Theory and Practice of Enhancing a Legacy Software System

Muffy Calder[(1)] and Evan Magill[(2)], Stephan Reiff-Marganiec[(1)], Vijeyananthan Thayananthan[(2)]

[(1)]University of Glasgow,
Department of Computing Science,
8-17 Lilybank Gardens, Glasgow G12 8RZ, Scotland, UK
sreiff@dcs.gla.ac.uk, muffy@dcs.gla.ac.uk

[(2)]University of Strathclyde,
Department of Electronic and Electrical Engineering,
204 George Street, Glasgow G1 1XW, Scotland, UK
e.magill@eee.strath.ac.uk, vijey@comms.eee.strath.ac.uk

1 Introduction

Telephone switching systems are an interesting class of legacy systems as they are complex, distributed and safety critical. As with most legacy systems, a simple replacement is impossible (or at least impractical), therefore changes in business practice and technology have induced a perpetual evolution of telephone switching systems.

One form of that evolution is the addition of *features*, additional functionality that provides new behaviour. But, new features may not interwork correctly, or consistently, with existing ones. This problem is known as *feature interaction*; it is particularly challenging in the context of legacy systems where we may not have access to the internal specification of the system. To obtain an intuition of the problem, consider two features *Call Waiting* and *Call Forwarding on Busy*, where the first signals the subscriber by a special tone during a call that another person attempts to call, the latter forwards all calls if the subscriber is busy. Both features work correctly, however if they are both subscribed by the same user it remains unclear which should handle an incoming call to a busy subscriber.

This paper describes work in progress on a joint project[1] whose aim is to develop an on-line feature manager that will detect and resolve feature interactions when new features are added to a legacy switching system. The architecture of the manager is based on the transactional approach of Marples and Magill [MM98] in which interactions can be detected. The key goals of the current project are to develop a theory of how interactions can be resolved, in a legacy context, and to implement and evaluate a resolution mechanism for a feature

[1] Hybrid Methods for Feature Interaction Detection and Resolution Techniques in collaboration with Mitel Telecom Ltd and Citel Technologies Ltd.

manager within a legacy switching system. The project therefore brings together two threads: formal modelling and switching systems engineering. In particular, we aim to integrate the results of an off-line analysis into an on-line system.

The transactional approach has already been described in [CMM99] and in [CR00]. In this chapter we concentrate on new results in formal modelling and theories of resolution, details of the chosen legacy system, and initial attempts to bring the two together.

In the next section we review some telecommunication concepts and terminology. Section 3 contains an overview of DESK switching system (the legacy software system) and in section 4 we give an overview of the off-line, formal modelling process and initial results. In section 5 we revisit DESK and present a set of common features that will be used both in the model and the DESK system. In section 6 we give some results from modelling the common features, and discuss how the model and the resolution can be refined in the light of that experience. Similarly, in section 7 we discuss extensions/refinements to the DESK system. In the final two sections we draw our conclusions and directions for further work.

2 Background

In modern telecommunications systems, control of the progress of calls and connections is provided by software at an exchange (or switch), this is referred to as a *stored program control* exchange. This software must respond (or react) to events such as lifting a handset or entering digits, as well as sending control signals to handsets and lines such as ringing tone or line engaged. Usually this software is referred to as Basic Call (Control) Software due to its fundamental rôle in allowing POTS (Plain Old Telephone System) basic calls. The interested reader can find further technical details in [RV94], [Bla98] and similar texts.

To illustrate a POTS basic call, consider an example of call set-up and clear-down for a call from user A to user B (from the perspective of A's basic call software):

Example 1
The basic call software receives an offhook from A, to which it responds with placing a dial_tone on A's line. Next, a sequence of digits is received, translated into a phonenumber (in this case B's) and triggers the sending of an alert to B and a ringing_tone to A. The fact that User B is free is signalled by receiving free. Assume user B goes offhook which leads to a connect being received, at which point a call is set up between A and B (meaning that the control has established a voicepath connecting A and B).
The basic call software receives an onhook from A indicating that the connection is to be cleared down, triggering a disconnect_request being sent to B. At this point the basic call is finished for A and a new call can be made.

Obviously POTS provides only basic functionality and many exchanges already deploy additional behaviour (i.e. features). For example, Private Branch

Exchanges (PABX) are a class of switches providing basic call behaviour and a large number of tightly integrated features. However, one may wish to add more features, in an open fashion, or integrate the PABX with another system which has further functionality (as illustrated in Fig. 1). This is the kind of scenario we attempt to address.

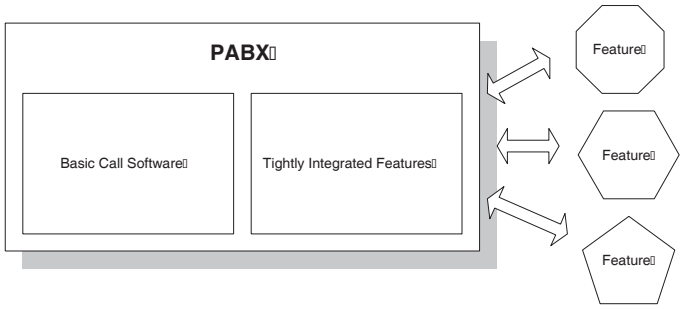

Fig. 1. PABX: Tight Integration and Open Addition of Features

3 Desk

DESK is a testbed for experimenting with feature interaction techniques, designed and developed at Strathclyde University [MTMS95]. In its current form it is a research orientated platform, however it is intended that it can be applied to real switching platforms. Indeed one goal of the current project is to port it to a MITEL switch. DESK was developed with a novel architecture that incorporates aspects of distributed database approaches such as rollback and a component for feature management (the *feature manager* (FM)). Initially, the effort focused on feature interaction detection, but while the detection aspect provided encouraging results, the resolution aspects detailed in [MM98] clearly require more effort.

In this chapter we describe a hybrid approach where we employ an off-line approach developed at Glasgow University to augment the resolutions abilities of DESK [CMM99]. To get a fuller appreciation of these changes, and in particular how legacy software is managed, it is useful to review the structure of DESK and its transactional capabilities. We note that this approach has a broad application in distributed systems (with or without legacy components), though DESK is in the telephony domain.

3.1 Desk Architecture

The DESK architecture explained in [MTMS95] and outlined in Fig. 2 consists of a software sub-system (SSS), and a hardware sub-system (HSS). These are two

distinct sections; it had been originally expected that the HSS would be replaced with real hardware. Hence the two sections communicate via a TCP/IP link with the HSS operating on a PC, and the SSS on a Unix machine. The HSS has not, to date, been replaced with physical terminals.

In essence the SSS represents the call control software of a telephone switching system, and so in addition to ordinary call control includes telephony features such as call waiting and three-way call. It also contains the Hardware Abstraction Models (HAMs) which represent hardware facilities in the HSS, and software for controlling, monitoring and managing signals between the HSS and SSS. A HAM is a state machine capturing the behaviour of a physical line or a trunk.

The HSS represents, in software, the physical components of the real switch; hardware such as trunks and telephony terminals. The user of DESK interacts with these components through the HSS Graphical User Interface (GUI). The user can initiate calls, answer phones, and generally perform normal telephony actions. Hence the HSS can pass the user initiated signals, such as *onhook* and *offhook*, to the SSS through the TCP/IP link. Of course the HSS must also generate responses from the switching system such as *ringing*, *dialtone*, and *ringtone*. These responses originate from the SSS in response to the user-initiated signals, and again traverse the communications link. Note that the current state of a terminal is actually stored in the HAM situated in the SSS, the HSS provides the hardware emulation.

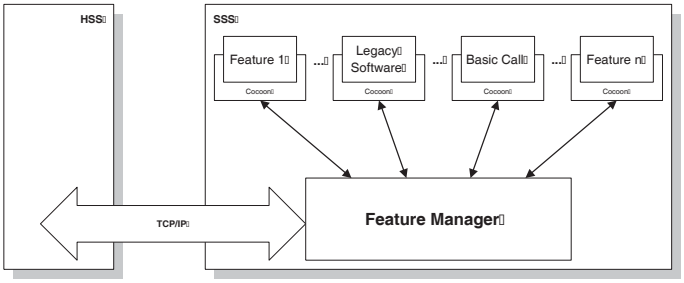

Fig. 2. Software Architecture of DESK

From Fig. 2 it is clear that the SSS has two main components, a Feature Manager and the call control. The feature manager is a central component as almost every action involved in DESK passes through it. This provides a point of control over each telephone call. In practice this represents a common approach that is used within many communications networks.

In its simplest form, a feature is additional functionality, such as call waiting, that adds to the basic operation of a plain telephone call. Features are usually added incrementally and implemented separately, each as a separate process. However in DESK we also consider the basic call control to be a feature, albeit an essential one. It therefore operates in parallel with any other feature. Moreover

existing call control software, with or without its own embedded features, is also considered as simply another feature.

How is it possible to treat such legacy software? As shown in Fig. 2, each feature is embedded in a cocoon. The cocoon simply runs one or more process instances. It does not care which type of feature it surrounds; the feature manager controls the number of instances. In essence the feature manager passes two types of signal to the cocoon, those to control the cocoon itself and those destined for a particular feature process instance. The feature instance reply will be dictated by the feature behaviour; be the feature a legacy system, the basic call control, or an incremental behaviour. The ability to support more than 1 instance of a feature in a single call is an essential part of DESK's novel approach to feature interaction.

On-Line Transactional Approach. The on-line technique employed in DESK is an efficient way to detect feature interaction while the switch is operational. Contrary to many run-time approaches, DESK does not require the use of predefined tables (or rules) to detect feature interaction. In effect "a priori knowledge" is not required. This makes it a scalable solution. Moreover the technique was developed mindful of the need to embed the approach in real systems.

An essential characteristic of this technique is the ability to roll back a feature. Here we have only space to outline the approach, however details can be found elsewhere [MM98]. In brief, rollback allows the FM to explore the behaviour of the features, and to roll back from scenarios it decides are undesirable, and to commit to the best. In practice the FM can only explore within stages of a call, such as while the phone is ringing as it cannot go beyond the point of the next user initiated event such as an *offhook*. Clearly it is also constrained by real-time issues.

This approach is achieved by the FM and the feature cocoon; the feature plays no part in the roll back behaviour. Hence the ability to handle legacy software. On receipt of a user-initiated event such as *digits received*, the FM passes the signal to *all* features. It then collects the responses. Any response back to the FM is *not* passed back to the HSS but is in turn sent to all the features. This continues until there are no responses. If at any stage more than one feature replies, that is, there is more than 1 response, then all of the received responses are bounced back to the features yet again. However they are not simply returned in any arbitrary order. Every possible combination is tried. If two responses were received, then firstly one response on its own is tried. Later the other is tried. Then one response followed by the other, and finally in the reverse order. This requires the FM to build up a tree of observable behaviour, it also means that the FM must instruct the cocoon to run a number of instances of the feature, one for each branch in the tree. In the example of two responses, there will be a branching factor of 4 at this point in the tree. In the case where there is only one reply there will be a linear series of states.

Branching at any point in fact highlights feature interaction and is an essential part of how DESK *detects* feature interaction. Resolution is the act of

choosing the best path through the tree. Once the best path is chosen, or indeed there is no branching and hence no interactions, the FM instructs the cocoon to commit to the chosen series of transitions, and the correct series of responses is passed to the HSS. The FM now awaits the next stimulus.

This is difficult to visualise without the aid of an example. Below we present an example of the dialogue between the FM and the feature cocoons.

Example of Feature Manager – Feature Dialogue. This is best illustrated with the aid of a Message Sequence Chart (MSC). The MSC shown in Fig. 3 above describes the dialogue between the FM and the basic call feature (BCF) and the call forwarding unconditional feature (CFU). Strictly messages such *START-TRANS* and *RESP-COMP* are "house keeping" signals linked to the cocoon rather than a feature. For example, the message *RESP-COMP* means that a feature has completed its response to the previous stimulus. Messages to and from the HSS are shown partly in lower case, and are not bounced back by FM when received from a feature.

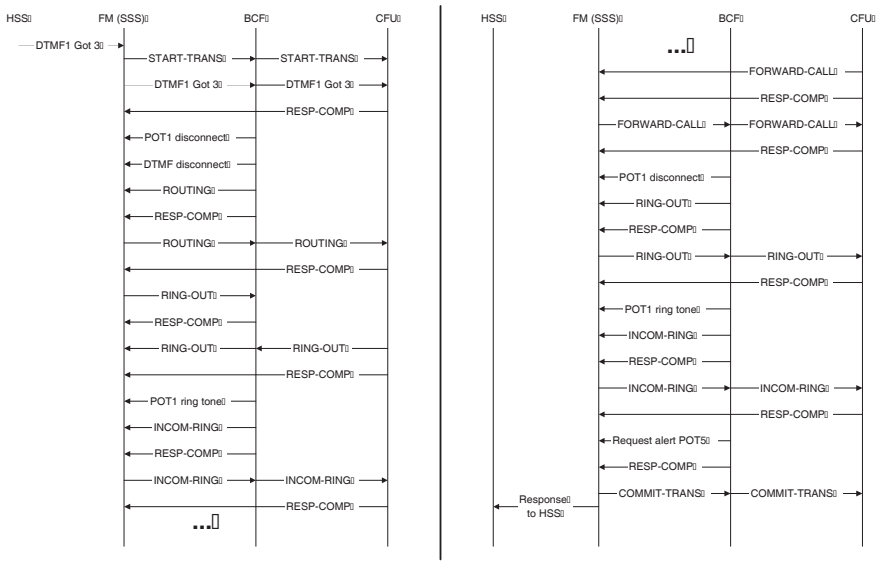

Fig. 3. MSC Diagram for *offhook* Operation in DESK

The example does not show the complete call, but the stage following the originator dialling the digit 3, to represent the terminator. In an earlier, unshown, phase of the call, a Dial Tone Multiple Frequency (DTMF) receiver is connected to the phone when the originating telephone is given dial tone. The receiver is a hardware component that translates the tones of the phone to a digit. When

the feature manager receives the *DTMF 1 got 3* event from the receiver in the HSS, in addition to some housekeeping, it passes *DTMF 1 got 3* event to all the features present, that is the BCF and CFU. Only the BCF replies; it disconnects the DTMF receiver and POT 1's dial tone, and starts routing of the call. Once the BCF has completed all its actions successfully, the FM receives the housekeeping message *RESP-COMP* from the BCF cocoon. Only the routing message is returned to the BC and CFU features, as the other two are HSS messages.

In response, the BCF sends the *RING-OUT* message to the FM. (Again, when this action is completed, the FM receives the message *RESP-COMP* from the BCF.) The *RING-OUT* message is returned by the FM to the BC and CFU features.

The BCF replies. The HSS message, *POT 1 ring tone*, will not as be echoed back. However the *INCOMING-RING* message is returned to the BC and C-FU features. The FM receives the messages *FORWARD-CALL* from the CFU which it echoes back to both features. Here the BCF replies with *POT 1 disconnected* and *RING-OUT* from the BC. The HSS message *POT 1 disconnected* disconnects the originator from ringing tone.

This exchange continues until the originator is re-connected to ringing tone and the new destination (POT 5) is ringing. The FM has no events to bounce back. No further events are possible until the new terminator (POT 5) answers or the originator hangs up, that is, a new HSS event is received. As there is only this one behaviour, (there is no branching factor greater than 1) there is no feature interaction and no need to rollback to try alternatives. In other words, only one feature answers to any given FM stimulus. Hence the FM commits the features using the *COMMIT-TRANS* message and relays the stored series of HSS messages to the HSS. The features and the committed features are now synchronised awaiting the next user event.

3.2 Experiment and Results

Details of the results are available elsewhere [MM98], here we simply outline the results to provide a backdrop to the changes being carried out in this project.

There are eight features selected for the experiments; they are Call Forwarding Unconditional (CFU), Call Forwarding Busy (CFB), Terminating Call Screen (TCS), Do Not Disturb (DND), Hot Line (HL), Originating Call Screen (OCS), Teen Line (TL) and Call Waiting (CW). The choice was, in part, influenced by a well known international competition on feature interaction [GBGO98].

Each scenario was run in turn, and by studying the behaviour trees generated by the FM it was possible to monitor the detection of feature interaction. The initial studies focused on *detection* rather than *resolution*. The results are summarised in Table 1. Only the boxes ticked by "X" show the pair-wise feature interactions. Grey boxes always provide same feature combination as in the upper triangle boxes. So, they can be avoided. Here, blank boxes are assumed that they have no feature interaction when particular type of feature interac-

tion detection technique is used. Here A and B are originator and terminator respectively.

Features	CFU (B)	DND (B)	OCS (A)	TL (A)	CFB (B)	HL (A)	TCS (B)	CW (B)
CFU (B)	X	X			X		X	X
DND (B)					X		X	
OCS (A)								
TL (A)						X		
CFB (B)					X			X
HL (A)								
TCS (B)								
CW (B)								

Table 1. Summarised Results of Feature Interaction from the DESK Features (This table is reproduced from [MM98])

These are clearly promising results for detection. However, initial studies [MM98] showed a limited ability of DESK to resolve interaction, certainly in a coherent and consistent manner. This project augments DESK with an off-line approach to permit a better ability to resolve interactions.

4 Modelling Features and the Feature Manager

As discussed above, we aim to augment DESK based on experience gained through formal modelling. The goal of the formal modelling process is therefore to answer questions such as:

- What does it mean for two or more features to *interfere*?
- What are the possible resolutions?
- What are bad/good resolutions?
- Can we answer those questions from the observable behaviour only?

In this section we outline salient components of our model and the results obtained.

4.1 Models of Features based on Observable Behaviour

The messages sent within the switching system are crucial, as they are the only observable component (we have no access to internal states).

A message consists of an event (a type and possibly a value) and an associated I/O aspect. Examples of event types are *dial* or *alert*; the value depends on

the type (e.g. *dial* has the dialled number, *dial-tone* does not require a value, an associated event value to *announce* indicates the message). The I/O aspect indicates whether the message is input to or output from a feature. We adopt the convention that input messages are preceded by $^+$, output messages are preceded by $^-$. Further, two messages a and b are said to be name equivalent when their event types are equal, written $a \equiv b$. If two messages only differ in their I/O aspect they are called inverse and if they do not differ we call them equal.

When we are only concerned about the I/O aspect of a message m, we write it as ^+m or ^-m. '= '-" denotes the null value and, by abuse of notation, τ is also a special null message (or silent transaction, indicating that a feature makes an internal state-change without sending an observable message).

Example messages are $^+(dial, n)$, $^-(announce,$ "screened") and $^-(i\_alert, -)$. Features are defined in terms of messages, as follows:

Definition 1 (Alphabets, Traces, and Features). *An alphabet is a set of messages partitioned into 2 non-empty sets: input messages and output messages. A trace over an alphabet is a non-empty finite sequence of elements of the alphabet starting with an input message. A feature is a set of traces over an alphabet such that each element of the alphabet occurs in at least one trace.*

To illustrate the previous definitions, the following example contains an alphabet, and two features.

Example 2

Consider the alphabet:

$$\alpha = \{^+(dial, n),\ ^-(billing\_reverse, -),\ ^+(onhook, -),\ ^+(offhook, -),$$
$$^-(announce, \text{"enter PIN?"}),^-(announce, \text{"wrong PIN"}),\ \tau\}.$$

A Reverse Charging Feature (RC) is defined by the trace set:

$$F_{TCS} = \{^+(dial, n)^-(billing\_reverse),$$
$$^+(dial, n)^-\tau\}.$$

A Teenline (TL) Feature is defined by the trace set:

$$F_{TCS} = \{^+(offhook, -)^-\tau,$$
$$^+(offhook, -)^-(announce, \text{"enter PIN?"})^+(onhook, -),$$
$$^+(offhook, -)^-(announce, \text{"enter PIN?"})^+(dial, n)^-\tau),$$
$$^+(offhook, -)^-(announce, \text{"enter PIN?"})^+(dial, n)$$
$$^-(announce, \text{"wrong PIN"})^+(onhook, -)\}.$$

Feature interactions are generally understood to mean "points of contact" between two features; they may be desired or undesired interactions. Since it is generally difficult to decide whether or not an interaction is desired, particularly within the context of a legacy system, we use the term *interference* to mean that two or more features interfere with each other's behaviour. The intuitive understanding is that features can only interfere if they have common input messages in their respective alphabets, or one alphabet contains an input message that is matched by an output message in another alphabet. Common output messages cannot lead to interference, simply because they cannot occur simultaneously, in

order to do so, they need to be triggered by the same event and hence a common input message would exist.

Definition 2 (Feature Interference). *n features $F_1 ... F_n$ with respective alphabets $\alpha_1 ... \alpha_n$ interfere iff there are 2 distinct features F_i and F_j such that*

$$(\exists^{+} a \in \alpha_i . (\exists b \in \alpha_j . a \equiv b))$$

Example 2 introduced two features, where the first trace of the RC feature contains $^{+}(dial, n)$ as do the last two traces of TL. Thus, these two features interfere

4.2 Resolving Feature Interference

The feature manager's purpose is to detect and resolve interference. That means it must *detect* every possible way in which features can interfere, generate every possible resolution, and then select an optimal set of resolutions. We assume here that detection has already occurred, and concentrate on resolution. Note that the final set may not be a singleton because either we cannot distinguish amongst resolutions, or because the best resolution depends on a later event which will be provided by a user.

What is a resolution? When n features interfere, then we propose that the interference can (*potentially)* be resolved in a number of ways:

- exactly one of the features is allowed to respond (i.e. all others are disabled),
- two of the features can respond, in either order,
- three of the features can respond, in any order,
- ...
- n features can respond, in any order.

The diagram in Fig. 4 shows a simple example of a resolution space (i.e. all possible resolutions) involving 3 features.

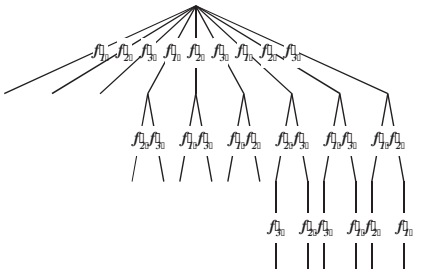

Fig. 4. A Simple Resolution Space

The above description assumes a *coarse* granularity, that is, features are considered as atomic. However, since we can observe messages (indeed, we cannot distinguish the "beginning" or "end" of a feature), our approach to resolution will be more *fine* grained. That is, we will adopt the approach described above, but we will interleave the messages of features, as well as the features themselves. In all case, our goal is to allow as many features to proceed as possible.

Overlapping Interleavings. Each possible resolution is obtained by a fine grained interleaving of messages *after* an interaction is detected. This is defined in more detail as follows.

When an interaction has been detected, we know that one trace starts with an input message that is either matched by an input message in the other trace or by an equal output message in the other trace. In the following, call the trace starting with the matched input a and the other trace b. Trace b might contain a subsequence before the match occurs, and the match may also be a subsequence. Figure 5 illustrates this situation: $a_1...a_j$ and $b_i...b_{i+j+1}$ are the matching subsequences, $b_1...b_i - 1$ is the subsequence before the match. Every resolution therefore begins with $b_1...b_{i-1}$, followed by a suitable arrangement of the messages in $a_1...a_j$ and $b_i...b_{i+j+1}$ (to account for input/output), followed by one of the possible interleavings of a and b (i.e. $a_{j+1}...a_n$ and $b_{i+j+2}...b_m$). A more complete definition is given in [Rei00].

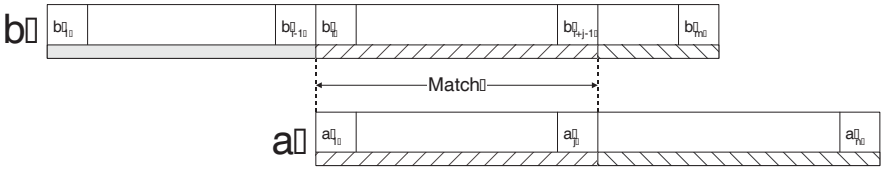

Fig. 5. Overlapping Interleavings

Combinatorially, such a brute force approach could be a disaster. For example, as seen from above, interleaving 3 features at the coarse level already reveals 15 resolutions; it remains to the reader to imagine the fine grained possibilities!

However, this complexity is managed in two ways. First, interference between more than 2 features is relatively rare ([KMMR00]). Moreover, we can rule out large numbers of potential resolutions, on the fly, if we have a theory of *good* and *bad* resolutions.

Selecting Resolution Set. After constructing the resolution space the remaining task is to choose those traces that represent good resolutions, or preferably the best resolution. Our approach is based on the elimination of unacceptable

(i.e. *bad*) resolutions and then an ordering of the remaining acceptable (i.e. *good*) ones.

Elimination is either a) *message independent*, or b) *message dependent*. While the former is the most desirable, so far we have not identified many powerful rules, they remain rather simple, such as removal of duplicates or subsequences. The latter is more promising, however, depending on assumptions that can be made about the semantics of messages. For example, consider a simple message of the form *forward(x, y)*. A resolution containing both *forward(x, y)* and *forward(x, z)* or *forward(x, y)* and *forward(u, v)* could be considered unacceptable, assuming the usual semantics for *forward*. On the other hand, a trace containing both *forward(x, y)* and *forward(y, z)* would be acceptable.

Further rules are based on classes of messages, one such class could be *treatments* (such as tones and announcements). A very basic example is that *treatment*-messages should only occur between and *offhook* and *onhook* messages.

The issue of identifying useful general rules is very interesting and we will revisit this in section 6. Recall that the aim of the project is a hybrid approach, advancing an operational legacy system. So far we have developed a model and within that model, a characterisation of all possible resolutions and the elimination of some of those which are unacceptable. We have not considered the relation and adaptability to of the same to the legacy system. As an initial step in harmonising the two aspects of the work, the next section introduces a common set of features.

5 Harmonising Feature Behaviour

To harmonise the model and DESK it was necessary to capture formally the behaviour of all the features in DESK, both the incremental features and the basic call. Nine finite state machines (FSMs) were extracted from a manual inspection of the code. As an exemplar, two FSMs are described below.

5.1 FSM for features based on Desk - CFU and CW

We use the labels A, B and C to represent originator, terminator and 3^{rd} party respectively. A circle represents a state, and the name of the state is shown within the circle. The directed edges are labelled in the form *input message / output message*. The edges represent the transition from one state to another on the receipt of an *input message*. It should be stressed that the names of the states are purely for ease of explanation and carry no other significance. Indeed the Feature Manager can only see the external behaviour of the features and is unaware of the states directly.

Figure 6 represents the FSM for the *call forwarding unconditional* (CFU) feature. CFU responds to two input messages, *MSG_INCOMING_RING (A, B)* and *MSG_TERM_BUSY (A, B)*. The response in both cases is *MSG_FORWARDCALL (B, C)*. In other words, always forward the call. Incremental features such as CFU in DESK are separate from the Basic Call feature, and so there is no internal

knowledge of the state of the terminal. Hence the FSM simply requires one state. The two incoming messages are generated by the basic call feature, and are "bounced back" from the FM. It is through the messages that the state of the terminal can be determined. Clearly this is more important in call forwarding busy, and indeed in the next feature call waiting.

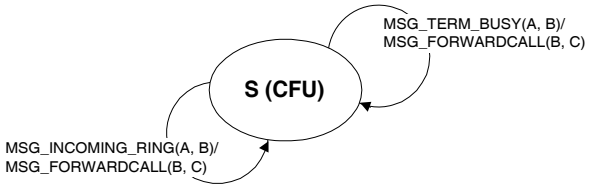

Fig. 6. FSM for CFU

Figure 7 represents the FSM for the *call waiting* (CW) feature. The three states alert, swap and idle represent (respectively) conditions where the CW tone is alerting the subscriber, where the subscriber can swap terminals, and where the feature is not active. In the alert state, the trigger message *MSG_TERM_BUSY (A, B)* shows that the terminator is busy and the FSM responds accordingly and moves to the alert state. Here a flash hook allows the subscriber to switch to the incoming call and in the new state swap the subscriber can use the switch hook to swap terminals indefinitely until the call finishes.

Again the FSM only knows the terminal states through the messages.

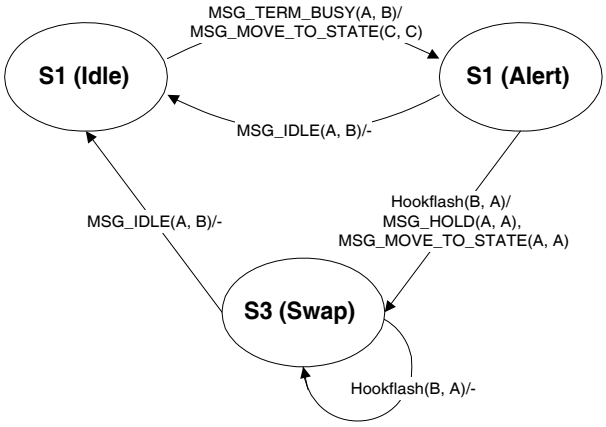

Fig. 7. FSM for CW

6 Improving the Model

Section 4 outlined the formal modelling, based on a hypothetical set of features and message formats. Now we evaluate this work in the context of the features described in section 5.

We note here that the entire process as an iterative one: modelling results (as in section 4) improve the feature manager's capability, allowing for better experimental results that in turn are fed back to improve the underlying theory. This iterative improvement process is shown in Fig. 8.

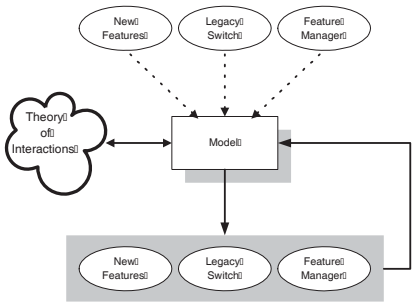

Fig. 8. Iterative Development Process

As pointed out earlier, the current rules for eliminating resolutions are rather simplistic. One could argue that the current rules should be obvious to anyone working in the field and could be derived from the legacy systems detection output. However, the systems current output only explores resolutions until a new user input is expected, whereas we explore longer traces possibly containing several inputs. The advantage of this exploration is that at the time we make a decision, its longer term effect is known.

Consider the two features CW and CFU as described previously. The message format used by the DESK testbed is similar to the format used in the model. It remains to adapt any elimination rules to specific DESK messages. For example we have the message dependent rule: any messages occurring after *msg_idle* provide inconsistent behaviour to the user. We note that obviously all message independent rules are still applicable. In order to produce more advanced general rules an *accepted semantics of messages* is needed, and before we can develop further rules we need to gain more experience about the semantics of the messages provided by the testbed.

Finally we return to the idea of ordering acceptable resolutions, as mentioned in 4. Such an ordering would be based on a notion of *satisfaction* of a set of features; a feature is said to be satisfied if it can proceed. This means, for example, that a resolution that satisfies say features f_1 and f_2 will be more acceptable

than a resolution that satisfies either only f_1 or f_2. We may consider this to be message independent.

Example 3 shows output generated by a Python[2] implementation of the model, where traces 17-19 and 20-21 represent the individual features Call Waiting and Call Forwarding Unconditional respectively. Applying the above-mentioned rules, we can remove traces 0-4 and trace 10, thus reducing the solution space from 22 traces to 16.

Example 3

```
       +------------ output without destination and origin (for brevity)
       |       +---- the feature manager as defined
       |       |
printshort(fm(desk_cw, desk_cfu))
    0 :  (+, msg_term_busy) (-, msg_move_to_state) (+, msg_idle) (-, msg_send_to_resource)
         (-, msg_forward_call)
    1 :  (+, msg_term_busy) (-, msg_move_to_state) (+, msg_idle) (-, msg_forward_call)
         (-, msg_send_to_resource)
    2 :  (+, msg_term_busy) (-, msg_move_to_state) (-, msg_forward_call) (+, msg_idle)
         (-, msg_send_to_resource)
    3 :  (+, msg_term_busy) (-, msg_forward_call) (-, msg_move_to_state) (+, msg_idle)
         (-, msg_send_to_resource)
    4 :  (+, msg_term_busy) (-, msg_move_to_state) (+, hookflash) (-, msg_hold)
         (-, msg_move_to_state) (+, msg_idle) (-, msg_forward_call)
    5 :  (+, msg_term_busy) (-, msg_move_to_state) (+, hookflash) (-, msg_hold)
         (-, msg_move_to_state) (-, msg_forward_call) (+, msg_idle)
    6 :  (+, msg_term_busy) (-, msg_move_to_state) (+, hookflash) (-, msg_hold)
         (-, msg_forward_call) (-, msg_move_to_state) (+, msg_idle)
    7 :  (+, msg_term_busy) (-, msg_move_to_state) (+, hookflash) (-, msg_forward_call)
         (-, msg_hold) (-, msg_move_to_state) (+, msg_idle)
    8 :  (+, msg_term_busy) (-, msg_move_to_state) (-, msg_forward_call) (+, hookflash)
         (-, msg_hold) (-, msg_move_to_state) (+, msg_idle)
    9 :  (+, msg_term_busy) (-, msg_forward_call) (-, msg_move_to_state) (+, hookflash)
         (-, msg_hold) (-, msg_move_to_state) (+, msg_idle)
   10 :  (+, msg_term_busy) (-, msg_move_to_state) (+, hookflash) (-, msg_hold)
         (-, msg_move_to_state) (+, hookflash) (+, msg_idle) (-, msg_forward_call)
   11 :  (+, msg_term_busy) (-, msg_move_to_state) (+, hookflash) (-, msg_hold)
         (-, msg_move_to_state) (+, hookflash) (-, msg_forward_call) (+, msg_idle)
   12 :  (+, msg_term_busy) (-, msg_move_to_state) (+, hookflash) (-, msg_hold)
         (-, msg_move_to_state) (-, msg_forward_call) (+, hookflash) (+, msg_idle)
   13 :  (+, msg_term_busy) (-, msg_move_to_state) (+, hookflash) (-, msg_hold)
         (-, msg_forward_call) (-, msg_move_to_state) (+, hookflash) (+, msg_idle)
   14 :  (+, msg_term_busy) (-, msg_move_to_state) (+, hookflash) (-, msg_forward_call)
         (-, msg_hold) (-, msg_move_to_state) (+, hookflash) (+, msg_idle)
   15 :  (+, msg_term_busy) (-, msg_move_to_state) (-, msg_forward_call) (+, hookflash)
         (-, msg_hold) (-, msg_move_to_state) (+, hookflash) (+, msg_idle)
   16 :  (+, msg_term_busy) (-, msg_forward_call) (-, msg_move_to_state) (+, hookflash)
         (-, msg_hold) (-, msg_move_to_state) (+, hookflash) (+, msg_idle)
   17 :  (+, msg_term_busy) (-, msg_move_to_state) (+, msg_idle) (-, msg_send_to_resource)
   18 :  (+, msg_term_busy) (-, msg_move_to_state) (+, hookflash) (-, msg_hold)
         (-, msg_move_to_state) (+, msg_idle)
   19 :  (+, msg_term_busy) (-, msg_move_to_state) (+, hookflash) (-, msg_hold)
         (-, msg_move_to_state) (+, hookflash) (+, msg_idle)
   20 :  (+, msg_incoming_ring) (-, msg_forward_call)
   21 :  (+, msg_term_busy) (-, msg_forward_call)
```

In order to improve the feature manager in the legacy system, it is necessary to show precisely what results can be expected from the modelling. The modelling steps produce intermediate results in three categories: relations on messages, grammars describing behaviour and examples. We consider each in turn.

Relations on messages allow grouping individual messages into larger classes that all have common properties; the previously named class of *treatments* is an example, other classes include user events (e.g. *offhook*, *onhook*, *flash*) and data

[2] Python is an interpreted programming language that allows rapid prototyping `http://www.python.org`

requests. Messages in one class may be inconsistent, therefore a user should not receive a sequence of such messages.

Behaviour, acceptable or unacceptable, can also be described by grammars that are based on both individual messages and classes. For example consider the messages *onhook* and *offhook* as well as the classes *treatments* (T) and *non-treatments/non-offhook* (N). Then a grammar of unacceptable traces is *onhook* T^* N^+ *offhook*; clearly no resolution should contain a word from this language.

We may not be able to encapsulate all rules by regular expressions, or even context free grammars; concrete examples describing positive or negative behaviour can make a valuable contribution. We may also use inductive inference ([AS83], [Gol76]) to learn grammars from positive or negative examples.

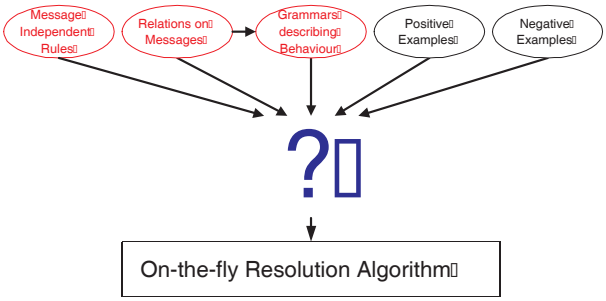

Fig. 9. Results from Modelling Process

Figure 9 illustrates the variety of results to be synthesised to produce our final goal: an on-the-fly algorithm for resolving feature interactions. It is important to observe that the complete solution space will never be constructed: the resolution process will be deeply intertwined with the detection algorithm, thus only branches leading to acceptable resolutions will be explored.

7 Modifications to the Feature Manager

Desk models a switching system with legacy software, therefore it is only appropriate to change the FM. Namely, the FM must use the information supplied by the off-line technique to improve its resolution. Initially this will take the form of a behaviour tree. More accurately, it will use fragments of a tree describing acceptable behaviour and fragments expressing illegal behaviour. This can be related to Fig. 9 showing positive and negative examples respectively.

The features used in both DESK and the model have been harmonised, so this allows behaviour trees fragments generated in the off-line phase to be passed to DESK. The fragments capture the trajectories through the solution space

representing either a resolution or a route to be pruned. Indeed the latter may be expressed through their absence. Earlier the ability of the off-line approach to look "further ahead" was described. This allows the off-line approach the ability to offer DESK a better resolution than it can provide locally.

However this should be seen as a preliminary approach. It is important for the modelling to have a more abstract representation. This implies the need for DESK to understand a more abstract set of resolution rules.

By harmonising DESK and the off-line model we form the basis for an effective resolution technique, but this tight coupling has a number of drawbacks. Not the least is keeping them harmonised, but more importantly is the desire to allow the off-line approach to generate generic rules that have a broader application. Moreover it is important that the generated resolutions are not specific to issues of DESK implementation.

Clearly a key issue here is the ability for the FM to understand more generic resolution rules such as, "no terminal can have more than one treatment after a single user event". A notation to express this has been discussed in the previous section and is shown as grammars describing behaviour in Fig. 9. Once this has been refined the FM will be adapted to replace the behaviour tree fragments with these behaviour descriptions. An algorithm to translate from this description to FM views of behaviour must be developed and embedded within the FM.

Here we describe a two step process, firstly using behaviour fragments, then second rules. It may be advantageous to concentrate on the latter. This is actively under review within the project.

8 Conclusion

The final result of this research will provide an on-line feature manager which detects and resolves interactions between new and legacy features that enhance legacy telecommunications systems. The feature manager will encapsulate a strong and efficient resolution mechanism.

In detail, the modelling area of this work provides a theory of features and interactions that forms the basis for the design of on-the-fly algorithms to resolve interactions. The feature manager based on this theory can handle any black-box system, provided the components can be encapsulated in transactional cocoons and an accepted semantics for the sent messages exists. The approach is applicable to legacy and third party systems – both of which do not guarantee the availability of design information for different reasons – as no design information is required.

The work on the DESK testbed shows that it is very hard to gain an insight into the functioning of a legacy system, even at the superficial level required for this project. However, it also shows that it is realistic to extend legacy systems with a feature manager and features in the proposed way.

9 Further Work

As the whole process is iterative, the knowledge gained through this work can be fed back and better solutions can be sought. In order to produce a prototype feature manager for the testbed further work in both areas, modelling and work on the DESK testbed, needs to be advanced individually but also requires continuous harmonisation to guarantee the relevance of the contributions towards the final aim.

Desk. Work will focus on altering the FM to improve its resolution. Behaviour fragments, and later rules, will be passed to the FM from the off-line approach. The FM will use this information to make more informed decision on permitted feature behaviour. Understanding and applying the rules will require an algorithm to be developed and embedded within the FM. This is expected to be the main focus of work within DESK over the coming year.

Modelling. A model underpinning a theory of features has been achieved, that in particular captures the construction of the resolution space and the fundamental ideas for resolution. Further work will concentrate on refining these resolution ideas, especially defining the mentioned classes of relations on events, grammars for describing behaviour and if required, examples of desired and unacceptable behaviour.

Acknowledgements

The authors thank Dave Marples (Telcordia Technologies) for stimulating discussions about both his approach and the DESK testbed.

References

[AS83] D. Angluin and C.H. Smith. A survey of inductive inference: theory and methods. *Comput. Surveys*, 15(3):237–269, 1983.

[Bla98] U. Black. *The Intelligent Network*. Prentice Hall, New Jersey, 1998.

[BV94] L. G. Bouma and H. Velthuijsen, editors. *Feature Interactions in Telecommunications Systems*. IOS Press (Amsterdam), May 1994.

[CGL+94] E. J. Cameron, N. Griffeth, Y.-J. Lin, M. E. Nilson, and W. K. Schnure. A feature interaction benchmark for IN and beyond. In *[BV94]*, pages 1–23, May 1994.

[CM00] M. Calder and E. Magill, editors. *Feature Interactions in Telecommunications and Software Systems VI*. IOS Press (Amsterdam), May 2000.

[CMM99] M. Calder, E. Magill, and D. Marples. A hybrid approach to software interworking problems: Managing interactions between legacy and evolving telecommunications software. *IEE Proceedings - Software*, 146(3):167–180, April 1999.

[CO95] K. E. Cheng and T. Ohta, editors. *Feature Interactions in Telecommunications Systems III*. IOS Press (Amsterdam), October 1995.

[CR00] M. Calder and S. Reiff. Modelling legacy telecommunications switching systems for interaction analysis. In Peter Henderson, editor, *Systems Engineering Business Process Change*, pages 182–195. Springer Verlag, London, May 2000.

[DBL97] P. Dini, R. Boutaba, and L. Logrippo, editors. *Feature Interactions in Telecommunication Networks IV*. IOS Press (Amsterdam), June 1997.

[GBGO98] N. Griffeth, R. Blumenthal, J.-C. Gregoire, and T. Ohta. Feature interaction detection contest. In *[KB98]*, pages 327–359, September 1998.

[Gol76] E. Gold. Language identification in the limit. *Inf. and Control*, 10:447–474, 1976.

[KB98] K. Kimbler and L. G. Bouma, editors. *Feature Interactions in Telecommunications and Software Systems V*. IOS Press (Amsterdam), September 1998.

[KK98] D. O. Keck and P. J. Kuehn. The feature and service interaction problem in telecommunications systems: A survey. *IEEE Transactions on Software Engineering*, 24(10):779–796, October 1998.

[KMMR00] M. Kolberg, E. H. Magill, D. Marples, and S. Reiff. Results of the second feature interaction contest. In *[CM00]*, pages 311–325, May 2000.

[MM98] D. Marples and E. H. Magill. The use of rollback to prevent incorrect operation of features in intelligent network based systems. In *[KB98]*, pages 115–134, September 1998.

[MTMS95] D. Marples, S. Tsang, E. H. Magill, and D. G. Smith. A platform for modelling feature interaction detection and resolution techniques. In *[CO95]*, pages 185–199, October 1995.

[Rei00] S. Reiff. Identifying resolution choices for an online feature manager. In *[CM00]*, pages 113–128, May 2000.

[RV94] F. J. Redmill and A. R. Valdar. *SPC: Digital Telephone Exchanges*. Peter Peregrinus Ltd. (Stevenage), 1994.

[TM97] S. Tsang and E. H. Magill. Behaviour based run-time feature interaction detection and resolution approaches for intelligent networks. In *[DBL97]*, pages 254–270, June 1997.

[TM98] S. Tsang and E. H. Magill. Learning to detect and avoid run-time feature interactions in intelligent networks. *IEEE Transactions on Software Engineering*, 24(10):818–830, October 1998.