Realising architectural feature descriptions using LOTOS

Kenneth J. Turner

Computing Science and Mathematics, University of Stirling
Stirling FK9 4LA, Scotland
kjt@cs.stir.ac.uk

ABSTRACT. *ANISE (Architectural Notions In Service Engineering) can be used to describe a range of telecommunications services, including those from the Intelligent Network. The approach is supported by the ANGEN (ANISE Generator) language for combining features, the ANISE language for defining features, and the ANTEST (ANISE Test) language for defining validation scenarios. It is explained how all three are translated into LOTOS (Language of Temporal Ordering Specification), permitting any number of features to be validated in isolation or in combination. The toolset to permit this and the ANISE feature library are discussed. Typical results are presented from the automated analysis of telecommunications services.*

RÉSUMÉ. *ANISE (Architectural Notions In Service Engineering) est utilisé pour la description d'une gamme variée de services de télécommunication, incluant ceux du Réseau Intelligent. L'approche offre le langage ANGEN (ANISE Generator) pour la combinaison des services, le langage ANISE pour la description des services, et le langage ANTEST (ANISE Test) pour la définition des scénarios de test. L'article explique comment traduire ces trois langages en LOTOS (Language of Temporal Ordering Specification). Ceci permet la validation des services spécifiés, soit individuellement soit combinés les uns aux autres. Les outils impliqués de même que la bibliothèque de services d'ANISE sont décrits. Des résultats typiques provenant de l'analyse automatisée de services de télécommunication sont présentés.*

KEYWORDS: *Feature Interaction, Formal Methods, Intelligent Network, LOTOS (Language of Temporal Ordering Specification), Service Architecture, Service Feature, Telecommunications*

MOTS-CLÉS : *Architecture de service, interaction de services, LOTOS (Language of Temporal Ordering Specification), méthodes formelles, Réseau Intelligent, télécommunications*

1. Introduction

1.1. *Context*

A telecommunications service is a set of network capabilities forming a package that users can subscribe to. A telecommunications (service) feature is a network capability that forms part of a service. The distinction between services and features is more of a marketing issue than a technical issue. For example, Abbreviating Dialling might be provided as a service in its own right, or it may be provided only as part of another service such as Credit Card Calling. For this reason, the author prefers to talk about a feature as both a building block and what is available to a user. However in deference to common usage, the term service will also be used where it seems more appropriate.

Since the range of services differentiates network operators, there is a strong incentive to support an ever increasing set of features. It is also likely that service subscriptions will be a significant source of future revenue, particularly since the cost of providing basic calls continues to fall. However as has been known for some decades, adding new features is problematic because they may conflict with existing features. The Feature Interaction Problem has been a source of active research for some time. For example, [CAM 93, DER 92] classify a number of well-known interactions and place them within a general framework.

Methods for dealing with feature interactions may be broadly split into offline and online techniques. Among the offline techniques are those based on formal specification and analysis of feature sets. Many different formal languages have been used to specify features. However, approaches using LOTOS (e.g. [BOU 93, FAC 97, KAM 98, STÉ 95, THO 97]) seem to have been particularly popular. MSCs (Message Sequence Charts [ITU 96b]) have also inspired several approaches (e.g. [BER 97, BLO 97, DSS 97]). [AMY 99] proposes an approach for the structuring of features based on use case maps. CHISEL [AHO 98] is another example of a graphically-based approach to structuring service specifications. The author has also developed translations to LOTOS (and SDL) for CHISEL [TUR 00].

Another class of offline techniques is termed ‘architectural’ by the author. The emphasis here is on proper structuring of features. Indeed the author feels that a good theory for structuring and specifying features is an essential foundation. If features are defined in an *ad hoc* manner, it is hardly surprising if they overlap and conflict.

The IN (Intelligent Network [ITU 93c, ITU 93d, ITU 96a]) and the AIN (Advanced Intelligent Network) claim to offer an architecture for defining features. However, the author’s view is that these are rather bottom-up and oriented towards engineering issues. Instead, top-down service architecture merits attention. TINA (Telecommunications Intelligent Network Architecture, e.g. [DUP 95]) offers a much higher level framework for defining services. Feature interaction within the TINA context is discussed in [GAA 93, KOL 98, MUL 92, OHT 93, SIN 99]. ANSA (Advanced Network Systems Architecture [LIN 94b]) presents a comparable architectural framework. Other workers, including the author, emphasise the role of architecture in formal specification. For example, a building block approach is presented in [LIN 94a].

SCEs (Service Creation Environments, e.g. the tools of Ericsson, Hewlett-Packard, Marconi Communications, Nortel or Siemens) are common in industry for developing telecommunications services. SCEs usually employ proprietary SIBs (Service Independent Building Blocks [ITU 93b, ITU 97b]). They tend to focus on pragmatic issues since their goal is to generate actual code for network SCPs (Service Control Points). Unfortunately this means that SCEs are usually specific to a particular platform. They are rather implementation-oriented, creating potential problems of portability, interoperability, and maintainability. Current SCEs might be caricatured as being good at service engineering but weak on service analysis.

1.2. *ANISE*

The author has been developing an architecturally based approach called ANISE (Architectural Notions In Service Engineering). The following explains the evolution of ANISE and how it relates to this paper. Although ANISE leads to formal specification and rigorous analysis of features, it might fairly be said that its emphasis is on structuring services. That is, ANISE supports a theory for defining services in a rigorous manner, using a hierarchy of behavioural building blocks and their combinations. In ANISE all behaviours are called features, whether they be elementary (such as dialling a number) or complex (such as Three-Way Calling).

The initial work on ANISE attempted to create a more rational service/feature hierarchy [TUR 97c]. This showed the possibility for structuring services progressively from simpler features. The foundation of the approach was then laid in [TUR 97a]. This demonstrated that the Basic Call and variants of it could be modelled using surprisingly simple building blocks. By way of contrast, the IN uses a variety of approaches for constructing services. SIBs are rather variable in size and complexity. More importantly, the ones presented for the IN have not been shown to be sufficient or necessary for the range of IN services. The notion of GSL (Global Service Logic [ITU 93b, ITU 97b]) for combining SIBs is in need of elaboration. The IN also presents another way of constructing services using Basic Call models and Trigger Detection Points [ITU 93a, ITU 97a]. The call models are rather different in nature from the approach taken by SIBs. The call models are also rather implementation-oriented and reflect the services that have been deployed so far (mainly single-ended, single point of control). It is not clear that the models would be appropriate for more advanced kinds of services (e.g. with distributed control or multimedia). For this reason, the Basic Call is *constructed* in ANISE. Other service bases could equally well be defined in ANISE.

Once the architectural approach of ANISE was established, work began on translating ANISE service descriptions into a formal specification language. Although the emphasis in ANISE remains the structuring of services, a formal basis was essential if feature behaviour and feature interaction were to be rigorously assessed. The author's experience of architectural approaches to formal specification (e.g. [TUR 93, TUR 94a]) is that architectural descriptions can be just a few percent in size of the corresponding formal specifications. This is because the descriptions are written using concepts taken directly from the problem domain. For example, ANISE supports description using concepts such as telephone line, subscriber profile, dialling and telephone call. Someone writing in a formal language will generally have to specify such concepts in detail before moving onto higher-level, more interesting aspects.

Formal methods are largely unused in industry due to their perceived (or actual) difficulty. ANISE tries to have the best of both worlds. The ANISE user defines services using building blocks and their combinations that have an intuitive interpretation. However ANISE is supported by translation to a formal language and by rigorous analysis using tools for that language. ANISE goes to some trouble to hide the existence of the formal language in order to make the approach more acceptable to industrial engineers. Yet the formal basis gives the same precision to ANISE as an approach that directly uses a formal method. Whichever language is used, it is essentially hidden from the ANISE user.

Like several other workers, the author has found LOTOS to be a flexible language that can be used in a wide variety of domains. LOTOS was therefore the language of choice as the formal basis for ANISE. However other languages such as SDL (Specification and Description Language [ITU 96c]) would have been possibilities. SDL is much more acceptable to the telecommunications industry. However it would probably be rather harder to use SDL rather than LOTOS to support the constructs described in this paper. A start was therefore made on the translation to LOTOS [TUR 98a]. The main value in the approach at this point was detecting feature interactions statically as overlaps among features. In particular, features that modify the same aspects of the Basic Call are potential suspects for interaction. This does not definitely indicate interaction, but experience of using ANISE [TUR 98a] suggests that such overlaps are very often true interaction problems.

To discover feature interaction requires rigorous evaluation of features in combination. [TUR 98b] reports the next stage in developing ANISE: the rigorous analysis (simulation) of features. In isolation, the desired characteristics of a feature can be confirmed. When features are combined, their joint behaviour can be checked for deviations (interactions) from their individual behaviours. The approach requires a well-defined way of combining feature descriptions. ANGEN (ANISE Generator) acts a language for describing features in isolation and combining them with some base service such as POTS (Plain Old Telephone Service). It is also necessary to have a formal way of characterising the behaviour expected of a feature. The idea of use cases [JAC 92] inspired the language ANTEST (ANISE Test) that allows features to be validated individually or together. Although ANTEST supports feature validation rather than verification, this was a pragmatic choice that reflects current industrial practice.

[TUR 97a, TUR 98a, TUR 98b] focus on architectural and telecommunications oriented aspects of ANISE. The present paper looks at the detailed mechanisms that support ANISE, and consolidates the results of these papers. The overall approach is illustrated in section 2, that shows typical feature descriptions and how they are processed by the tools. Section 3 explains how ANGEN combines individual features to form a composite ANISE description, detecting static overlaps as it works. The translation from ANISE to LOTOS is presented in section 4. Section 5 shows how validation scenarios in ANTEST are translated into LOTOS and evaluated against the translated ANISE description. Appendix A gives summaries of the languages used in the article: ANISE, ANGEN, ANTEST and LOTOS.

2. Overview of approach

This section shows how a base service such as POTS can be described using ANISE. Feature description using ANGEN and feature validation using ANTEST are then discussed. The automated tools supporting the whole process are presented. It is not feasible to present a tutorial on the approach here. More detailed explanations appear in [TUR 97a, TUR 98b, TUR 98a].

2.1. *POTS description*

The ANISE language (Architectural Notions in Service Engineering) is used to describe a service, perhaps including some features. The language is applicative in style, with combinators applied to parameters to define new composite behaviour. For reference, a summary of ANISE is given in tables 3, 4 and 5 of the appendix.

As noted earlier, a Basic Call model is intentionally not built into ANISE. ANISE may thus be used to describe other kinds of services with a different basis. However IN services require POTS as a starting point. Figure 1 shows the approach to structuring the Basic Call from simpler behavioural elements. The innermost behaviours are instances of the elementary features. Various combinators (the outer ovals) progressively combine these, leading to three phases: *Dialling*, *Ringing* and *Answering*.

Although figure 1 is a convenient way of visualising the structure of an ANISE description, ANISE is a textual language. Figures 2 and 3 show the comparable description of POTS. The description is divided (for presentation only) into four parts: elementary features, basic call behaviour, subscriber profiles, and global behaviour.

Elementary features: The elementary features are the simplest form of behaviour that can be described in ANISE. They correspond to basic actions like dialling a number, answering a call, and hanging up. To illustrate how an elementary feature is defined, consider one example from figure 2. This is the action of seizing the line when going off-hook:

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

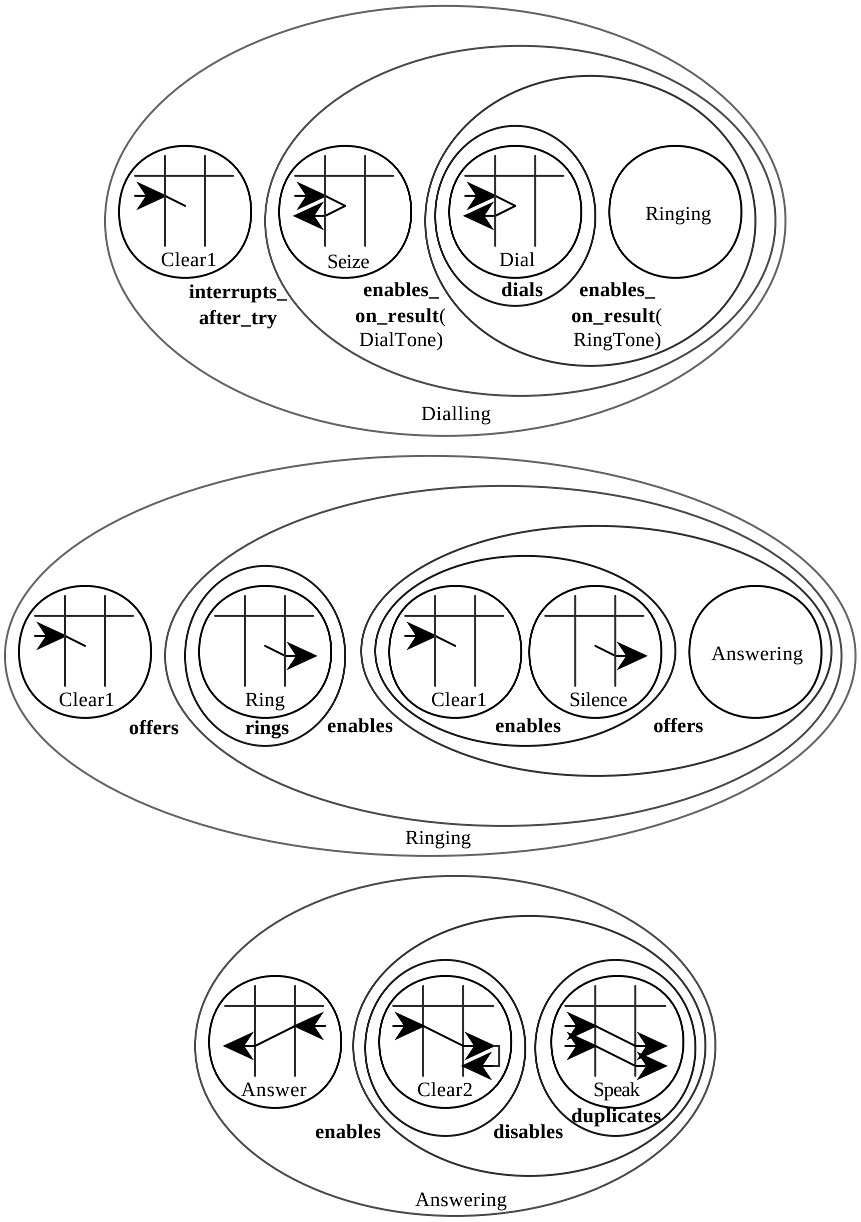

Figure 1. *Graphical Representation of Basic Call Structure in* A*NISE*

```
% Feature:
declare(Seize,                                        % seize the line
  feature(12,local_confirmed,single,OffHook,(CallingMess)))
declare(Dial                                          % dial number and connect
  feature(12,local_confirmed,single,Dial(Num),(CallingMess)))
declare(Ring,                                         % start ringing number
  feature(12,provider_initiated,single,StartRing))
declare(Silence,                                      % stop ringing number
  feature(12,provider_initiated,single,StopRing))
declare(Answer,                                       % answer call
  feature(21,unconfirmed,single,Answer))
declare(Speak,                                        % one-way speech
  feature(12,unconfirmed,ordered,Speech(Voice)))
declare(Clear1,                                       % clear by caller
  feature(12,user_initiated,single,OnHook))
declare(Clear2,                                       % clear by caller or called
  feature(12,asymmetric_confirmed,single,OnHook,OnHook))
% Call:
declare(Call,                                         % overall basic call
  loops(enables(Dialling,Ringing)))                   % dial phase, ring/answer phase
declare(Dialling,                                     % dial phase
  interrupts_after_try(                               % clear interupts after seize starts
    Clear1,                                           % caller clear
    enables_on_result(                                % seize then dial if dial tone
      Seize(00),                                      % seize line in fresh call
      DialTone,                                       % if seize got dial tone then ...
        enables_on_result(                            % dial then exit if ring tone
          dials(Dial),                                % dial selects destination
          RingTone,exit))))                           % if ring tone then 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
```

Figure 2. *ANISE call declarations for POTS*

```
% Profile:
profile(124,53) ...                    % number 124/id 53, etc.
% Global:
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 3. *ANISE global declarations for POTS*

The **declare** keyword introduces the name of the feature and its definition. In fact such a declaration defines a macro *Seize* that may be used multiple times later in the description. The **feature** keyword defines the parameters of an elementary feature. Its direction is between a pair of subscribers, numbered generically *1* (the caller) and *2* (the called). *12* means from subscriber 1 to subscriber 2 (subscriber 1 seizes the line in the above). Elementary features are classified according to their behaviour pattern. In the above case, **local_confirmed** means that a request by the subscriber is locally confirmed by the service provider (the telephone network). As with a telephone network, going off-hook normally leads to dialling tone from the local exchange. Elementary features also have an ordering property that states how repeated instances of the feature are related (e.g. strictly in sequence or overlapped). In the above case there is a **single** instance (the line is seized once per call).

There now follows a service primitive used to invoke the feature. This has a name (*OffHook* to seize the line) and may have parameters. By default the request that initiates a feature has the same name and parameters as other primitives that form part of the behaviour. However if acknowledgement of a request uses a different name or parameters, these can be given as the last part of a **feature** definition. When seizing the line, the acknowledgement is a signal that carries dialling tone (or similar). The *CallingMess* type covers all messages that may be sent to a calling user, and is supplied in the *Seize* acknowledgement (hence parentheses around *CallingMess* as a parameter).

Basic call: The Basic Call is built up from the elementary features. These are generally invoked just by citing their name (e.g. *Seize*). It is possible, however, to give as parameter the identification of the subscribers involved. For example, *21* would invoke the feature from subscriber 2 to subscriber 1. As a special case, *00* means no subscribers. *Seize* is instantiated this way since the call starts out unattached to any subscriber. The act of seizing the line will cause *Seize* to set the calling number.

In the top-level *Call* description, *Dialling* leads to *Ringing* (which leads to *Answering*). This behaviour is repeated indefinitely. Although a single declaration could be given for the Basic Call, it is clearer to present it in phases: *Dialling* (seizing the line, dialling the other party), *Ringing* (ringing the other party), and *Answering* (answering the call, speaking, and final hang-up). Each phase makes use of the elementary features. The ANISE combinators are used to build these into progressively more complex patterns. For example in *Call*, **enables** requires successful completion of *Dialling* before *Ringing* can begin. On termination of *Ringing* (and possible *Answering*), **loops** causes repetition of the whole behaviour.

Most combinators have a fairly straightforward interpretation: **disables** (permanent interruption), **duplicates** (behaviour repeated in both directions), **exit** (successful termination), **interrupts** (interruption and resumption of behaviour), **offers** (choice of behaviour). However services require some special combinators as well. Sometimes a particular result is needed before later behaviour is enabled; for example, **enables_on_result** requires *Seize* to obtain a *DialTone* result before dialling is permitted. The use of **interrupts_after_try** delays the possibility of *Clear1* (i.e. the caller hanging up) until *Seize* has been tried. The use of *Clear2* (i.e. both parties hanging up) is qualified by **collides**, which correctly handles both parties clearing simultaneously: their requests to clear the call can collide and mutually reinforce each other inside the network.

Subscriber profiles: ANISE supports explicit declaration of subscriber profiles. At the least, a profile gives the subscriber's telephone number and line identifier. This distinction is necessary since several numbers may be associated with the same line (e.g. for Distinctive Ring). As will be seen later, subscription to selected services is noted in the profiles. Although ANISE requires specific telephone numbers and line identifiers to be given, these are arbitrary and are used only for definiteness.

Global behaviour: As well as the behaviour of a single call, ANISE gives the overall behaviour of the network. This is partly because calls may interfere. For example **checks_busy** monitors whether lines are engaged or not, ensuring that calls to an engaged line receive busy tone. The maximum number of concurrent calls is also declared globally using **instances**. Finally, the **global** combinator takes the telephone service name (*tel*) and the top-level behaviour. This builds hierarchically on all the behaviour combinations, right down to the elementary features.

2.2. *Feature description*

ANISE follows the conventional approach of describing features as changes to POTS. In another service domain, features would be described as 'deltas' from a different basis. The ANGEN language (ANISE generator) is the means of stating such differences. For reference, a summary of ANGEN is given in table 6 of the appendix.

As an example, figure 4 shows how CW (Call Waiting) is defined using ANGEN. For convenience, the description is split into the same sections as for POTS. Figure 4 actually shows two languages: ANISE embedded in ANGEN. For clarity only the ANGEN keywords have been set in bold. A small complication for the tools is that ANISE comments as well as ANGEN comments are present (both '%'). The modifications carried out by ANGEN preserve the comments (and layout) of the embedded ANISE.

Since a feature is defined in terms of how it changes POTS, ANGEN is effectively an editing language that allows the root description to be modified. This does tie a

```
header(                                        % give feature header
 % cw.angen – Call Waiting, K. J. Turner, 3rd August 2000)
% Feature:
prefix(% Call,                                 % suspend/resume call
  declare(Recall,feature(21,user_initiated,single,Recall)))
prefix(% Call,                                 % dial digit to select action
  declare(Select,feature(21,user_initiated,single,Select(Dig))))
prefix(% Call,                                 % clear caller only
  declare(Clear3,feature(21,remote_confirmed,single,OnHook,OnHook)))
% Call Waiting:
append(% Call,                                 % define call waiting behaviours
  declare(CallWaitClear,                       % clear during call waiting
    offers(                                    % clear ends call waiting
      Clear1,                                  % caller clears
      enables(                                 % called clears
        Clear1(21),                            % called clears and then rings
        rings(Ring),                           % ring called again
        offers(Clear1,Clear3))))               % caller clears or is cleared
  declare(CallWait,                            % call waiting during speech
    disables(                                  % clear/recall interrupts speech
      offers(                                  % clear or recall/select
        CallWaitClear,                         % clear finishes call waiting
        enables(                               % recall, clear/select
          Recall,                              % called suspends call
          offers(                              % clear, select or recall
            CallWaitClear,                     % clear finishes call waiting
            enables_on_result(                 % select decides action
              Select,                          % called dials digit
              1,                               % if 1 dialled then ...
                offers(Clear1,Clear3),         % caller clears or is cleared
              2,                               % if 2 dialled then ...
                Clear1,                        % caller clears while held
              CallWait),                       % else call waiting continues
            enables(Recall,CallWait)))),       % call waiting continues
        duplicates(Speak))))                   % speech in both directions
wrap(disables,                                 % modify behaviour after answer
  waits(CallWait),Answering)                   % allowing call waiting
% Profile:
append('124,53',call_wait)                     % call waiting for num 124/id 53
% Global:
fill(checks_busy,waits_call)                   % put call waiting into busy check
```

Figure 4. *ANGEN declarations for Call Waiting*

feature description to the root description; the basis for any changes must be known. This is almost unavoidable in any such approach, but is fortunately not serious since the root service is generally well-defined and stable.

ANGEN is implemented using the very flexible capabilities of Perl for modifying strings. Nearly all editing operations have the form *alteration(old,new,context)*. The context can be omitted, meaning that the first instance of *old* in the base description is changed. This is sometimes ambiguous, in which case the *context* is used to define where to start searching for *old*. The modification operators are relatively smart in that they check whether the old or new text is a combinator call or a parameter. Parentheses around the parameters of a call are therefore respected automatically. Apart from the modification operators, **header** is used to give author, date, purpose, etc. This is preferable to an ANGEN comment since header contents are preserved in the generated ANISE and LOTOS, providing an 'audit trail' back to the original feature description.

Call Waiting introduces three new elementary features: *Recall* (pressing the recall button), *Select* (dialling a digit to control the call) and *Clear3* (caller only is cleared). The description of Call Waiting in figure 4 follows how it is normally implemented in the UK. When a subscriber is alerted by call waiting tone, pressing recall and then a digit indicates the desired action. Dialling 1 rejects the waiting call. Alternatively, Dialling 2 swaps the waiting call and the current call. If any other digit is dialled or the called party presses recall again, the current call is resumed. At any point the caller or the called party may hang up, causing the waiting call to be activated. Note that this procedure differs from that normally implemented in the US, where flash-hook is used to swap between calls. The UK approach avoids the ambiguity of flash-hook, which is also used to control Three-Way Calling in the US implementation.

The **prefix** operator inserts the new text before the old text. For Call Waiting, this is used for the new elementary features. The string '% Call' identifies where the Basic Call description begins, i.e. the end of the elementary features. In fact the new elementary features could be inserted anywhere reasonable; '% Call' merely preserves the conventional organisation of an ANISE description.

For convenience, the behaviour of Call Waiting is given as two separate declarations: *CallWait* for the main behaviour, and *CallWaitClear* for the action on clearing during Call Waiting. Each of these is introduced using the **append** operator that places the declarations just where the Basic Call behaviour starts. Again, the positioning of the declarations is just for consistency. At the level of a call, Call Waiting modifies the speech phase (in the behaviour defined by *Answering*). In the POTS description, clearing disables speech. The **wrap** operator envelopes this portion of the behaviour with the **waits** combinator that specifies Call Waiting. This combinator takes *CallWait* as the top-level definition of how Call Waiting works. Note that the introduction of **waits** corresponds to provisioning Call Waiting in the network. Individual subscription to Call Waiting is still necessary.

Features generally require subscriber profiles to be set up, since this is where subscriptions to a feature are recorded. In this example, it is supposed that subscriber

124 (line 53) has chosen Call Waiting. The profile for this subscriber is changed by appending **call_wait**, which is simply an indicator that Call Waiting is available. Since commas separate operator parameters, '124,53' is quoted to indicate that the embedded comma is literal text.

Features generally modify the Basic Call description. Some features also have an effect at a global level if they change the way that multiple calls are coordinated. This is the case for Call Waiting since it alters the meaning of a line being busy. An engaged line may receive a waiting call (it 'rings' with call waiting tone). In passing, note that a further call attempt when a call is waiting will genuinely receive line busy. The **fill** operator is the inverse of **wrap**; it inserts text within an existing combinator. In the case of Call Waiting, the **waits_call** combinator is inserted inside the use of **checks_busy** in POTS. The idea is that Call Waiting checks for line busy first, and passes this condition onto the usual busy check only if Call Waiting is not possible (because Call Waiting has not been subscribed to, or because waiting is already in progress).

The easiest way to understand the effect of ANGEN is to see how the declarations in figure 4 change those of figure 2 and 3. An outline of the new ANISE description appears in figure 5. (For readability, the layout and comments have been slightly modified from the actual ANGEN output.)

2.3. *Feature validation*

The ANTEST language (ANISE Test) is used to express validation scenarios. Such tests are currently defined manually using the designer's understanding of how a feature should work. There is therefore no formal coverage of a feature's behaviour, much as with other use case approaches. Testing theory (e.g. [JI 99, TRE 96]) allows rigorous generation of tests from specifications. This would be useful for deriving tests of actual feature implementations. However it would be pointless for ANTEST since it would merely produce tests of a feature specification from itself. Instead, intuitive expectations of features are validated. An alternative approach would be to derive tests from formally stated properties of features.

For reference, a summary of ANTEST is given in table 7 of the appendix. As an example of ANTEST, figure 6 presents a test of Call Waiting. The aim is to check that the originator of a call can successfully participate in Call Waiting. Other tests of Call Waiting check cases like the called party entering Call Waiting, controlling calls by dialling a digit or hanging up, etc.

The scenario in figure 6 involves subscribers 296 (Tom), 456 (Fiona) and 624 (Simon). Their names do not appear in the test, but they are introduced to make the following explanations more natural. Fiona, who has subscribed to Call Waiting, calls Tom and begins talking. Simon now calls Fiona. Although she is technically busy, Simon hears ringing tone while Fiona hears call waiting tone. Fiona asks Tom to hold, and switches to the waiting call. Simon now speaks to Fiona. Fiona swaps back to

```
header(pots.angen − Plain Old Telephone Service, K. J. Turner, 3rd August 2000)
header(cw.angen − Call Waiting, K. J. Turner, 3rd August 2000)
% Feature:
...                                     % elementary features as POTS
declare(Recall,...)                     % suspend/resume call
declare(Select,...)                     % dial digit to select action
declare(Clear3,...)                     % clear caller only
% Call:
declare(CallWaitClear,...)              % clear during call waiting as CW
declare(CallWait,...)                   % call waiting as CW
declare(Dialling,...)                   % dial phase as POTS
declare(Ringing,...)                    % ring phase as POTS
declare(Answering,                      % answer phase, POTS with CW
  enables(                              % answer, stop ring, speak, clear
    Answer,                             % answer call
    waits(                              % allow call waiting
      CallWait,                         % call waiting behaviour
      disables(                         % clear stops speech
        collides(Clear2),               % caller/called clear
        duplicates(Speak)))))           % speech in both directions
declare(Call,...)                       % overall basic call as POTS
% Profile:
profile(124,53,call_wait) ...           % num 124/id 53 with CW
% Global:
declare(MaxCalls,100)                   % number of concurrent calls
global(tel,                             % name for telephone service
  checks_busy(                          % monitor line free/busy
    waits_call(                         % call waiting busy check
      instances(MaxCalls,Call))))       % instantiate calls
```

Figure 5. *ANISE declarations for POTS modified by Call Waiting*

Tom but then decides to hang up, causing Tom to hang up as well. Fiona's telephone rings as Simon is reconnected. She picks up the call and speaks to Simon. Finally both of them hang up.

A test declaration gives its name and behaviours. The test in figure 6 should run successfully, so the **succeeds** operator is used. This takes a sequence of actions to be performed. A **send** action occurs when a subscriber sends a signal to the network, while a **recv** action corresponds to receiving a signal. Such actions give the subscriber's number and the name of the signal. Although signals correspond to service primitives, test descriptions are simplified by not having to explicitly state whether a primitive is a request, indication, response or confirm. This is automatically inferred from the ANISE context (or more exactly, using the LOTOS generated from ANISE). For example, the first action in figure 6 is actually *OffHook.Request*. The parameter of a primitive

```
test(CW_Originator_Swaps,                   % test originator call swap
  succeeds(                                 % successful sequence
    send(456,OffHook),                      % Fiona picks up
    recv(456,OffHook,DialTone),             % Fiona hears dial tone
    send(456,Dial,296),                     % Fiona dials Tom
    recv(456,Dial,RingTone),                % Fiona hears ringing tone
    recv(296,StartRing,NormRing),           % Tom starts ringing
    send(296,Answer),                       % Tom answers, stops ringing
    recv(456,Answer),                       % Fiona hears ringing tone stop
    send(456,Speech,"Hello Tom"),           % Fiona speaks message
    recv(296,Speech,"Hello Tom"),           % Tom hears message
    send(624,OffHook),                      % Simon picks up
    recv(624,OffHook,DialTone),             % Simon hears dial tone
    send(624,Dial,456),                     % Simon dials Fiona
    recv(624,Dial,RingTone),                % Simon hears ringing tone
    recv(456,StartRing,CallWaitTone),       % Fiona hears call waiting tone
    send(456,Speech,"Please hold Tom"),     % Fiona speaks message
    recv(296,Speech,"Please hold Tom"),     % Tom hears message
    send(456,Recall),                       % Fiona recalls exchange
    send(456,Select,2),                     % Fiona selects Simon
    send(456,Answer),                       % Fiona hears waiting tone stop
    recv(624,Answer),                       % Simon hears ringing tone stop
    send(624,Speech,"It's me, Simon"),      % Simon speaks message
    recv(456,Speech,"It's me, Simon"),      % Fiona hears message
    send(456,Recall),                       % Fiona recalls exchange
    send(456,Select,2),                     % Fiona selects Tom
    send(296,Speech,"This is Tom again"),   % Tom speaks message
    recv(456,Speech,"This is Tom again"),   % Fiona hears message
    send(456,OnHook),                       % Fiona hangs up
    recv(456,StartRing,NormRing),           % Fiona starts ringing again
    recv(296,OnHook),                       % Tom told to hang up
    send(296,OnHook),                       % Tom hangs up
    send(624,Speech,"Simon was waiting"),   % Simon speaks message
    recv(456,Speech,"Simon was waiting"),   % Fiona hears message
    send(456,Speech,"Sorry to keep you"),   % Fiona speaks message
    recv(624,Speech,"Sorry to keep you"),   % Simon hears message
    send(624,OnHook),                       % Simon hangs up
    send(456,OnHook)))                      % Fiona hangs up
```

Figure 6. *An ANTEST test scenario for Call Waiting*

```
test(TCS_Screened_Number,                   % test incoming call screened
  refuses(                                  % refused sequence
    send(624,OffHook),                      % Simon picks up
    recv(624,OffHook,DialTone),             % Simon hears dial tone
    send(624,Dial,296),                     % Simon dials Tom
    recv(624,Dial,BarredMess),              % Simon hears barred message
    fails(                                  % failure sequence
      recv(296,StartRing,NormRing))))       % Tom must not start ringing
```

Figure 7. *An ANTEST test scenario for Terminating Call Screening*

may optionally be given, so the second action is *OffHook.Confirm(DialTone)*. ANTEST accepts values of the ANISE built-in types. For example, *NormRing* (normal ring cadence) is a value of *CalledMess* (message issued to a called party). Speech in a call is represented by text strings. Although the control aspects of a call can be tested without giving explicit speech messages, for a feature like Call Waiting it is important to state which pair of parties is communicating. Some approaches do not model speech in calls, thereby omitting this crucial aspect.

Figures 6 and 8 are simple acceptance tests – they confirm that the network behaves as expected. The ANISE library also includes the more important refusal tests that checks the network will not do the unexpected. Figure 7 illustrates TCS (Terminating Call Screening) where Tom has barred calls from Simon. If Simon does call, then Tom's telephone must not start ringing. The **refuses** operator introduces behaviour that terminates in something forbidden. In this test, **fails** defines the action of Tom's telephone ringing as illegal. Although the last parameter of **refuses** is always forbidden behaviour, **fails** is used to declare any behaviour (possibly composite) as incorrect.

Figures 6 and 7 are also sequential tests – actions are performed in a fixed order. The ANISE library includes more challenging tests that involve concurrency. Indeed, certain kinds of problems are more likely to be uncovered by concurrent tests since they can exercise more of a feature's behaviour.

In figure 8, both Tom and Fiona try to call Simon simultaneously (as indicated by **interleaves**). The **sequences** operator declares actions without stating if they should be successful or not; this is separately stated using **succeeds** or **fails**. After dialling, there is a choice of outcomes (as indicated by **offers**): the call is put through and answered, or the caller receives subscriber busy tone. When this test is evaluated against the network specification, all possible interleavings will be checked. In fact the tests allows for a race condition: both Tom and Fiona may dial at the same time, and it is non-deterministic which call is put through. Note that this test merely allows simultaneous calls to the same number. Another variant of this test ensures that only one call at a time is put through.

ANTEST contains yet more operators for defining useful tests. For example, a test may be made conditional on whether a feature has been subscribed to. Thus **depends**(**screen_in**(296,624),...) will be executed if Tom has barred calls from Simon.

```
test(POTS_Interdependent_Calls,                % test competing calls
  interleaves(                                 % interleaved sequences
    sequences(                                 % event sequence
      send(296,OffHook),                       % Tom picks up
      recv(296,OffHook,DialTone),              % Tom hears dial tone
      send(296,Dial,624),                      % Tom dials Simon
      offers(                                  % alternative behaviour
        succeeds(                              % successful sequence
          recv(296,Dial,RingTone),             % Tom hears ringing tone
          recv(624,StartRing,NormRing),        % Simon starts ringing
          send(624,Answer),                    % Simon answers, stops ringing
          recv(296,Answer),                    % Tom hears ringing tone stop
          send(296,OnHook),                    % Tom hangs up
          recv(624,OnHook),                    % Simon told to hang up
          send(624,OnHook)),                   % Simon hangs up
        succeeds(                              % successful sequence
          recv(296,Dial,SubsBusyTone),         % Tom hears subscriber busy tone
          send(296,OnHook)))),                 % Tom hangs up
    sequences(                                 % event sequence
      send(456,OffHook),                       % Fiona picks up
      recv(456,OffHook,DialTone),              % Fiona hears dial tone
      send(456,Dial,624),                      % Fiona dials Simon
      offers(                                  % alternative behaviour
        succeeds(                              % successful sequence
          recv(456,Dial,RingTone),             % Fiona hears ringing tone
          recv(624,StartRing,NormRing),        % Simon starts ringing
          send(624,Answer),                    % Simon answers, stops ringing
          recv(456,Answer),                    % Fiona hears ringing tone stop
          send(456,OnHook),                    % Fiona hangs up
          recv(624,OnHook),                    % Simon told to hang up
          send(624,OnHook)),                   % Simon hangs up
        succeeds(                              % successful sequence
          recv(456,Dial,SubsBusyTone),         % Fiona hears subscriber busy tone
          send(456,OnHook))))))                % Fiona hangs up
```

Figure 8. *An ANTEST test scenario for POTS with interleaved calls*

2.4. *Tool support for analysis*

To be a practical (and industrially acceptable) technique, ANISE needs tool support. Figure 9 shows that the main tools are essentially hidden (within the light grey box). This emphasises the point that the use of a formal method is not visible. Furthermore, the internal language could be something other than LOTOS, such as SDL. The tool user creates one or more ANISE feature descriptions. For each feature description the tool user writes a collection of validation scenarios (use cases) to confirm the desired operation of the feature. These scenarios are fed into the toolset to automatically generate reports.

A static interference report indicates where features modify the same aspect of the Basic Call. This does not, in itself, mean an interaction – but it is a significant hint. Static interference identifies a potential problem that can be studied in more detail through simulation. Examples of static interference for a typical feature set are given later in this subsection and in section 3. The dynamic interference report indicates where the characteristics of a feature are changed when it is combined with other features. Examples of dynamic interference are given in section 5. Since ANISE hides the use of a particular formal language, all reports are translated from their internal form (LOTOS oriented) back into the form written by the user (ANISE, ANTEST). This makes the reports intelligible to the non-formalist.

Unlike some methods, ANISE allows *all* features to be checked in combination. It is therefore not restricted to checking, say, pairs of features. It could detect an interaction that involved many features. Although dynamic interference appears to subsume static interference, the latter is still interesting for two reasons. First, the static interference report identifies the specific features and where they overlap; the dynamic interference report simply states the point at which a feature fails to work. Static interference is therefore helpful in pinpointing the cause of interaction. Typically it helps to identify where features need to be prioritised relative to each other. Second, the report on dynamic interference is only as complete as the validation scenarios. Hopefully the scenarios give a realistic set of use cases. But it is possible for static interference to unearth a problem that it is not explicitly allowed for during validation. This would suggest that the test scenarios be extended.

To ensure portability, the ANISE tools are written using readily available languages. Overall tool control is embedded in Perl scripts (about 750 non-comment lines). The ANISE, ANGEN and ANTEST translators use the GNU *M4* macro processor [SEI 97]. Although *M4* is just a macro processor, it is surprisingly flexible [TUR 94b]. The advantage of *M4* relative to a conventional *lex*/*yacc* translator is that it has good facilities for text processing and translation. The parsing capabilities of *lex*/*yacc* would be of limited value to ANISE since the languages to be translated have a very simple structure.

The translators are relatively robust and check the static correctness of their input descriptions. The library contains around 60 tests that have been used to evaluate the standard features mentioned later in table 1. The tools detect the vast majority of static errors in an ANISE, ANGEN or ANTEST description, though a few types of errors can slip through to the translated LOTOS (where they cause an obvious error). *M4* allows the ANISE library to be written in a modular fashion. There are 12 modules, occupying about 8150 non-comment lines, that describe the combinators and operators of the languages. It is easy to extend the library with new modules, or to add new features within existing modules.

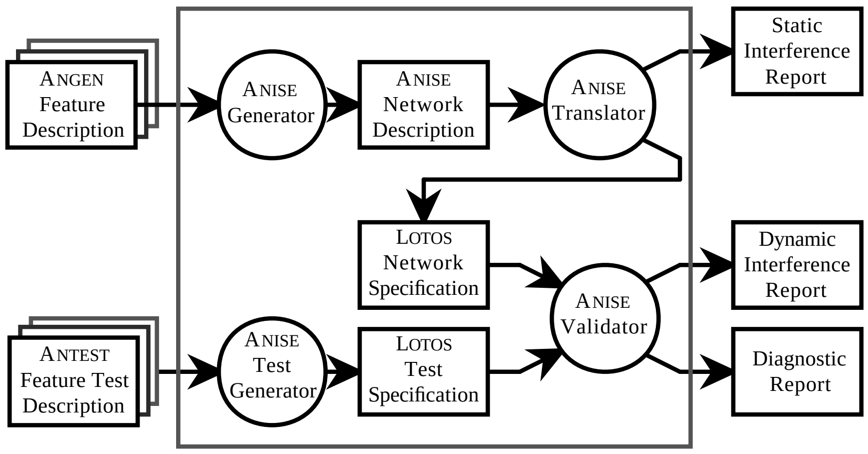
ANGEN Feature Description
ANISE Generator
ANISE Network Description
ANISE Translator
Static Interference Report
LOTOS Network Specification
ANISE Validator
Dynamic Interference Report
ANTEST Feature Test Description
ANISE Test Generator
LOTOS Test Specification
Diagnostic Report

Figure 9. *Tool Support for ANISE*

Although the ANISE validator seems to have the same status as the other tools, it is actually a self-standing toolset: TOPO and LOLA (LOTOS Laboratory [PAV 95]) developed by the Polytechnic University of Madrid. These provide a comprehensive set of tools for analysing LOTOS. In particular, LOLA has convenient facilities for checking a test process against a specification. LOLA also automates the procedure for introducing such a test at the top level of a specification. The *TestExpand* function of LOLA automatically checks for deadlocks and non-determinism (either of which is symptomatic of a feature interaction). CADP (Cæsar Aldébaran Development Package [FER 96]) would be an alternative LOTOS toolset to use. If ANISE used a different formal language, the ANISE validator would be replaced by a different tool (such as the Telelogic TAU/SDT validator and its associated toolset).

The user interface to the toolset is a simple command-line call like:

```
angen pots abd acb cfbl ccbs cnd cw dr one ocs cfu tcs twc
```

This combines the ANGEN and ANTEST files defining each feature, translates them to LOTOS, and runs the validator. The base description (POTS) must be given first. The features are applied in the order given, thus defining a relative priority where necessary. In fact only a partial ordering is required among the above features for correct prioritisation: ABD < ONE < OCS < CFU. The order of other features is unimportant. It would be straightforward to enforce prioritisation rules with ANGEN, but this is not done so that various feature combinations can be evaluated.

On the author's PC (a 450 MHz Pentium), the **angen** command above takes 8.5 minutes to complete. The tool output is (in part):

```
Testing ABD Normal Call          Pass   3 succ 0 fail 3.0 sec
Testing ABD Normal Call          Pass   3 succ 0 fail 3.0 sec
Testing ABD Abbreviated Call     Pass   3 succ 0 fail 1.1 sec
...
Testing POTS Interdependent Calls Pass 102 succ 0 fail 1.6 min
Testing POTS Independent Calls    Pass 120 succ 0 fail 2.9 min
```

This shows each test being run and its outcome. In most cases there are a number of alternative paths that lead to success of the test. The last few tests have a large number of success paths because of their high degree of concurrency.

The validation time is roughly proportional to the square of the number of concurrent call instances (three for the results above). Three concurrent calls serve to identify nearly all realistic feature interactions. The number of network subscribers, 13 in the tests above, was rather larger to allow for interesting combinations of features in the subscriber profiles. The validation time does not vary much according to the number of subscribers. The validation time does not vary too seriously with the number of features. For example, the time to validate POTS plus the first three features above (ABD, ACB, CFBL) is roughly doubled after adding all the other features. Validation in the presence of extra features is lengthened mainly by the extra success paths they permit (due to additional internal events). Since only some features (e.g. CCBS, CW, TWC) cause additional internal events, the approach scales to more complex networks.

Table 1 gives an overview of the results from using the toolset. Cells in the table contain a symbol where ANISE reports a problem for a pair of features. These (potential) interactions have been categorised here as: interaction is well-known, feature needs applied in the correct order, feature needs extension to handle aspects of other features, and potential interaction needs further investigation. A more detailed analysis appears in [TUR 98a]. Sample observations drawn from this table are as follows:

CF/CF: Call forwarding loops are detected statically when subscriber profiles are analysed by ANISE.

CND/CF: As an example of a potential interaction needing further consideration, it is necessary to decide if the number of the real caller or that of the forwarding line is delivered to the called party.

CW/CF: As an example of a well-known feature interaction, Call Forwarding on Busy Line conflicts with Call Waiting.

CW/CW: A commonly reported ambiguity with Call Waiting [CAM 93] does not arise in ANISE because cancelling the current call always returns to the waiting call even if it is on hold.

CW/TWC: A commonly reported ambiguity with flash-hook [CAM 93] does not arise in ANISE because Call Waiting follows the UK model (in which flash-hook is not used).

OCS/ABD: As an example of feature ordering, Originating Call Screening must be applied after Abbreviated Dialling to ensure that the full number is screened.

TWC/ABD: As an example of feature extension, Three-Way Calling must allow Abbreviated Dialling when the second leg of a call is established.

ABD	ACB	CCBS	CF	CND	CW	DR	OCS	ONE	TCS	TWC	
							>	?		+	ABD
		>	✓		✓			?	>	?	ACB
			?	?	>	?	?		?	✓	CCBS
				?	✓	?	✓	+	✓	+	CF
					?	?			>		CND
						?			>	✓	CW
								?	>		DR
								>		+	OCS
										+	ONE
										+	TCS
											TWC

✓	Well-known interaction	>	Feature ordering needed
+	Feature extension needed	?	Further consideration needed

ABD	Abbreviated Dialling	ACB	Automatic Call Back
CCBS	Call Completion to Busy Subscriber	CF	Call Forwarding (all forms)
CFBL	Call Forwarding on Busy Line	CFDA	Call Forwarding Don't Answer
CFU	Calling Forwarding Unconditional	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

Table 1. *(Potential) interactions among ANISE features*

3. Generating ANISE

ANGEN was discussed in section 2.2 and is summarised in table 6. As an example of ANGEN, the Call Waiting feature is defined in a file called *cw.angen* whose contents are shown in figure 4. The command-line call *angen cw* carries out the translation to ANISE. The *angen* tool is normally used to combine a number of features at once. One of the *M4* library modules contains definitions of the ANGEN operators. In fact the applicative syntax of ANGEN is also that of *M4*. The *angen* tool invokes *M4* on the ANGEN file, resulting in a temporary file of Perl editing commands. This is then used on the base description (*pots.angen*) to create a composite ANISE description of POTS with Call Waiting. The Perl commands report errors such as not being able to find the old text or its context.

ANGEN automatically checks for overlapping alterations by different features in the same part of the base description. For example, a **change** by two features is clearly incompatible. The check is achieved by recording which parts of the base description are modified in which way. It is safe to use the **fill** operator on a part of the base description that is altered by different features in other ways. If the **append** or **prefix** operator is used with other **append** and **prefix** operators in an identical context, a warning is issued that there are inconsistent alterations and the change is not made.

For other modification operators in an identical context, a warning of overlap is issued.

Most ANGEN warnings concern features that need to be prioritised relative to each other. For example when combining Abbreviated Dialling, Call Forwarding Unconditional, Outgoing Call Screening and One Number, ANGEN reports:

```
Warning - Allowed overlapping changes of 'dials':
  fill in ABD, CFU, OCS, ONE
```

All these change the **dials** behaviour. The ANISE user must decide if this warning needs action or not. As it happens, these features must be applied in the correct order since they all affect dialling. Abbreviated Dialling must be applied first to the number dialled by the user, expanding it to its full form. One Number must now be used to determine the actual number to be called. Outgoing Call Screening cannot be applied earlier in case Abbreviated Dialling or One Number expands to a barred number. Finally, Call Forwarding Unconditional must be applied after Outgoing Call Screening to avoid this being bypassed.

Not respecting this order of application would indeed lead to feature interactions. For example, a devious teenager could still call a barred number by defining an abbreviated code for it. Note that the possibility of interaction is identified *statically* by ANGEN from the nature of the structural alterations to POTS. However the ANISE user must still decide if the interaction is real and apply the features in the correct order.

When combining Calling Number Delivery, Distinctive Ring and Terminating Call Screening, ANGEN reports the following warning:

```
Warning - Allowed overlapping changes of 'rings':
  fill in CND, DR, TCS
```

All of these change the **rings** behaviour. Again this raises interesting questions. For example, should Terminating Call Screening be applied before Calling Number Delivery, i.e. should a barred number be displayed on the called party's telephone even if the call is not put through? Is it technically possible to combine Calling Number Delivery with Distinctive Ring? Probably the warnings are harmless, but they raise issues for consideration.

ANGEN also warns that Automatic Call Back, Call Completion to Busy Subscriber, Call Forwarding on Busy Line and Call Waiting may not be compatible:

```
Warning - Allowed overlapping changes of 'checks_busy':
  fill in ACB, CCBS, CFBL, CW
```

The warning identifies that all four features change the meaning of line busy (combinator **checks_busy**). Call Completion to Busy Subscriber and Automatic Call Back are probably compatible since a manual request to return the call can be subsumed by an automated return call. However, Call Forwarding on Busy Line interacts with Automatic Call Back. If the number obtained after call forwarding is also busy, should this or the originally dialled number return the call? If a caller with Automatic Call Back calls a busy line with Call Waiting, should the caller be held or should the call be returned? Again, questions like these require careful resolution.

Module	Purpose
anise	generic definitions and inclusion of other modules
anise_comb_gen	general-purpose combinators
anise_comb_tel	telecommunications service combinators
anise_feat	elementary features
anise_prim_int	internal types
anise_prim_par	service primitives
anise_prim_queue	service primitive queues
anise_prim_type	service types
anise_prof	subscriber profiles

Table 2. *ANISE combinator library modules*

In general, correct feature ordering has been ensured in ANISE by combining features in the required sequence. Feature extensions have been incorporated as needed in the definitions of the telecommunications combinators. Other problems have also been resolved during the design of ANISE, so that the ANISE feature library can be used without interactions.

4. Translating ANISE to LOTOS

This section explains how ANISE is translated to LOTOS. The overall shape of the generated specification is given first. Then it is shown how typical elementary features and combinators are translated. Finally, the global behaviour of the specification is discussed. Refer to section 2.1 and tables 3, 4, 5 of the appendix for an overview of ANISE. A summary of LOTOS syntax appears in table 8 of the appendix.

4.1. *LOTOS generation*

An ANISE description *network.anise*, probably generated by ANGEN from POTS and feature descriptions, is translated to LOTOS with the command-line call *anise network*. A small script translates the ANISE description to LOTOS using *M4*, automatically incorporating a number of library modules listed in table 2.

The overall shape of the generated LOTOS specification is as follows:

(* header detailing ANGEN modules that generated the specification *)
specification TelService [tel] : **noexit**
  library types
  service primitive parameter types
  service primitive types
  service profile types
  service primitive internal types
behaviour
  hide ctl **in** *top-level process*

```
where
   elementary feature processes
   combinator processes
   top-level processes
   test processes
endspec
```

The ANGEN description gives the name of service (e.g. *tel*). This is used to name the specification and the gate used to communicate with subscribers. An internal control gate *ctl* is used for coordination of calls as described in section 4.4.

The generated LOTOS contains a substantial number of data types used to support the rest of the specification:

Service primitive parameters: These are predefined types that may be used in an ANISE description. *CalledMess* defines messages to a called party (e.g. various ring patterns). *CallingMess* corresponds to messages for the calling party (e.g. various tones). *Dig* and *Num* define digits and telephone numbers; '#' and '*' are also allowed. *Voice* is used for speech messages. Because LOTOS does not have a built-in syntax for numbers and strings, telephone numbers and speech messages use a somewhat awkward syntax internally. A more conventional syntax is available to the ANISE user, and is translated into the internal LOTOS format.

Service primitives: A complex set of type definitions is generated for the service primitives used in calls. These primitives are not built in; they are declared as elementary behaviours in POTS and the features it is combined with. The corresponding data types are generated automatically. Each elementary feature gives rise to a set of internal operations and equations. A further data type allows queues of service primitives whose occurrences may overlap. When POTS is combined with the IN features in table 1, the service primitive data types amount to about 270 non-comment lines of LOTOS. These are mainly taken up with equational definitions of the internal operations.

Service profiles: Each feature has a data type for defining its use in a profile. For example *Abbrevs* defines abbreviated dialling codes, and *ScrIns* defines incoming numbers to be screened. The subscriber profiles are held as the global value *Profiles*. Since LOTOS does not allow global variables, *Profiles* is actually a constant (although generated on translation from the profile information in the ANISE description).

Service primitive internals: It is necessary to give a number of internal types not visible in an ANISE description. These include line identifiers, and associations of line identifiers or telephone numbers in calls. When a call is made between two subscribers, their line identifiers and telephone numbers are temporarily associated. Other internal types are used for control of calls and checking the results of combinators.

After the types come the processes. Each elementary feature declaration and each combinator usage gives rise to a process definition as described in sections 4.2 and 4.3. When ANTEST tests are translated to LOTOS, they appear at the end of the specification.

4.2. *Elementary features*

The following example of an elementary feature declaration was discussed in section 2.1:

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

This is translated into LOTOS as follows:

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

In fact a feature declaration generates two things: a process definition like the above and the means to call this process. In the context of a declaration like **declare***(Seize,...)*, *Seize* is defined as a potential call of the off-hook process. If *Seize* is used literally later in the ANISE description, it expands to an instantiation of the process above: *OffHookFeat [g1, g2] (n1, n2)*. However if logical subscribers are identified, as in *Seize(32)*, the number parameters are changed: *OffHookFeat [g1, g2] (n3, n2)*. For the special case of *Seize(00)*, which appears in figure 2 and is explained in section 2.1, the number parameters are supplied as a null value (*NoNum*).

An elementary feature operates between a pair of subscribers, and is parameterised by a gate and telephone number at each subscriber. Both gates are normally the service gate (e.g. *tel*); the use of the control gate is discussed in section 4.4. If subscriber 124 (line 53) goes off-hook, this corresponds to the LOTOS event *tel !53 !OffHookReq*. The process defining an elementary feature has a template that is determined by its pattern (**local_confirmed** in this case) and its ordering property (**single** in this case). In the library, the template is embedded in a macro for this particular pattern-property pair. (There are substantial similarities among many such macros.) The generated process is named after the service primitive (*OffHook* in this case), whose name is also be used as part of the definition (*OffHookReqKind* in the above).

A locally-confirmed service primitive has a request followed by a confirm. The feature process first accepts a primitive of kind off-hook request (*OffHookReqKind*) whose line identifier corresponds to the subscriber's number (*IsId*). Note that the process is parameterised by telephone numbers, but events use line identifiers. This is because several numbers may correspond to one line, and events happen on physical lines. Following the request, the feature process accepts a primitive that logically follows the previous one (*IsNextPrim*). The ordering of primitives depends on their pattern. For a locally-confirmed primitive, a request is followed by a confirm. The gate and line identifier for the confirm are the same as for the request.

Finally the feature process exits with a pair of telephone numbers and the result of the last primitive. A call starts out as a process with an undefined pair of associated telephone numbers. The appropriate elementary features bind these parameters to actual telephone numbers. Going off-hook fixes the calling number, while dialling fixes the called number. The *IsId* predicate used above allows for this. If the number is unbound then any line identifier may be used in an event, but the process will exit with a number corresponding to this. If the number is known, then only the corresponding line identifier may be used in an event. The subscriber profiles define the mapping from numbers to identifiers; the mapping may be many to one. The feature exit values include the result of the last primitive (*ResultOf*). This value is made available to combinators that depend on it. For example as discussed in section 2.1, **enables_on_result** requires an off-hook confirm to result in dialling tone before dialling is permitted.

More complex patterns have more complex templates. For example, a two-party clear (*Clear2* in figure 2) translates to:

```
process OnHookFeat1 [g1, g2]                  (* two-party clear ... *)
 (n1, n2 : Num) : exit (Num, Num, Result) :=  (* yields numbers/result *)
  g1 ?id1 : Id ?prim1 : Prim                  (* allow primitive if ... *)
    [IsKind (prim1, OnHookReqKind) and        (* on-hook request ... *)
      IsId (id1, n1)];                        (* for first number *)
  g2 ?id2 : Id ?prim2 : Prim                  (* allow primitive if ... *)
    [IsNextPrim (prim2, prim1) and            (* on-hook indication ... *)
      IsId (id2, n2)];                        (* for second number *)
  g2 !id2 ?prim3 : Prim                       (* allow primitive if ... *)
    [IsNextPrim (prim3, prim2)];              (* on-hook response *)
  exit (IdNum (id1, n1), IdNum (id2, n2),     (* yield numbers and ... *)
    NoRes)                                    (* no result *)
endproc (* OnHookFeat1 *)
```

If several elementary feature have the same primitive name (like *OnHook*), subsequent feature processes are numbered (*1* in the above case).

More complex properties such as **ordered** allow several invocations of an elementary feature to co-exist. The corresponding service primitive occurrences are queued and delivered in the order they happened. For example this applies to *Speak* in figure 2:

```
process SpeechFeat [g1, g2]                   (* one-way speech ... *)
 (n1, n2 : Num) : noexit :=                   (* never exits *)
  SpeechFeatA [g1, g2] (n1, n2)               (* speech in isolation *)
||                                            (* synchronised with ... *)
  SpeechFeatB [g1, g2] (n1, n2, <> )          (* speech ordering *)
where
  process SpeechFeatA [g1, g2]                (* speech in isolation ... *)
   (n1, n2 : Num) : noexit :=                 (* never exits *)
    g1 ?id1 : Id ?prim1 : Prim                (* allow primitive if ... *)
      [IsKind (prim1, SpeechReqKind) and      (* speech request ... *)
```

```
          IsId (id1, n1)];                       (* for first number *)
    (
        g2 ?id2 : Id ?prim2 : Prim               (* allow primitive if ... *)
          [IsNextPrim (prim2, prim1) and         (* speech indication ... *)
            IsId (id2, n2)];                     (* for second number *)
        stop                                     (* speech instance done *)
    |||                                          (* in parallel with ... *)
        SpeechFeatA [g1, g2] (n1, n2)            (* new speech instance *)
    )
  endproc (* SpeechFeatA *)
  process SpeechFeatB [g1, g2] (n1, n2 : Num,  (* speech ordering ... *)
   primq : PrimQueue) : noexit :=              (* has queue, never exits *)
    g1 ?id1 : Id ?prim1 : Prim                   (* allow primitive if ... *)
      [IsKind (prim1, SpeechReqKind)];           (* speech request *)
    SpeechFeatB [g1, g2] (IdNum (id1, n1), n2, (* repeat ordering ... *)
      prim1 + primq)                             (* adding new speech *)
  []                                             (* or ... *)
    g2 ?id2 : Id ?prim2 : Prim                   (* allow primitive if ... *)
      [IsNextQueue (prim2, primq)];              (* next in speech queue *)
    SpeechFeatB [g1, g2] (n1, IdNum (id2, n2)  (* repeat ordering *)
      RemPrev (prim2, primq))                    (* removing old speech *)
  endproc (* SpeechFeatB *)
endproc (* SpeechFeat *)
```

This follows the LOTOS constraint-oriented style [TUR 97b, VIS 91]. One constraint (*SpeechFeatA*) defines the basic pattern of speech as **unconfirmed**. The other constraint (*SpeechFeatB*) reflects the **ordered** property that requires a queue of primitives. '<> ' is the initially empty queue, '+' appends to the queue, and *RemPrev* removes a previous occurrence of a queued primitive. For yet more complex properties like **unreliable**, queue operations allow primitives to get out of order and even to be lost.

4.3. *Feature combinators*

A number of combinators translate very easily into LOTOS. For example, Call Waiting in figure 4 offers a choice of the caller the called clearing: **offers***(Clear1,Clear3)*. This becomes a LOTOS choice between instances of the corresponding elementary features:

```
process Offers [tel, ctl]                        (* choice of clearing ... *)
 (n1, n2 : Num) : exit (Num, Num, Result) :=     (* yields numbers/result *)
  OnHookFeat [tel, tel] (n1, n2)                 (* caller clears *)
[]                                               (* or ... *)
  OnHookFeat2 [tel, tel] (n1, n2)                (* called is cleared *)
endproc (* Offers *)
```

Like a feature process, a combinator process is parameterised by a pair of gates and numbers corresponding to the two subscribers. The process is named after the combinator (with a numerical suffix if there are several of the same one). Combinators, like features, also generate a process definition and yield the means of calling this process. A combinator call will therefore work with any parameters, whether invocations of elementary features or of other combinators. For example, figure 4 specifies a sequence of clearing by the caller, ringing, and finally a choice of clears: **enables**(Clear1(21),**rings**(...),**offers**(...)). The **enables** combinator takes the process calls generated by the nested constructs and uses them in sequence:

```
process Enables [tel, ctl]                    (* clear/ring/clear ... *)
 (n1, n2 : Num) : exit (Num, Num, Result) :=  (* yields numbers/result *)
  OnHookFeat [tel, tel] (n2, n1)              (* caller clears *)
>> accept n2, n1 : Num, res : Result in       (* and then ... *)
  Rings [tel, ctl] (n1, n2)                   (* called is rung *)
>> accept n1, n2 : Num, res : Result in       (* and then ... *)
  Offers [tel, ctl] (n1, n2)                  (* caller clears/is cleared *)
endproc (* Enables *)
```

In this way, the hierarchical structure of an ANISE description is translated into a hierarchical set of calls on LOTOS processes. Note that because the first feature is in the reverse direction (*21*), the numbers are swapped inside the combinator definition.

An important translation rule is that a combinator treats its parameters as calls of unknown processes, thus ensuring compositionality of translation rules. This considerably complicates translating some combinators. As an example, figure 2 deals with simultaneous clearing by both parties: **collides***(Clear2)*. The complication is to define **collides** without knowledge of the specific nature of the behaviours that are combined.

```
process Collides [tel, ctl]                   (* behaviour collision ... *)
 (n1, n2 : Num) : exit (Num, Num, Result) :=  (* yields numbers/result *)
  (
    (
      OnHookFeat1 [tel, tel] (n1, n2)         (* on-hook feature *)
    [>                                        (* disabled by ... *)
      exit (n1, n2, NoRes)                    (* numbers/no result *)
    )
  |||                                         (* interleaved with ... *)
    (
      (
        OnHookFeat1 [tel, tel] (n2, n1)       (* on-hook swapped *)
      >> accept n2, n1 : Num, res : Result in (* and then ... *)
        exit (n1, n2, res)                    (* numbers/given result *)
      )
    [>                                        (* disabled by ... *)
      exit (n1, n2, NoRes)                    (* numbers/no result *)
    )
  )
```

```
|[tel]|                                         (* synchronised with ... *)
  CollidesA [tel, ctl]                          (* possible collision *)
where

  process CollidesA [tel, ctl] :                (* allow for collision ... *)
   exit (Num, Num, Result) :=                   (* yields numbers/result *)
    tel ?id : Id ?prim : Prim;                  (* allow first primitive *)
    tel ?id : Id ?prim : Prim;                  (* allow second primitive *)
    (
      [IsReq (prim)] ->                         (* if request ... *)
        exit (any Num, any Num, any Result)     (* any numbers/result *)
    []                                          (* or ... *)
      [not (IsReq (prim))] ->                   (* if not request ... *)
        CollidesB [tel, ctl]                    (* any primitive *)
    )
  endproc (* CollidesA *)
  process CollidesB [tel, ctl] :                (* allow any primitive ... *)
   exit (Num, Num, Result) :=                   (* yields numbers/result *)
    tel ?id : Id ?prim : Prim;                  (* allow any primitive *)
    exit (any Num, any Num, any Result)         (* any numbers/result *)
  endproc (* CollidesB *)
endproc (* Collides *)
```

Again, this is constraint-oriented in style. The top-level behaviour interleaves the given behaviour (process *OnHookFeat1*, corresponding to declaration *Clear2*) in each direction. Either may be interrupted at any point by an **exit**, i.e. it may simply cease. Coordination of the two directions is handled by synchronisation with *CollidesA*. This initially allows a pair of primitives. With collision this will be a pair of requests, but without collision there will be a request followed by an indication. If the second primitive is a request then collision has occurred, and *CollidesA* forces an immediate exit. Otherwise there has been no collision, and behaviour proceeds as *CollidesB*. This simply allows the remaining behaviour to unfold and then to terminate successfully.

A further complication handled by *Collides* is that one or both of the numbers in the call may not yet have been fixed. The exits with **any** values allow these to be determined if necessary by the **collides** behaviour parameter. *Collides* also exits with whatever result the given behaviour yields.

The intention of combinators like **collides** and related ones is relatively easy to grasp. However as just seen, their formalisation is quite complex. This underlines the flexibility and appropriateness of LOTOS as the formal language chosen for ANISE. In other languages, such as SDL, the translation would be more complex and could probably not be generic in the sense that **collides** is.

The combinators illustrated above are general-purpose and could be used for other kinds of service. For example the author developed a similar approach called SAGE

(Service Attribute Generator [TUR 93]) for describing services in OSI (Open Systems Interconnection [ISO 94]). The telecommunications combinators listed in table 5 are translated in broadly similar ways as the generic combinators.

4.4. *Global behaviour*

The top level of an ANISE description is generally a call of **checks_busy**, translated into the LOTOS top-level behaviour:

```
process ChecksBusy [tel, ctl] : noexit :=           (* monitor line busy/free *)
  RingsBack [tel, ctl]                              (* normal/ring-back calls *)
||
  ChecksBusyA [tel, ctl] (<>  of IdAssocs,          (* busy checks *)
    <>  of IdAssocs, <>  of NumAssocs, <>  of NumAssocs)
where
  process ChecksBusyA [tel, ctl]                    (* busy checks *)
    (Calls, NoWaits : IdAssocs, Dials, Returns : NumAssocs) : noexit :=
    ...
  endproc (* ChecksBusyA *)
endproc (* ChecksBusy *)
```

Inside *ChecksBusy*, *ChecksBusyA* monitors the status of calls, particularly the busy/free state of lines. '<> ' here is an empty list. *ChecksBusyA* maintains a number of (global) status variables:

Calls: lists the associations between subscribers in current calls; it is simply a list of calling and called line identifier pairs

NoWaits: deals with lines where Call Waiting is no longer allowed because a subscriber has decided to cancel Call Waiting for the current call

Dials: lists pairs of numbers needed for Last Number Redial

Returns: lists pairs of numbers needed for Automatic Call Back

A significant challenge at the topmost level of behaviour is to make the combinators act compositionally. Unfortunately there are strong interrelationships among Automatic Call Back, Call Completion to Busy Subscriber, Call Forwarding on Busy Line and Call Waiting. Each of the corresponding combinators therefore inserts separate sections of LOTOS into the definition of *ChecksBusyA*. Where there are overlaps, the translation procedure combines these sections.

Another challenge is control of calls. Since call control is internal to the network it cannot be exerted through the normal LOTOS gate used for communication with subscribers (*tel*). Instead, all call processes have an internal gate (*ctl*) that can be used to manipulate calls. Internal interactions at the control gate are synchronous, so its introduction does not cause spurious deadlocks.

The control gate is used for features such as Call Waiting. A waiting call exchanges normal signals with its subscriber, but is not connected to the other party. Instead any events at the waiting end take place at the control gate. This is used to cause the connection to be cleared, say if the called party decides to reject the waiting call. The control gate is also used for internal signalling concerned with automatic redial and normal and ring-back calls.

ChecksBusyA synchronises with *RingsBack*. This represents regular subscriber-initiated calls as well as network-initiated normal and ring-back calls. *RingsBack* has the form:

```
   Instances [tel, ctl]                          (* normal calls *)
|||                                              (* interleaved with ... *)
   RingsBackA [tel, ctl]                         (* ring-back calls *)
```

Instances contains the required number of calls declared with the **instances** combinator. Each call is an instance of the Basic Call behaviour defined in ANISE. It starts out with no associated subscribers. When a subscriber goes off-hook, any free call instance synchronises with the subscriber and remains in use until the call is cleared. Then the call instance may be used again. The number of instances is therefore the concurrent capacity of the network; there may be more subscribers than call instances. However the number of ring-back call instances varies dynamically. When ring-back is required, a call instance is created dynamically. Ring-back calls are not subject to the **instances** limit, and disappear when the call is cleared.

Suppose that there are two call instances, one of which is currently idle and one of which connects number 124 (line 53) to number 456 (line 37). Suppose also that a ring-back call is pending from number 296 to number 456. The LOTOS process structure in this situation is equivalent to:

```
   (Call [tel, ctl] (NoNum, NoNum) ||| Call [tel, ctl] (124, 456))
||
   ChecksBusyA [tel, ctl] (< [53,37]> ,<> ,< [124,456]> ,< [296,456]> )
```

Since LOTOS does not have a built-in syntax for lists, < [...,...], ...> is used above as a shorthand for a list of pairs. A pair of associated numbers or line identifiers is built up gradually. When line 53 goes off-hook, an embryonic association *[53,NoId]* is created. When line 37 is called, this changes to *[53,37]*. Suppose that line 53 is later put on hold due to Call Waiting. In this situation, the association is changed to *[Held(53),37]*.

The first call instance above has no associated subscribers (*NoNum*). The second call instance above links numbers 124 and 456. Both call instances operate independently in parallel, but are synchronised with *ChecksBusyA* for call control. The parameters of this show that lines 53 and 37 are associated (the second call instance), Call Waiting has not been suspended by any subscribers, 124 last dialled 456, and a ring-back call from 296 to 456 is pending. When the call from 124 to 456 clears, 296 and 456 will be ‘claimed’ for the ring-back call. This will be created as a further call instance interleaved with the two regular call instances.

5. Translating ANTEST to LOTOS

ANTEST was introduced in section 2.3 and is summarised in table 7. ANTEST scenarios are translated quite readily to LOTOS. The command-line call *antest cw* translates the file *cw.antest* into LOTOS. As mentioned in section 2.3, the signal names used by **send** and **recv** are translated to their LOTOS equivalents. One small complication is translating the conventional syntax for decimal numbers and speech messages into an internal LOTOS syntax. A number is translated as a digit string, while a speech message is translated into a string of (ASCII) binary values. Number and string values could have been translated into internal constants, with possible savings in validation time.

As examples of ANTEST translations, figure 10 shows how figures 7 and 8 are translated. For readability, the conventional syntax for numbers is used here. Each test corresponds to a LOTOS process with the same name. A test process is parameterised by the standard service gate (*tel*) and one for indicating successful test completion (*OK*).

The actions of **send** and **recv** are translated to LOTOS events. The main difference is that the telephone number in an action is translated into the corresponding line identifier (as explained in section 4.2). A test with **refuses** is a sequence of events followed by a choice of the invalid behaviour (which leads to deadlock) or successful termination. The **interleaves** operator requires synchronisation only on the *OK* gate; events at the *tel* gate are indeed interleaved. The **sequences** operator simply gives events in sequence, while **offers** corresponds to LOTOS choice. The other test operators have straightforward counterparts. The only unusual operator is **depends**, which statically checks the subscriber profiles during test generation and produces test code as required.

All the test files are normally translated together as a list of LOTOS processes. These are combined automatically with the translation of the generated ANISE. This results in a single LOTOS file that is used for validation. The tests are then run automatically one by one. The *TestExpand* feature of LOLA internally modifies the specification to place the test process in parallel with the top-level behaviour. The composite behaviour is then exhaustively explored. If all paths lead to the success event, the test is said to be ‘must pass’. However if only some of the paths lead to success, the test is said to be ‘may pass’. If no paths lead to success, the test is said to fail. In ANTEST terms, these correspond to success, inconclusive and failure verdicts.

If a test does not succeed, ANTEST runs it a second time to determine all the paths that lead to failure. These are then printed out in a diagnostic report. Since there may be several equivalent paths leading to failure, the duplicates are eliminated in the report. The diagnostic report also converts the LOTOS syntax for events back into the form originally used by ANTEST. This avoids exposing the ANISE user to the vagaries

```
process Test_TCS_Screened_Number [tel, OK] : noexit :=
   tel ! 32 ! OffHookReq;       tel ! 32 ! OffHookCon (DialTone);
   tel ! 32 ! DialReq (296);    tel ! 32 ! DialCon (RingTone);
   (
      tel ! 27 ! StartRingInd (NormRing); stop
   []
      i; OK; stop
   )
endproc (* Test_TCS_Screened_Number *)

process Test_POTS_Interdependent_Calls [tel, OK] : noexit :=
   (
      tel ! 54 ! OffHookReq;    tel ! 54 ! OffHookCon (DialTone);
      tel ! 54 ! DialReq (296);
      (
         tel ! 54 ! DialCon (RingTone);
         tel ! 18 ! StartRingInd (NormRing);
         tel ! 18 ! AnswerReq; tel ! 54 ! AnswerInd;
         tel ! 54 ! OnHookReq;
         tel ! 18 ! OnHookInd; tel ! 18 ! OnHookReq;
         OK; stop
      []
         tel ! 54 ! DialCon (SubsBusyTone);
         tel ! 54 ! OnHookReq;
         OK; stop
      )
   |[OK]|
      tel ! 95 ! OffHookReq;    tel ! 95 ! OffHookCon (DialTone);
      tel ! 95 ! DialReq (296);
      (
         tel ! 95 ! DialCon (RingTone);
         tel ! 18 ! StartRingInd (NormRing);
         tel ! 18 ! AnswerReq; tel ! 95 ! AnswerInd;
         tel ! 95 ! OnHookReq;
         tel ! 18 ! OnHookInd; tel ! 18 ! OnHookReq;
         OK; stop
      []
         tel ! 95 ! DialCon (SubsBusyTone);
         tel ! 95 ! OnHookReq;
         OK; stop
      )
   )
endproc (* Test_POTS_Interdependent_Calls *)
```

Figure 10. *Sample translations of ANTEST scenarios*

of LOTOS.

Suppose that number 296 (Tom) has subscribed to Automatic Call Back. Tom has also chosen to bar calls from number 624 (Simon). Tom calls Simon when he is busy and then hangs up. When Simon is free, should the network call Tom back? Automatic Call Back would require this, while Terminating Call Screening would forbid it. A test that created this situation would be reported as follows:

```
Testing Screen CallBack     Inconclusive 1 succ 1 fail 7.4 sec
   send(624,Offhook)
   send(296,Offhook)
   recv(296,Offhook,DialTone)
   send(296,Dial,624)
   recv(296,Dial,SubsBusyTone)
   send(296,Onhook)
   send(624,Onhook)
   <failure point>
```

The choice of network actions leads to success in one case (the call is returned) and failure in the other case (the call is not returned). In fact this problem was discovered and cured during the development of ANISE by checking for call screening before honouring the request for ring-back.

Apart from non-determinism, feature interaction also appears as straight failure of a test:

```
Testing POTS Dial Check       Fail 0 succ 3 fail 4.8 sec
   send(124,Offhook)
   recv(124,Offhook,DialTone)
   send(124,Dial,196)
   <failure point>
```

Three failure paths are reported because three concurrent call instances are allowed.

6. Conclusion and discussion

It has been seen that ANISE offers facilities for structuring, describing, combining and validating features. The approach has been evaluated using POTS and a range of typical IN features. Through static and dynamic analysis, interesting interactions can be found – some of them well-known and some requiring further consideration. Although LOTOS has been successfully used as the internal formal representation, its existence is hidden from the tool user.

In principle this allows a different choice of internal language in future. SDL and conventional programming languages (e.g. C++ or Java) would be attractive possibilities, but there would be significant challenges in translating the ANISE constructs into these languages. ANISE was developed largely from architectural principles. In fact, it is a derivative of the earlier SAGE language (Service Attribute Generator [TUR 93]) used to express OSI services. It should be said, however, that ANISE was developed hand-in-hand with its translation to LOTOS. A number of the generic ANISE combinators are rather LOTOS-like in style (e.g. **disables**), but a number are not (e.g. **interrupts**). The telecommunications combinators were inspired purely by the IN, and so are not LOTOS-oriented. Most of the combinators could be represented readily in another language. However, as seen in section 4.3 some of the combinators (e.g. **collides**) are complicated to translate. The constraint-oriented style in LOTOS works very well for these. In a more 'conventional' language, such combinators would probably have to be translated using a set of case-specific templates. The advantage of the LOTOS-based approach is that combinators can be translated almost independently of their (behavioural) parameters.

The ANISE language allows a range of typical telecommunications features to be described. ANISE derives its flexibility from a set of simple building blocks (elementary features) and ways of combining them (generic or telecommunications combinators). In this sense, ANISE supports an architecture for defining services. Although POTS and the Basic Call form the basis for IN features, the ANISE approach is more general than this and could be used for other kinds of services. Being derived from SAGE, ANISE can certainly be used for OSI-like services. ANISE is also deliberately designed to handle a wide range of IN features, and not just single-ended single-point-of-control features. Specifically, ANISE is targeted at defining the relationship among a number of calls (e.g. as required for multiway calls or conferences).

ANISE also takes a rather high-level view of services as seen by communicating users. It should therefore be possible to model other kinds of services such as arise in TINA or in multimedia systems. ANISE would handle continuous bit streams as it currently handles speech – as streams of packetised data. Issues such as synchronisation and session control could well require new ANISE combinators. Service management, such as billing and service subscription, are not currently dealt with in ANISE. There is a deliberate focus on ordinary user behaviour, so management would require an extension of the model.

Since ANISE is a textual language, it is less attractive to industrial users. A prototype tool has been developed for giving ANISE descriptions in a graphical form resembling figure 1. The textual form of ANISE can be derived automatically from this. The only reason for considering a graphical syntax is to improve comprehensibility – particularly for novice users. A graphical description can provide a good overview of structure, as is hopefully evident from figure 1. The textual form of ANISE, seen in figure 2, does not show the overall relationship between behavioural elements so well.

The LOTOS specifications generated by ANISE are quite readable (by those familiar with LOTOS!). This is helped by the neat layout and the automatically generated comments (which also provide links back to the ANISE source). However the extensive use of abstract data types and constraint-oriented composition makes the specifications somewhat remote from a programmatic implementation. The intention was, of course, to represent feature behaviour compactly and to facilitate simulation and analysis.

The ANGEN language allows features to be defined in isolation and then to be automatically combined with a base description. ANGEN generates reports of static overlaps among features as a by-product of combing features. Typically these overlaps require prioritisation or extension of features.

The ANTEST language allows validation scenarios (use cases) to be defined for features. These are automatically applied, and allow features to be evaluated in isolation or in combination. It is possible to check all features simultaneously, and thus to discover arbitrary *n*-way interactions. ANTEST looks even more like LOTOS, yet it essentially expresses sequence, choice and concurrency. It should therefore be feasible to translate most of ANTEST to other languages. The main difficulty would be in handling concurrency (awkward in SDL, threads in Java). Refusal tests would be expressed by actions that led to an internal error. MSCs (Message Sequence Charts [ITU 96b]) could be used as an alternative to ANTEST, thus connecting to a well-known formalism. Yet, ANTEST has the advantage of being closed tied in with ANISE (e.g. the use of shorthands for signal names). A graphical approach could also be used with ANTEST.

Note that ANTEST merely *validates* rather than *verifies* features, so it would be useful to derive rigorous tests automatically. The author has shown [JI 99] that techniques for protocol test derivation are more generally applicable, and should be adaptable for telecommunications services. It would also be useful to define symbolic tests that do not rely on particular subscribers and particular profiles. There is some prospect of achieving this using LOLA. Using an ANISE specification to generate useful feature implementation tests is an area for further study.

Acknowledgements

The author warmly thanks the anonymous referees for a careful and insightful appraisal of the draft article. Jean-Michel Hufflen (Université de Franche-Comté) kindly provided technical support for the typesetting.

7. References

[AHO 98] AHO A. V., GALLAGHER S., GRIFFETH N. D., SCHELL C. R., SWAYNE D. F., "SCF3/Sculptor with Chisel: Requirements Engineering for Communications Services", in: *Proc. 5th. Feature Interactions in Telecommunications and Software Systems* (ed. by KIMBLER K., BOUMA W.), pp. 45–63, IOS Press, Amsterdam, Netherlands, September 1998.

[AMY 99] AMYOT D., BUHR R. J. A., GRAY T., LOGRIPPO L. M. S., "Use Case Maps for the Capture and Validation of Distributed Systems Requirements", in: *Proc. 4th. IEEE International Symposium on Requirements Engineering*, pp. 44–53, Institution of Electrical and Electronic Engineers Press, New York, USA, June 1999.

[BER 97] BERGSTRA J., BOUMA W., "Models for Feature Descriptions and Interactions", in: *Proc. 4th. International Workshop on Feature Interactions in Telecommunication Networks* *and Software Systems* (ed. by Dini P., Boutaba R., Logrippo L. M. S.), pp. 31–45, IOS Press, Amsterdam, Netherlands, June 1997, ISBN 90-5199-3471.

[BLO 97] Blom J., "Formalisation of Requirements with Emphasis on Feature Interaction Detection", in: *Proc. 4th. International Workshop on Feature Interactions in Telecommunication Networks and Software Systems* (ed. by Dini P., Boutaba R., Logrippo L. M. S.), pp. 61–77, IOS Press, Amsterdam, Netherlands, June 1997, ISBN 90-5199-3471.

[BOU 93] Boumezbeur R., Logrippo L. M. S., "Specifying Telephone Systems in Lotos", *IEEE Communications Magazine*, pp. 38–45, August 1993.

[CAM 93] Cameron E. J., Griffeth N. D., Lin Y.-J., Nilson M. E., Schnure W. K., Velthuijsen H., "A Feature-Interaction Benchmark for IN and Beyond", *IEEE Communications Magazine*, pp. 64–69, March 1993.

[DER 92] Derrett N., "Classification of Feature Interactions", in: *Proc. 1st. International Workshop on Feature Interactions in Telecommunications Software Systems* (ed. by Velthuijsen H., Griffith N., Lin Y.-J.), pp. 153–154, Florida, USA, December 1992.

[DSS 97] Dssouli R., Somé S., Guillery J.-W., Rico N., "Detection of Feature Interactions with REST", in: *Proc. 4th. International Workshop on Feature Interactions in Telecommunication Networks and Software Systems* (ed. by Dini P., Boutaba R., Logrippo L. M. S.), pp. 271–283, IOS Press, Amsterdam, Netherlands, June 1997, ISBN 90-5199-3471.

[DUP 95] Dupuy F., Nilsson G., Inoue Y., "The Tina Consortium: Towards Networking Telecommunications Information Services", *IEEE Communications Magazine*, pp. 78–83, November 1995.

[FAC 97] Faci M., Logrippo L. M. S., Stépien B., "Structural Models for specifying Telephone Systems", *Computer Networks*, Vol. 29, no. 4, pp. 501–528, March 1997, Elsevier Science Publishers.

[FER 96] Fernández J.-C., Garavel H., Kerbrat A., Mateescu R., Mounier L., Sighireanu M., "CADP (Cæsar/Aldébaran Development Package): A Protocol Validation and Verification Toolbox", in: *Proc. 8th. Conference on Computer-Aided Verification* (ed. by Alur R., Henzinger T. A.), no. 1102 in Lecture Notes in Computer Science, pp. 437–440, Springer-Verlag, Berlin, Germany, August 1996.

[GAA 93] Gaarder K., Audestad J. A., "Feature Interaction Policies and the Undecidability of a General Feature Interaction Problem", in: *TINA IV*, pp. II.189–II.200, September 1993.

[ISO 94] ISO/IEC, *Information Processing Systems – Open Systems Interconnection – Basic Reference Model*, ISO/IEC 7498, International Organization for Standardization, Geneva, Switzerland, 1994.

[ITU 93a] ITU, *Intelligent Network – Distributed Functional Plane for Intelligent Network Capability Set 1*, ITU-T Q.1214, International Telecommunications Union, Geneva, Switzerland, 1993.

[ITU 93b] ITU, *Intelligent Network – Global Functional Plane for Intelligent Network Capability Set 1*, ITU-T Q.1213, International Telecommunications Union, Geneva, Switzerland, 1993.

[ITU 93c] ITU, *Intelligent Network – Q.120x Series Intelligent Network Recommendation Structure*, ITU-T Q.1200, International Telecommunications Union, Geneva, Switzerland, 1993.

[ITU 93d] ITU, *Intelligent Network – Q.121x Series Intelligent Network Recommendation Structure*, ITU-T Q.1210, International Telecommunications Union, Geneva, Switzerland, 1993.

[ITU 96a] ITU, *Intelligent Network – Q.122x Series Intelligent Network Recommendation Structure*, ITU-T Q.1220, International Telecommunications Union, Geneva, Switzerland, 1996.

[ITU 96b] ITU, *Message Sequence Chart (MSC)*, ITU-T Z.120, International Telecommunications Union, Geneva, Switzerland, 1996.

[ITU 96c] ITU, *Specification and Description Language*, ITU-T Z.100, International Telecommunications Union, Geneva, Switzerland, 1996.

[ITU 97a] ITU, *Intelligent Network – Distributed Functional Plane for Intelligent Network Capability Set 2*, ITU-T Q.1224, International Telecommunications Union, Geneva, Switzerland, 1997.

[ITU 97b] ITU, *Intelligent Network – Global Functional Plane for Intelligent Network Capability Set 2*, ITU-T Q.1223, International Telecommunications Union, Geneva, Switzerland, 1997.

[JAC 92] JACOBSON I., *Object-Oriented Software Engineering — A Use Case Driven Approach*, Addison-Wesley, Reading, Massachusetts, USA, 1992.

[JI 99] JI HE, TURNER K. J., “Protocol-Inspired Hardware Testing”, in: *Proc. Testing Communicating Systems XII* (ed. by CSOPAKI G., DIBUZ S., TARNAY K.), pp. 131–147, Kluwer Academic Publishers, London, UK, September 1999.

[KAM 98] KAMOUN J., LOGRIPPO L. M. S., “Goal-Oriented Feature Interaction Detection in the Intelligent Network Model”, in: *Proc. 5th. Feature Interactions in Telecommunications and Software Systems* (ed. by KIMBLER K., BOUMA W.), pp. 172–186, IOS Press, Amsterdam, Netherlands, September 1998.

[KOL 98] KOLBERG M., MAGILL E. H., “Service and Feature Interactions in TINA”, in: *Proc. 5th. Feature Interactions in Telecommunications and Software Systems* (ed. by KIMBLER K., BOUMA W.), pp. 78–84, IOS Press, Amsterdam, Netherlands, September 1998.

[LIN 94a] LIN F. J., LIN Y.-J., “A Building Block Approach to detecting and resolving Feature Interactions”, in: *Proc. 2nd. International Workshop on Feature Interactions in Telecommunications Systems and Software Systems* (ed. by BOUMA L. G., VELTHUIJSEN H.), pp. 86–119, IOS Press, Amsterdam, Netherlands, 1994.

[LIN 94b] VAN DER LINDEN R., “Using an Architecture to help beat Feature Interaction”, in: *Proc. 2nd. International Workshop on Feature Interactions in Telecommunications Systems and Software Systems* (ed. by BOUMA L. G., VELTHUIJSEN H.), pp. 24–35, IOS Press, Amsterdam, Netherlands, 1994.

[MUL 92] MULDER H., JANMAAT R., “Service/Feature Interaction Resolution Framework (Guidelines)”, in: *TINA III, Narita, Japan,* pp. G1.1.1–G1.1.7, PTT Research, Dr. Neher Laboratories, 2260 AK, Leidschendam, The Netherlands, January 1992.

[OHT 93] OHTA T., TAKAMI K., TAKURA A., “Acquisition of Service Specifications in Two Stages and Detection/Resolution of Feature Interactions”, *TINA IV*, pp. 173–187, September 1993.

[PAV 95] PAVÓN GOMEZ S., LARRABEITI D., RABAY FILHO G., “LOLA User Manual (Version 3R6)”, Technical Report, Department of Telematic Systems Engineering, Polytechnic University of Madrid, Spain, February 1995.

[SEI 97] SEINDAL R., “GNU *m4* (Version 1.4)”, Technical Report, Free Software Foundation, 1997.

[SIN 99] SINNOTT R. O., KOLBERG M., “Engineering of Interworking TINA-based Telecommunication Services”, in: *Proc. TINA’99 Conference* (ed. by SVENTEK J.), Oahu, Hawaii, April 1999.

[STÉ 95] STÉPIEN B., LOGRIPPO L. M. S., “Feature Interaction Detection Using Backward Reasoning with LOTOS”, in: *Proc. Protocol Specification, Testing and Verification XIV* (ed. by VUONG S.), pp. 71–86, North-Holland, Amsterdam, Netherlands, October 1995.

[THO 97] THOMAS M. H., “Modelling User Views of Telecommunications Services for Feature Interaction Detection and Resolution”, in: *Proc. 4th. International Workshop on Feature Interactions in Telecommunication Networks and Software Systems* (ed. by DINI P., BOUTABA R., LOGRIPPO L. M. S.), pp. 168–182, IOS Press, Amsterdam, Netherlands, 1997.

[TRE 96] TRETMANS J., “Conformance Testing with Labelled Transition Systems: Implementation Relations and Test Generation”, *Computer Networks*, Vol. 29, pp. 25–59, 1996, Elsevier Science Publishers.

[TUR 93] TURNER K. J., “An Engineering Approach to Formal Methods”, in: *Proc. Protocol Specification, Testing and Verification XIII* (ed. by DANTHINE A. A. S., LEDUC G., WOLPER P.), pp. 357–380, North-Holland, Amsterdam, Netherlands, June 1993, Invited paper.

[TUR 94a] TURNER K. J., SINNOTT R. O., “DILL: Specifying Digital Logic in LOTOS”, in: *Proc. Formal Description Techniques VI* (ed. by TENNEY R. L., AMER P. D., UYAR M. Ü.), pp. 71–86, North-Holland, Amsterdam, Netherlands, 1994.

[TUR 94b] TURNER K. J., “Exploiting the *m4* Macro Language”, Technical Report CSM-126, Department of Computing Science and Mathematics, University of Stirling, UK, September 1994.

[TUR 97a] TURNER K. J., “An Architectural Foundation for Relating Features”, in: *Proc. 4th. International Workshop on Feature Interactions in Telecommunication Networks and Software Systems* (ed. by DINI P., BOUTABA R., LOGRIPPO L. M. S.), pp. 226–241, IOS Press, Amsterdam, Netherlands, 1997.

[TUR 97b] TURNER K. J., “Incremental Requirements Specification with LOTOS”, *Requirements Engineering Journal*, Vol. 2, pp. 132–151, November 1997, Springer-Verlag.

[TUR 97c] TURNER K. J., “Relating Services and Features in the Intelligent Network”, in: *Proc. 4th. International Conference on Telecommunications* (ed. by KUNŠTIĆ M.), pp. 235–243, University of Zagreb, Croatia, June 1997.

[TUR 98a] TURNER K. J., “An Architectural Description of Intelligent Network Features and Their Interactions”, *Computer Networks*, Vol. 30, no. 15, pp. 1389–1419, September 1998, Elsevier Science Publishers.

[TUR 98b] TURNER K. J., “Validating Architectural Feature Descriptions using LOTOS”, in: *Proc. 5th. Feature Interactions in Telecommunications and Software Systems* (ed. by KIMBLER K., BOUMA W.), pp. 247–261, IOS Press, Amsterdam, Netherlands, September 1998.

[TUR 00] TURNER K. J., “Formalising the CHISEL Feature Notation”, in: *Proc. 6th. Feature Interactions in Telecommunications and Software Systems* (ed. by CALDER M. H., MAGILL E. H.), pp. 241–256, IOS Press, Amsterdam, Netherlands, May 2000.

[VIS 91] VISSERS C. A., SCOLLO G., VAN SINDEREN M., “Architecture and Specification Style in Formal Descriptions of Distributed Systems”, *Theoretical Computer Science*, Vol. 89, pp. 179–206, 1991.

A. Language summaries

The ANISE service combinators are summarised in tables 3, 4 and 5. Tables 3 and 4 give generic combinators that are applicable to a range of services, and not just those in telecommunications. Table 5 gives telecommunications combinators that are particular to services such as those found in the IN. The modification operators of ANGEN are summarised in table 6. The test operators of ANTEST are summarised in table 7. LOTOS syntax is summarised in table 8.

Statement	**Meaning**
% *comment text*	informal explanation
alternates(*behaviour*)	alternates in each direction
behaviour	feature declaration/combination
collides(*behaviour*)	executes separately in each direction, reinforcing on collision
declare(*name*,*behaviour*)	gives name to behaviour
direction	local/remote subscribers, e.g. *00* (none) or *21*
disables(*behaviour1*,*behaviour2*)	execution of *behaviour1* stops *behaviour2*
duplicates(*behaviour*)	executes independently in each direction simultaneously
enables(*behaviour1*,*behaviour2*,...)	when each behaviour terminates, the next may start
enables_after_ack(*behaviour1*,*behaviour2*,...)	after response/confirm of each behaviour, the next may start
enables_after_try(*behaviour1*,*behaviour2*,...)	after request/indication of each behaviour, the next may start
enables_on_result(*behaviour1*,*result2*, *behaviour2*,*result3*,*behaviour3*,...,*behaviourN*)	*behaviour1* result decides future: if *result2* then *behaviour2*, ..., *behaviourN* by default
exit	successful termination
feature(direction)	named elementary behaviour, with optional direction
feature(*direction*,*pattern*,*property*,*group*(*param*))	elementary behaviour of given characteristics, for one group
feature(...,*group1*(*param*),*group2*(*param*))	elementary behaviour of given characteristics, for reques/acknowledgement groups

Table 3. *ANISE generic service combinators*

Statement	**Meaning**
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, e.g. *OffHook*
header(*commentary*)	preserves *commentary* as a header in the generated LOTOS
instances(*count*,*behaviour*)	number of independent behaviour instances
interleaves(*behaviour1*,*behaviour2*,...)	independent execution
interrupts(*behaviour1*,*behaviour2*)	*behaviour1* may interrupt then restart *behaviour2*
interrupts_after_ack(*behaviour1*,*behaviour2*)	*behaviour1* may interrupt then restart *behaviour2* after response/confirm from *behaviour2*
interrupts_after_try(*behaviour1*,*behaviour2*)	*behaviour1* may interrupt then restart *behaviour2* after request/indication from *behaviour2*
loops(*behaviour*)	repeats on successful termination
name	identifier (letters, digits, underscores), e.g. *Call_Clear*
overtakes(*behaviour1*,*behaviour2*)	*behaviour1* may begin later than *behaviour2* but finish earlier
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*,*line*,...)	subscriber number, line identifier and parameters
property	**consecutive**, **ordered**, **reliable**, **single**, **unreliable**
reverses(*behaviour*)	executes in opposite direction

Table 4. *ANISE generic service combinators*

Statement	**Meaning**
calls_three(*behaviour1*,*behaviour2*)	three-way calls follow *behaviour1*, normal calls *behaviour2* (TWC)
checks_busy(*behaviour*)	monitors and respects line busy state
dials(*behaviour*)	dialled number determines called number
dials_code(*behaviour*)	abbreviated dialling code is expanded for behaviour (ABD)
dials_one(*behaviour*)	called universal number is translated to called identifier (ONE)
diverts_always(*behaviour*)	called line is changed unconditionally (CFU)
diverts_ busy(*behaviour*)	called line is changed if called is busy (CFBL)
diverts_unanswered(*behaviour*)	called line is changed if call is unanswered (CFDA)
returns_ automatic(*behaviour1*,*behaviour2*)	ring-back calls follow *behaviour1*, normal calls *behaviour2* (ACB)
returns_ manual(*behaviour1*,*behaviour2*)	ring-back calls follow *behaviour1*, normal calls *behaviour2* (CCBS)
rings(*behaviour*)	called gets appropriate ring pattern
rings_display(*behaviour*)	called get calling number (CND)
rings_preference(*behaviour*)	called gets ring preference (DR)
screens_ in(*behaviour*)	behaviour for banned calling number will not terminate (TCS)
screens_out(*behaviour*)	behaviour for banned called number will not terminate (OCS)
waits(*behaviour1*,*behaviour2*)	waiting calls follow *behaviour1*, normal calls *behaviour2* (CW)
waits_call(*behaviour*)	handle call waiting globally (CW)

Table 5. *ANISE telecommunications service combinators*

Statement	**Meaning**
% *comment text*	informal explanation
append(*old,new,context*)	*old* has *new* appended, in the optional *context*
change(*old,new,context*)	*old* is changed to *new*, in the optional *context*
delete(*old,context*)	deletes *old*, in the optional *context*
fill(*old,new,context*)	*old* has *new* inserted, in the optional *context*
header(*commentary*)	preserves *commentary* as header in the generated ANISE
prefix(*old,new,context*)	*old* has *new* placed in front, in the optional *context*
wrap(*old,new,context*)	*old* has *new* wrapped around it, in the optional *context*

Table 6. *ANGEN modification operators*

Statement	**Meaning**
% *comment text*	informal explanation
decides(*behaviour1,behaviour2,...*)	tester makes behaviour choice
depends(*condition1,behaviour1, condition2,behaviour2,...,behaviourN*)	if *condition1* holds then *behaviour1*, ..., else *behaviourN* by default
exits(*behaviour1,behaviour2,...*)	*behaviour1*, ... in sequence, and then termination
fails(*behaviour1,behaviour2,...*)	*behaviour1*, ... in sequence, and then deadlock
interleaves(*behaviour1,behaviour2,...*)	independent parallel behaviours
offers(*behaviour1,behaviour2,...*)	network makes behaviour choice
recv(*number,primitive,parameter*)	subscriber at *number* receives *primitive*, with optional *parameter*
refuses(*behaviour1,behaviour2,..., behaviourN*)	*behaviour1*, ... in sequence, and then *behaviourN* must not happen
send(*number,primitive,parameter*)	subscriber at *number* sends *primitive*, with optional *parameter*
sequences(*behaviour1,behaviour2,...*)	*behaviour1*, ... in sequence
succeeds(*behaviour1,behaviour2,...*)	*behaviour1*, ... in sequence, and then terminates successfully
test(*name,behaviour*)	test name and behaviour

Table 7. *ANTEST test operators*

Statement	**Meaning**
(* *comment text* *)	informal explanation
exit	behaviour terminates successfully
exit(*results*)	successful termination with result values
exit(*results*) ≫ **accept** *declarations* **in** *behaviour*	termination with export of result values
gate	a ‘port’ at which event offers may synchronise
gate !*value*	event offer synchronises on given value
gate ?*variable*:*sort*	event offer synchronises on any value of sort, binding actual value to variable
gate !... ?... [*predicate*]	event offer with predicate on values
process *name* [*gates*] (*parameters*) : **noexit** := *behaviour*	named process definition with given gates and parameters, but no termination
process *name* [*gates*] (*parameters*) : **exit**(*results*) := *behaviour*	process that terminates successfully with given result sorts
process [*gates*] (*parameters*)	instantiation of named process
stop	behaviour does nothing (no further action)
event ; *behaviour*	prefixes event offer to some behaviour (‘followed by’)
[*guard*] -> *behaviour*	offers behaviour only if guard condition is satisfied (‘if’)
behaviour1 [] *behaviour2*	offers choice between two behaviours (‘or’)
behaviour1 ≫ *behaviour2*	if *behaviour1* exits, *behaviour2* follows (‘enables’)
behaviour1 [> *behaviour2*	*behaviour2* may disrupt *behaviour1*, unless *behaviour1* exits first (‘disabled by’)
behaviour1 || *behaviour2*	two behaviours in parallel, but fully synchronised on their events (‘synchronised with’)
behaviour1 ||| *behaviour2*	two behaviours in parallel, but independently occurring events (‘interleaved with’)
behaviour1 |[*gates*]| *behaviour2*	two behaviours in parallel, synchronising at given gates (‘synchronised on *gates* with’)

Table 8. *LOTOS syntax summary*

Titre en français

Descriptions architecturales de services avec LOTOS