with other features. In the interests of modularity, each feature has a separate file of validation checks. These are written in the language ANTEST (ANISE Test) that captures typical scenarios or use-cases. A feature is thus associated with two files: its alterations to POTS (ANGEN) and its test suite (ANTEST).

ANTEST makes it convenient to define feature tests, much as TTCN (Tree and Tabular Combined Notation [4]) was introduced for OSI. ANTEST also isolates the user from the specific language that is used to give semantics to ANISE. In the current implementation, ANISE tests are translated to LOTOS and evaluated by a LOTOS validator. Test results from the validator are then re-interpreted in terms of ANTEST, i.e. the user is not expected to understand a LOTOS view of test results (particularly if the test fails). This is consistent with the ANISE philosophy of not forcing formality onto the user. Because ANTEST and LOTOS are decoupled, tests could be translated to and executed by another formal framework such as SDL.

A test is given a name and a behaviour; the name is just for identification during testing:

test(OCS_Screened_Number,*TestBehaviour*) % test calls to screened numbers

A test defines a succession of signals between the user and the network:

send(685,OffHook) % 685 picks up
recv(685,OffHook,DialTone) % 685 hears dial tone

A **send** is when a signal is sent by the user to the network, a **recv** is when the user receives a signal from the network. The telephone number used is the first parameter. The name of the service primitive is the second parameter. Service primitive parameters may be optionally given. Note that ANTEST automatically infers the role of the service primitive from the context: technically, the signals above correspond to primitives *OffHook.Req* and *OffHook.Con(DialTone)*. However ANTEST allows a simpler test description to be given.

Signals can be sequenced flexibly to allow different kinds of tests. The simplest form is an acceptance test that runs through a sequence of signals. Here is a basic confidence check of POTS that confirms subscriber 124 can dial subscriber 162, speak once, and then hang up:

```
succeeds(                                   % successful sequence
  send(124,OffHook),                        % 124 picks up
  recv(124,OffHook,DialTone),               % 124 hears dial tone
  send(124,Dial,162),                       % 124 dials 162
  recv(124,Dial,RingTone),                  % 124 hears ringing tone
  recv(162,StartRing,NormRing),             % 162 starts normal ringing
  send(162,Answer),                         % 162 answers, stops ringing
  recv(124,Answer),                         % 124 hears ringing tone
  stop send(124,Speech,"Hello and Goodbye"), % 124 speaks message
  recv(162,Speech,"Hello and Goodbye"),     % 162 hears message
  send(124,OnHook),                         % 124 hangs up
  recv(162,OnHook),                         % 162 told to hang up
  send(162,OnHook))                         % 162 hangs up
```

If the network specification allows the sequence of signals as given, the test is said to pass. If some signal in the sequence is impossible, the test is said to fail; this would indicate an incorrect specification. Sometimes the test may pass on one path through the network behaviour but fail on another. This can arise if the network specification allows alternative behaviours, usually due to non-determinism. In such a case the test is said to be inconclusive. This is normally an undesirable situation that indicates an error in the specification.

Simple signal sequences can define useful tests. However, ANTEST allows more sophisticated possibilities. Since **succeeds** can be used to build only as the last part of a test sequence, more complex tests use **sequences** instead to build a chain of test signals. For example, the following allows the network to decide the ring pattern according to the called profile:

```
sequences(                                  % signal sequence
  send(780,OffHook),                        % 780 picks up
  recv(780,OffHook,DialTone),               % 780 hears dial tone
  send(780,Dial,684),                       % 780 dials 684
  recv(780,Dial,RingTone),                  % 780 hears ringing tone
  offers(                                   % alternative behaviour
    succeeds(                               % successful sequence
      recv(684,StartRing,NormRing),         % 684 starts normal ring
      send(780,OnHook)),                    % 780 hangs up
    succeeds(                               % successful sequence
      recv(684,StartRing,DistRing2),        % 684 starts ring pattern 2
      send(780,OnHook)))))                  % 780 hangs up
```

This is a test that allows the called number (684) to have either a normal ring pattern or distinctive ring pattern 2. (A better approach using the profile ring preference is mentioned later.) If a choice is to be made at the discretion of the user rather than the network, the **decides** directive may be used instead of **offers** to make a non-deterministic choice.

Since networks exhibit concurrent behaviour, some problems show up only if the test itself is concurrent. The following is taken from a test that interleaves calls to the same number so as to confirm that the busy condition is correctly handled in all cases:

```
interleaves(                                % interleaved sequences
  sequences(                                % signal sequence
    send(124,OffHook),                      % 124 picks up
    recv(124,OffHook,DialTone),             % 124 hears dial tone
    send(124,Dial,296), ...)                % 124 dials 296
  sequences(                                % signal sequence
    send(162,OffHook),                      % 162 picks up
    recv(162,OffHook,DialTone),             % 162 hears dial tone
    send(162,Dial,296), ...))               % 162 dials 296
```

As well as acceptance tests, it is often important to have rejection tests. These check that the network specification does not allow forbidden behaviour. A rejection test has two parts: the initial behaviour required to reach the critical point, and then the behaviour that should be rejected. Both parts may be simple sequences, but more complex tests with a choice or interleaving of signals are possible. A refusal test is introduced with **refuses**; the failure part is defined with **fails**. As an example, here is a test to check that a number (780) with Terminating Call Screening will not accept a call from a forbidden number (304):

```
refuses(                                    % refused sequence
  send(304,OffHook),                        % 304 picks up
  recv(304,OffHook,DialTone),               % 304 hears dial tone
  send(304,Dial,780),                       % 304 dials 780
  fails(                                    % failure sequence
    recv(304,Dial,RingTone)))               % 304 must not hear ringing tone
```

During validation of the features mentioned in this paper, the author found that *test* interactions were commoner than *feature* interactions! The problem is that tests of a feature may be invalidated by the presence of other features. For this reason it is possible to make a test dependent on the presence or absence of another feature. The parameters used in profiles may be used as values in signals:

```
recv(124,Dial,RingTone),                          % 124 hears ringing tone
send(196,StartRing,ring_preference(196))          % 196 starts ringing as per profile
```

or may be used in conditional tests:

```
depends(return_automatic(124),true,               % if 124 has automatic ring-back
  succeeds(                                       % then ...
    send(124,OnHook),                             % 124 hangs up
    recv(124,StartRing,NormRing), ...),           % 124 ring-back starts
  succeeds(                                       % else ...
    send(124,OnHook)))                            % 124 hangs up
```

4.2 Validating Features in Isolation

A feature can be tested in isolation by combining it alone with POTS. This allows the functionality of a feature to be checked in absolute terms. The tests associated with the feature are run automatically and the results are returned to the user:

```
Testing CFBL Normal Call ...      Pass   3 succ 0 fail 3.7 secs
Testing CFBL Forward 1 ...        Pass   3 succ 0 fail 7.3 secs
Testing CFBL Forward 2 ...        Pass   3 succ 0 fail 9.4 secs
Testing CFBL Opt. Forward 2 ...   Pass   3 succ 0 fail 7.2 secs
```

Due to specification parallelism there may be several ways to satisfy a test. The three successes reported above arise from the concurrent call limit of three used in the ANISE test. There is no technical problem to increase this limit, but the time to run the tests varies as the square of the call limit. The times given above were taken from tests using a 100 MHz processor. The most complex tests have non-determinism and concurrency, taking several minutes on this processor. Since three concurrent calls are sufficient to conduct nearly all important tests, the number is restricted so as to limit execution times. Adding more concurrent calls does not improve the detection power of tests (except in rare cases that are more like 'soak tests').

If a test fails or is inconclusive, ANTEST automatically determines the signal sequences that cause problems. These are reported using the notation of ANTEST; since a LOTOS validator is actually performing the test, the LOTOS interpretation of failure must be translated back into terms the user can understand. The following *erroneous* test run fails because the test does not expect ringing tone to follow dialling:

```
Testing POTS Dial Result ...      Fail   0 succ 3 fail 7.0 secs
  send(124,Offhook)
  recv(124,Offhook,DialTone)
  send(124,Dial,196)
  <failure point>
```

The outcome of a testing feature in isolation is confidence that the feature has been described correctly and is compatible with POTS at least.

4.3 Dynamic Interference Checks

The next stage of testing is to validate features in combination, i.e. adding the features of interest to POTS. Usually all features are combined for testing, but selected features may be checked for compatibility. The tests of each feature are run as before, but in the presence of other features. Some interactions may depend on the communicating subscribers having specific features enabled. For this reason some additional, multi-feature tests are defined among three numbers with all possible features. (Three numbers are required to detect multiple call interactions such as might happen with Call Forward, Call Waiting or Call Back.) These composite tests supplement the normal feature-by-feature tests.

Feature interaction is often taken to mean that other features change how a given feature works (usually undesirably). This shows up during validation because the feature fails to pass the tests that work in isolation. One possibility is that the feature is prevented from completing its task (i.e. the test deadlocks). In this case ANTEST will report the signal sequences that fail. Incorrect application of ABD and OCS, for example, will fail the test that shows call screening at work. Another possibility is that the presence of another feature results in non-determinism (i.e. the test result is inconclusive). For example, ACB and CFBL may conflict in this way because it is unclear whether to forward a call on busy or to ring back the caller.

Although ANTEST detects and reports such interactions, it does not explain why the interaction occurred (specifically, which features interacted). This is technically possible because the LOTOS validator can be asked to list behaviours that deadlock or exhibit non-determinism. However, this more detailed diagnosis is not yet supported by ANTEST.

Through formalising the features discussed in this paper, the author identified and corrected a number of feature interactions. Although this meant that a number of well-known interactions did not arise during validation, this is of course the point of a formal approach. Nonetheless, validation showed up some feature interactions that the author did not expect. Suppose that a number with ACB (791) calls a number that is busy (792), but that TCS forbids calls from this number (792 may not call 791). A test of TCS would not expect ring-back due to ACB. If this occurred, the failure might be reported as:

```
Testing Screen CallBack ...Inconclusive  1 succ 1 fail 10.7 secs
  send(792,Offhook)
  send(791,Offhook)
  recv(791,Offhook,DialTone)
  send(791,Dial,792)
  recv(791,Dial,SubsBusyTone)
  send(791,Onhook)
  send(792,Onhook)
  <failure point>
```

To cure this problem the ring-back feature had to be changed so that TCS was applied before accepting the request for ring-back.

4.4 Tool Support

ANTEST is also implemented by a translation to LOTOS using *m4*. The translator makes static checks on the validity of the tests. For example it is checked that a refusal test does indeed end with failure sequences, and that malformed tests like **succeeds***(...***succeeds***(...))* are forbidden.

Service primitive names in an ANTEST file are expanded to their correct LOTOS form according to the context (e.g. signal *OffHook* becomes LOTOS event *OffHookReq*). Parameter types (e.g. telephone number, voice message) are translated into their LOTOS representation (digit string, octet string) without the user having to know these.

The LOTOS tests are automatically combined with the LOTOS description derived by ANGEN from the ANISE description of POTS and the features. A *perl* script automates the running of the tests by controlling the LOLA (LOTOS Laboratory) simulator for LOTOS. This has a *TestExpand* command that evaluates a test in conjunction with the main specification. The LOLA test results are interpreted in a meaningful form for the user. For a failed or inconclusive test, LOLA is run a second time to generate the failure sequences. These are returned by LOLA in prefix normal form (choices among sequences) and re-interpreted in ANTEST terms.

5 Extending ANISE

Suppose that the service engineer wishes to define and evaluate a new feature. The stages required are as follows.

Combinator Definition: A new feature usually requires a new combinator. Such a combinator applies either to the basic call (if it operates at the level of an isolated call) or to the global call control (if it operates at the level of multiple calls). The existing feature library provides many examples of features that can be used to guide the definition of the new combinator. Of course the specifier must be familiar with *m4*, LOTOS and the library in order to define the combinator. As figure 2 shows, a new combinator typically requires 20 to 70 lines of description to be written (*m4* plus LOTOS). This is a relatively modest effort. The new combinator is added to the ANISE library, in a separate module if this is felt to be desirable. The combinator is likely to make use of existing *m4* macros in the library.

Special Call Definition: Some combinators at the global level require a special call to be defined. This can generally be done using ANISE without the need for new combinators.

Profile Definition: Most features require the addition of a new parameter to the subscriber profile. This is usually very easy (20 to 30 lines of *m4* plus LOTOS). The existing profile parameters are likely to be a good guide.

Integration Description: The way the feature is integrated with POTS is then defined in an ANGEN file. This is a straightforward and short task.

Test Description: Tests of the feature then have to be written in an ANTEST file. This is again straightforward, though a thorough sets of tests may take some effort.

Validation: The feature is first tested with POTS alone, then with all other features. Static checks by ANGEN may highlight problem areas for investigation. Dynamic checks by ANTEST may identify unexpected interactions.

ANISE provides a simple command-line interface to the tools. Having defined the ANGEN and ANTEST files for the new feature, the user simply gives the command `angen` and the names of the feature files. The merging of ANGEN files, the translation from ANISE to LOTOS, and the evaluation of tests proceed automatically. If desired, there are command-line options for interactive control and tracing of the procedure.

ANISE provides a substantial framework for the development of new features. For someone who is familiar with this framework (the author!) it is estimated that the time to specify and validate a new feature is from half a day to ten days, depending on its complexity. This is a small time to invest compared to the effort that telecommunications companies usually spend on development of a new feature (measured in person-years). The major advantage is that ANISE allows very early evaluation of a proposed new feature. The feature can be formalised and validated well before any commitment is made to its design and implementation. This, of course, is a major goal of service creation environments.

6 Conclusions

A strategy and tool for translating ANISE descriptions to LOTOS have been outlined. These give a formal basis to ANISE descriptions and allow them to be rigorously analysed. The ANGEN language and tool permit features to be described in isolation as alterations to the behaviour of POTS. It has been seen how these descriptions can be merged, with static warnings of potential interactions. The ANTEST language allows test scenarios to be defined for each feature. These tests can be applied to features in isolation or in combination, helping to discover flaws in feature descriptions and dynamic interactions between features. An indication has been given of how new features can be added to the ANISE library.

Future work areas include verification and graphical support. Current efforts have been directed towards validation because this is most practicable. But of course this has all the limitations of testing. The author intends to investigate verification in future. For example, it would be useful to know that a feature combined with POTS is behaviourally equivalent to that feature with POTS and all other features. It may be feasible to formulate properties of features and to show by model-checking that these properties are satisfied [8]. However the author has doubts about the practicality of this verification approach. A perhaps more workable idea is automated derivation of tests from their LOTOS specification, using the well-developed testing theory for LOTOS [1].

The ANISE, ANGEN and ANTEST languages and tools have proven themselves capable of describing and analysing realistic features. It has been shown how formality can be exploited without requiring the user to see the formalism. Although LOTOS is the current target language for the tools, it would be possible to consider other languages such as SDL. The effort required to use the approach is small compared to typical development manpower for features. The benefits of early feedback on problems are therefore felt to be worthwhile.

References

[1] Ed Brinksma. A theory for the derivation of tests. In Sudhir Aggarwal and Krishan K. Sabnani, editors, *Proc. Protocol Specification, Testing and Verification VIII*. North-Holland, Amsterdam, Netherlands, June 1988.

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

[3] 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.

[4] ISO/IEC. *Information Processing Systems – Open Systems Interconnection – Conformance Testing Methodology and Framework – Part 3: The Tree and Tabular Combined Notation (TTCN)*. ISO/IEC 9646-3. International Organization for Standardization, Geneva, Switzerland, 1991.

[5] 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.

[6] ITU. *Specification and Description Language*. ITU-T Z.100. International Telecommunications Union, Geneva, Switzerland, 1992.

[7] Jose A. Mañas, Tomas de Miguel Moro, Tomas Robles Valladares, Joaquín Salvachua, Gabriel Huecas, and Marcelino Veiga. TOPO user manual (version 3R6). Technical report, Department of Telematic Systems Engineering, Polytechnic University of Madrid, Spain, January 1995.

[8] 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.

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

[10] 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.

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

[12] Kenneth J. Turner. An architectural description of intelligent network features and their interactions. *Computer Networks and ISDN Systems*, 1998. To appear.