Identifying Resolution Choices for an Online Feature Manager

Stephan Reiff*
Department of Computing Science
University of Glasgow, Glasgow, UK
email: sreiff@dcs.gla.ac.uk

Abstract. The feature manager developed by Marples and Magill provides an attempt to detect and resolve interactions online; it is particularly suitable for legacy software. In order to improve the relatively weak resolution mechanisms a deep understanding of the approach and the potential solution space that it entails is required. Consequently, we have developed a formal description of the approach; this allows us to define the solution space and to suggest resolution mechanisms. Throughout, we consider only *observable* behaviour of features and legacy software.

1 Introduction and Motivation

There are a multitude of approaches to solving the feature interaction problem in telecommunications systems, but very few address the special requirements imposed by legacy and proprietary systems. Legacy systems usually involve large, fragile collections of code that are often sparsely or unreliably documented; moreover, further evolution of such a system often leads to breakage. Similar problems arise from proprietary systems, where access to documentation or code is often restricted for economic reasons.

Nevertheless, the development and advancement of the underlying technology, as well as the users' rising requirements, requires extensions to legacy systems and the integration of new components within legacy and proprietary systems. Clearly feature interactions can indeed occur between the components of the combined system. In both cases, any approach to feature interaction based on knowledge about internal behaviour of the components, or requiring changes of the same, is not feasible.

The feature manager approaches of Tsang and Magill [TM97] and Marples and Magill [MM98] are unusual, as they both treat features (and legacy software) as *black boxes*. The former approach requires a learning phase of knowledge acquisition about a feature for later use by the resolution mechanism. However, during the learning phase only one feature can be present (or active) in the system. This can cause problems with features on the far end of a call – how can we guarantee that only one feature is activated (especially in a multi-vendor environment)? The latter approach overcomes this problem by proposing a *transactional* feature manager that does not require a learning phase (c.f. section 2). But, one drawback of this work is that so far, the emphasis is on detection of feature interactions, and Marples and Magill suggest that further work is required to improve the (relatively) weak resolution techniques. Currently, resolution is either manual or by simple precedences, but an automated, more sophisticated resolution technique is required.

*This author is funded by the EPSRC project *A Hybrid Approach to Feature Interaction Detection and Resolution*.

We aim to develop automated resolution techniques that can be integrated with the feature manager approach of Marples and Magill. Our work is part of a joint research project between the Feature Interaction Groups at the University of Glasgow and the University of Strathclyde [HFI], drawing on the individual experience of the partners in formal modelling and verification and the engineering aspects of switching systems. The relevance of the two aspects is reflected in our goal of a new *hybrid approach* to interaction detection and resolution; the overall goal of the project is to construct and refine algorithms for the resolution of interactions and develop a theory to guide the resolution process handled by the online feature manager. The latter will initially be embedded within the DESK test-bed as described in [MTMS95], with the longer term aim of developing a prototype implementation in a commercial switch.

The concept of a hybrid approach that combines the strength of offline and online techniques has certainly been considered before, notably [AC97]. Our approach is novel in that it neither uses state information (such as hookstate) for the formal model, nor requires knowledge available only at design time, but can extract all knowledge it requires from transmitted messages at runtime. Initially we are required to model example features in the offline-phase to identify general rules. However, after deployment, the feature manager does not require any a prior information about the features in the system.

Here, we concentrate on the problem of characterising the solution space, so that better resolution strategies can be developed. A crucial step involves formalising the transactional approach of Marples and Magill. Because we are concerned with legacy systems, we treat features and legacy systems as black box components, therefore the formal model is based on observable behaviour represented by input and output.

1.1 Overview of Paper

The relevant aspects of Marples and Magill's feature manager approach are summarised in section 2 and an overview of our hybrid approach is given in section 3. Section 4 introduces example features which will be used in sections 5 and 6 where we define features, the feature manager and the construction of the solution space. Section 7 introduces prototype rules for reducing the solution space, these are illustrated with an example. Finally we discuss the achievements, further work, and draw some conclusions.

2 The Feature Manager Approach

The aim of this section is to provide an overview of Marples and Magill's feature manager. See [MM98] for full details.

The role of the on-line feature manager is to mediate between new features, the legacy system, and the other components of the system (e.g. terminal devices). Specifically, the feature manager passes incoming messages from the switching hardware to the legacy software and all additional features; for simplicity we assume that this happens in parallel. The legacy software and the features are embedded within a cocoon that offers a consistent interface. For the remainder of this section we use the term *feature* to denote both the new features as well as the basic call software. (We expect the latter to be included in the legacy software.)

When a message is processed by a feature, two possible behaviours can occur: the message triggers a response (one or more messages) or it does not. All triggered responses are sent back to the feature manager, concluding with a *transaction finished* message. The latter is also sent by features not responding with a proper message[1].

The responses are collected by the feature manager and then evaluated. There are three possible outcomes:

1. no proper messages were received,
2. exactly one proper message were received,
3. multiple proper messages were received.

In the first two cases, the feature manager's role is straightforward: the basic call progresses (the former case) or the feature takes control of the call (the latter case). It is the third case which is interesting: the possibility that more than one feature might reply and the consequent potential for an interaction is indeed the motivation for the feature manager. Rather than resolve an interaction at this point, we explore the *space* of possible resolutions in the following way.

The feature manager stores the current state and initiates copies of the feature processes. The current state includes the list of events and other local information; it describes a *snapshot of the system from the viewpoint of the feature manager*. Assuming that at least one message has been received, the first one is fed to the copies of the features, the responses are gathered and processed. Part of this processing may involve sending and receiving further messages to and from the (copies of the) features.

This process terminates when there are no further responses; at this point, we have a sequence of messages and responses, which we consider as a branch in a *behaviour tree*. To construct the rest of the tree, we restore the initial state, farm out the next event, generate a new branch, and so on. Once all events are processed we have defined a full *behaviour tree*. It remains to select the most promising path in that tree. *Which path is to be chosen?* is the key question our research project will answer, by providing a theory of features and events that ultimately guides the choice. The theory will also provide mechanisms to detect unsuitable paths early in their construction process. For the close interaction of the underlying theory and the transactional online feature manager we refer to the proposed approach as *hybrid*, for further details we refer to section 3.

When the path is selected, it is committed by sending out the appropriate messages to the switching hardware. In detail, this means that all those messages produced by the features in response to the trigger event that are part of the selected branch need to be replayed and this time send out to the user or trunk. This playback is no longer piped through the feature manager's detection and resolution mechanism.

The approach depends on certain assumptions about the system, namely that we can create copies of all the associated processes. In a legacy system, one might emulate process copies if the system provides the functionality to reset a process to a certain state. In this case, to emulate multiple copies we maintain stacks of all messages, reset the control process and replay messages from the stack to obtain a process in the desired state. This is very much like failure recovery in database systems employing a rollback-commit strategy. For further details of the construction of the behaviour trees, see [CMM99].

The approach originally described by Marples and Magill only considers the reaction of features as described by the initial event and direct responses (i.e. feature behaviour that can be described as trigger-response(s) pair), whereas we will also consider hypothetical events to obtain a more complete picture of the solution space. This extension

[1] Any message apart from the *transaction finished* message.

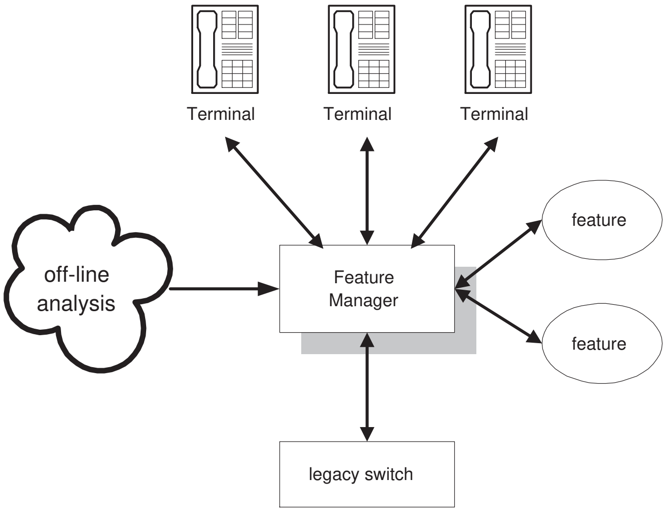

Figure 1: An Online Feature Manager – Systems Architecture

allows us to deal with features that require further input (e.g. from a user) and improves the quality of the solutions found, by considering the influence of a decision over a longer timescale. To motivate our concern with the longer term effects of a decision consider the following scenario: a user subscribes to three way calling and two features (A and B), both requiring a PIN. During a call both A and B request a PIN, triggered by the same event; the requests do not clash, hence an "ask for PIN" is committed to the user. The user enters a PIN, which is correct for one feature only (say A), so we have to resolve an interaction between "continue call" and "end call", as required by the features. We decide to continue the call, hence B is reset to the last stable state (awaiting a PIN) and the call continues. The user now flashes to initiate a three way call, and on the request to dial he dials a wrong number which coincidently is the correct PIN for feature B. The feature B now triggers its actions, whereas the user would have expected a message informing him/her that the telephone number was not recognised.

3 A Hybrid Approach

The idea behind the hybrid approach is described above, in this section we provide more detail and give the context for the remaining sections.

The role of our on-line feature manager is to mediate between new features, the legacy system, and the other components of the system (e.g. terminal devices). Figure 1 illustrates the role of the feature manager; for simplicity, we label all other system components as 'terminal' , for terminal devices, etc.

The feature manager must both detect potential interactions, between new features and the legacy system, and between new features themselves, *and* resolve them in a satisfactory way. Sometimes, the only resolution is to suppress one feature, then the question is which one; other times, it may be possible to interleave features or to run them sequentially, in a particular order. In all cases, the feature manager must be able to make that decision.

We have followed the the *transactional* approach proposed by Magill and Marples in [MM98] where the legacy switching system and the features are treated as black-boxes, i.e. they receive inputs and respond with outputs, embedded in transactional "cocoons". The cocoons permit a rollback and commit facility, allowing us to experiment with

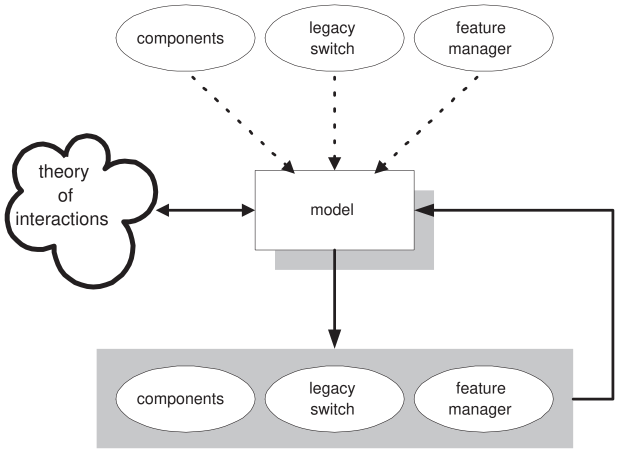

Figure 2: Role of Formal Modelling

sequences of possible inputs and responses and thereby choose the best possible resolution to a feature interaction (assuming one has been detected). The precise details of the transactional approach are not relevant here and they have been described in more detail in section 2 and in [CMM99]. What is relevant is to note that the information required in order to choose a best resolution is derived from an off-line analysis of feature behaviour. We stress, that this information is in the form of general rules (whose performance depends on the realism and detail of the features under experiment) but is independent of the features actually deployed in the system. Crucial to this analysis is a formal model of the system.

The role of the formal model is more than just an abstraction, or specification, of intended and/or required behaviour; it is actually part of the incremental process of developing those intentions/requirements. That is, it is an part of a *adaptive*, experimental process, as we have advocated in [Cal98] and [CR99]. Figure 2 illustrates this process.

The initial step in the process is to develop an (initial) formal model of the online system, i.e. the legacy switch, a relatively "uninformed" feature manager, and the system components. Dotted lines denote this step. The (initial) model provides us with a platform, or test-bed, in which to experiment with and reason about observable behaviour of the legacy system and the new features. Properties of this system will enable us to postulate theories of features and system events such as "feature x should have precedence over feature y", "event x should never be followed by event y, otherwise deadlock will occur", "event x and event y cannot be offered to a user simultaneously (e.g. a spoken announcement and a busy tone)".

These properties are results of exploring the solution space, which obviously has to be identified first. The resolution process can be visualised as filtering; the filter is formed by the properties and general criteria such as deadlocks, the input is the solution space. We call this process *pruning*.

The next step is to use the formal model to guide the feature manager, which will consequently alter system behaviour. We can then derive further properties of the altered system, postulate further theories and further enhance the feature manager, observe new system behaviour, and so on.

During the remainder of this paper we concentrate our attention on the initial step, specifically identifying the solution space.

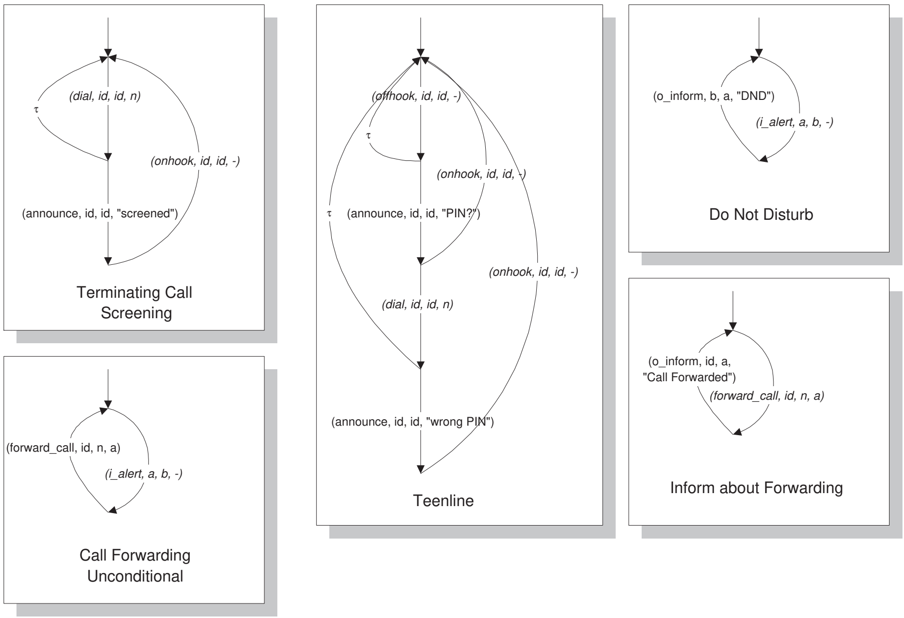

Figure 3: Pictorial Representation of Features

4 A Running Example

For the purpose of illustration we will refer to the following features:

1. Do Not Disturb: the subscriber cannot be disturbed by incoming calls, callers hear a pre-recorded message.

2. Call Forwarding Unconditional: all calls to the subscriber are forwarded to another number.

3. Terminating Call Screening: this features disallows the subscriber to dial a certain number.

4. Teenline: depending on time of day, requests a PIN when detecting an offhook message and disallows dialing in case the wrong PIN was entered.

5. Inform about Forwarding: informs the caller when a forwarding attempt is detected.

The first two features are identical to those used by Marples and Magill in [MM98], the third and fourth are designed to give examples that are not just trigger-response pairs. The last example is included to show the case when when one feature produces output that triggers another feature.

Figure 3 contains descriptions using labelled transition systems. Messages representing an input are in italics; all other messages are output. Transitions labelled τ indicate that the feature can make a transition without producing any observable behaviour.

5 Prelimary Definitions

This section introduces the notation and conventions used for describing features and the behaviour of the feature manager.

5.1 Messages

A message consists of an event (a type and possibly a value), an origin and destination and an associated I/O aspect. The origin and destination are numeric arguments, 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.
Convention: input messages are preceded by $^+$, output messages are preceded by $^-$.

We write a message m as $(eventtype, origin, destination, eventvalue)$ or when we are only concerned about the I/O aspect as ^+m or ^-m. "-" denotes the null value and, by abuse of notation, τ is also a special null message.

Example 1

The following are valid messages: $^+(dial, id, id, n)$, $^-(announce, id, id,$ "screened"$)$ and $^-(i\_alert, a, b, -)$.

We will need to describe relations between messages. Two messages a and b are said to be

name equivalent when their event types and values are equal, written $a \equiv b$.

equal when they are name equivalent and their I/O aspects are equal, written $a = b$.

inverses when they are name equivalent but their I/O aspects are not equal, written $a \doteq b$.

Example 2

- $^+(i\_alert, a, b, -) \equiv\ ^-(i\_alert, c, b, -)$ and $^+(i\_alert, a, b, -) \equiv\ ^+(i\_alert, c, b, -)$
- $^+(i\_alert, a, b, -) =\ ^+(i\_alert, c, b, -)$
- $^+(i\_alert, a, b, -) \doteq\ -(i\_alert, c, b, -)$

5.2 Features

Definition 1 (Alphabet) An alphabet is a set of messages partitioned into 2 non-empty sets: input messages and output messages.

Definition 2 (Trace) A trace over an alphabet is a non-empty finite sequence of elements of the alphabet starting with an input message. $First(t)$ is the first element in a trace t.

Convention: A trace is obtained by juxtapositioning elements of the alphabet.

Example 3

Consider the alphabet:

$\alpha = \{^{+}(dial, id, id, n),\ ^{-}(announce, id, id, \text{“screened”}),\ ^{+}(onhook, id, id, -),\ \tau\}$.

Possible traces over α are:

$^{+}(dial, id, id, n)^{-}(announce, id, id, \text{“screened”})$ and

$^{+}(onhook, id, id, -)\ ^{-}(announce, id, id, \text{“screened”})\ ^{-}(announce, id, id, \text{“screened”})$

The Terminating Call Screening Feature is defined by the trace set:

$$F_{TCS} = \{^{+}(dial, id, id, n)^{-}(announce, id, id, \text{“screened”})^{+}(onhook, id, id, -),\ ^{+}(dial, id, id, n)^{-}\tau\}.$$

Definition 3 (Feature) A feature is a set of traces over an alphabet. We assume the alphabet is minimal (i.e. each element of the alphabet occurs in at least one trace).

Convention: feature F_i has alphabet α_i.

Example 4

This example shows the trace sets for the features from section 4. The number at the right column will be used later to reference the respective trace.

Do Not Disturb:

$$F_{DND} = \{^{+}(i\_alert, a, b, -)^{-}(o\_inform, b, a, \text{“DND”})\} \quad (1)$$

Call Forwarding Unconditional:

$$F_{CFU} = \{^{+}(i\_alert, a, b, -)^{-}(forward\_call, id, n, a)\} \quad (2)$$

Terminating Call Screening:

$$F_{TCS} = \{^{+}(dial, id, id, n)^{-}(announce, id, id, \text{“screened”})^{+}(onhook, id, id, -), \quad (3)$$
$$^{+}(dial, id, id, n)^{-}\tau\} \quad (4)$$

Teenline:

$$F_{TL} = \{^{+}(offhook, id, id, -)^{-}(announce, id, id, \text{“PIN?”})^{+}(onhook, id, id, -), \quad (5)$$
$$^{+}(offhook, id, id, -)^{-}\tau, \quad (6)$$
$$^{+}(offhook, id, id, -)^{-}(announce, id, id, \text{“PIN?”})^{+}(dial, id, id, n)^{-}\tau, \quad (7)$$
$$^{+}(offhook, id, id, -)^{-}(announce, id, id, \text{“PIN?”})^{+}(dial, id, id, n)$$
$$^{-}(announce, id, id, \text{“wrong PIN”})^{+}(onhook, id, id, -)\} \quad (8)$$

Inform about Forwarding:

$$F_{IAF} = \{^{+}(forward\_call, id, n, a)^{-}(o\_inform, id, a, \text{“Call Forwarded”})\} \quad (9)$$

5.3 Feature Interference

Usually the feature interaction community talks about feature interactions meaning “points of contact” between two features, often not discriminating between desired and undesired interactions. It is generally difficult to decide whether or not an interaction is desired, as factors like user expectations play a major role in this classification. To avoid this discussion, we will use the term *interference* here to mean that two or more features behaviours interfere with each other.

Definition 4 (Feature Interference) n features $F_1 ... F_n$ with respective alphabets $\alpha_1 ... \alpha_n$ interfere iff

$$\exists i, j : ((1 \leq i, j \leq n) \wedge (i \neq j)) \rightarrow (\exists^{+} a \in \alpha_i \rightarrow (\exists b \in \alpha_j : a \equiv b))$$

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.

5.4 The Feature Manager

The feature manager's purpose is to detect and resolves interferences, hence the feature manager has to distinguish two main case which need to be treated differently: the features in question interfere or they do not interfere in the sense of Definition 4.
Let $F_1, ..., F_n$ be features and let e be an input message in $\alpha_1 \cup ... \cup \alpha_n$.

Definition 5 (Feature Manager) The feature manager is a function, defined as:

$$FM(e, F_1, ..., F_n) = \begin{cases} Extract(Prune(Construct(F_1, ..., F_n))), e) & \\ \qquad \text{if } F_1, ..., F_n \text{ interfere} \\ Extract((F_1 \cup ... \cup F_n), e) & \\ \qquad \text{otherwise} \end{cases}$$

$Extract(T, e)$ returns a set of traces T' such that $T' \subseteq T$ and $\forall t : (t \in T') \rightarrow First(t) = e$.
$Prune(T)$ returns a set of traces T', such that $T' \subseteq T$, where all the traces in T' represent possible resolutions.
$Construct(F_1, ..., F_n)$ constructs the solution space and will be considered in detail in section 6.

The reader might be surprised to find that *Extract* can return more than one trace, after all we are looking for a single resolution. To motivate our definition, consider a simple single feature example: $F = \{^+a^-b^+c^-d,\ ^+a^-b^+e^-f\}$. It is obvious that both traces are equally good resolutions when the trigger is a, but when the call progresses either c or d will occur as trigger, forcing the other trace to be irrelevant. However, this is not an interaction and we expect the feature manager to return both traces. This is meaningful because the decision which trace finally determines the behaviour is dependant on a later trigger event. Therefore, we cannot make the decision at this point and must return a set. Similar scenarios can arise between multiple features, thus leading to the above understanding of *Extract*.

6 The Solution Space

The solution space contains all possible traces representing the combined behaviour of the features under consideration. That is, it includes all possible traces that can arise in a system with those features present. The solution space for non-interfering features is simply the union of their behaviour; For interfering features, *Construct* finds the traces that constitute the solution space and prune and extract reduce that space.

A very simple example, when 2 features (f_1 and f_2) are involved, is illustrated in figure 4. It should be noted that the solution space might contain many traces that lead to live- or deadlocks, as well as to the violation of required properties (i.e. there might be traces that do present incorrect behaviour). The reader can imagine what happens with the shown solution space when further features are included: for 3 features there are already 15 possibilities. Including finer grain interleaving, i.e. interleaving individual messages, this numbers grows even faster. Here, we define the full solution space for any number of features allowing *fine grain* interleaving.

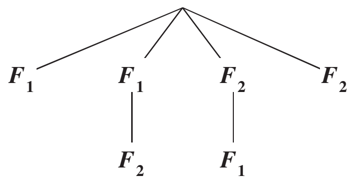

Figure 4: A simple Solution Space

The role of the *Construct* function is to construct the solution space, i.e. the set of all potential resolutions. The complexity of the definition reflects the complexity of the task: *Construct* returns a set containing all the individual features traces and all traces representing interleavings of multiple features. The latter are constructed using the function *overlap*.

$$\begin{aligned}
Construct(F_1, ..., F_n) &= SingleFeatures(F_1, ..., F_n) \cup MultipleFeatures(F_1, ..., F_n) \\
SingleFeatures(F_1, ..., F_n) &= \bigcup_{i=1}^{n} (F_1, ..., F_n) \\
MultipleFeatures(F_1, ..., F_n) &= \bigcup_{i=1}^{|F_1|*...*|F_n|*n!} overlap(t_{gj}, ..., t_{hk})
\end{aligned}$$

We assume the following conditions on the variables: $g \neq h$ and $g, h \in \{1..n\}$ and $j \in \{1..|F_g|\}$ and $k \in \{1..|F_h|\}$.

Example 5

This example constructs the solution space from the Do not Disturb and Call Forwarding Unconditional features.
$Construct(F_{DND}, F_{CFU}) = SingleFeatures(F_{DND}, F_{CFU}) \cup MultipleFeatures(F_{DND}, F_{CFU}) = F_{DND} \cup F_{CFU} \cup overlap(t_{(1)}, t_{(2)})$
Note: *MultipleFeatures* returns a singleton set in this example.

A crucial function is $overlap(t_1, ...t_n)$ which returns the set of all "overlapping interleaving" sequences that can be generated from the t_i in the input. This includes all those generated from just 2 traces, those from 3 traces, up to those from n traces. We consider the function, $oi2$, that computes all "overlapping interleavings" of 2 traces first and then build up a function for n traces.
Assumptions:
Let α_i be minimal and $1 \leq i, j \leq 2$ and $i \neq j$.
Let $T(t_1, t_2) = \{t \mid t \text{ is an overlapping interleaving of } t_1 \text{ and } t_2\}$ s.t. T is maximal.
$oi2$ returns either an empty set or a set of traces, as follows:

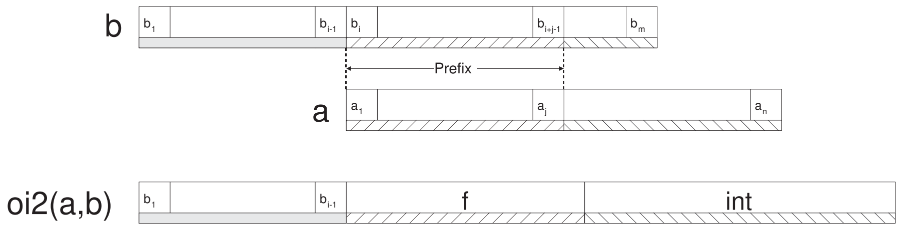

Figure 5: Overlapping Interleavings

$$oi2(t_1,t_2) = \begin{cases} \{\} & \nexists a : (\ {}^{+}a,\ {}^{-}a \in \alpha_i) \rightarrow\ {}^{+}a = First(t_j) \\ T(t_1,t_2) & \text{otherwise} \end{cases}$$

Example 6

A trivial example overlapping interleaving is:
$oi2(t_{(2)}, t_{(3)}) = \{\}$ Reason: The alphabets are distinct.

Example 7

Further example overlapping interleavings are:
$oi2(t_{(9)}, t_{(2)}) = T_1$ Reason: The output of $t_{(2)}$ is input to $t_{(9)}$.
$oi2(t_{(3)}, t_{(8)}) = T_2$ Reason: $t_{(3)}$ and $t_{(8)}$ have a common input.

We have to define precisely what an "overlapping interleaving" is. The following definition, illustrated with figure 5, makes precise this concept.

In the following, to increase legibility we substitute $a = a_1...a_n$ for t_1 and $b = b_1...b_m$ for t_2. Let *concat* be a function concatenating sequences. Let i be the first occurrence of the longest prefix of a occurring[2] in b. Let j be the length of that prefix.

Definition 6 A trace t is an overlapping interleaving of t_1 and t_2, if and only if

$$t = concat(b_1...b_{i-1}, f(i,j,a,b), int(a_{j+1}...a_n, b_{j+i}...b_m))$$

To complete this definition, we have to define the function f, which computes the event sequence arising from the prefix, and the function *int*, which computes an interleaving. f is defined as follows:

[2]a message m occurs in a and b if there are messages a_x and b_y such that $a_x \equiv b_y$.

$$f(x,1,a,b) = \begin{cases} b_x & \text{if } a_1 \equiv b_x \\ b_x a_1 & \text{if } a_1 \doteq b_x \end{cases}$$

$$f(x,y,a,b) = \begin{cases} concat(f(x,y-1,a,b),a_{y-1}) & \text{if } a_{y-1} \equiv b_{x+y-1} \\ concat(f(x,y-1,a,b),a_{y-1},b_{x+y-1}) & \text{if } a_{y-1} \doteq b_{x+y-1} \text{ and} \\ & a_{y-1} \text{ is an output message} \\ concat(f(x,y-1,a,b),b_{x+y-1},a_{y-1}) & \text{if } a_{y-1} \doteq b_{x+y-1} \text{ and} \\ & b_{x+y-1} \text{ is an output message} \end{cases}$$

int is defined as follows: $int(s_1, ..., s_n)$, an interleaving of sequences $s_1, ..., s_n$ with length $|s_1|, ..., |s_n|$ respectively, is a new sequence S

- whose length is $\sum_{i=1}^{n} |s_i|$,
- $\forall a : (a \in S) \leftrightarrow (a \in s_1 \vee ... \vee a \in s_n)$,
- which preserves the relative order of the elements (i.e. if an element a occurs in a sequence before b, then so they do in S).

We are now in a position to present more interesting examples which illustrate the above definitions. For readability we represent each message by its event and argument (if required) only.

Example 8

oi2($\mathbf{t_{(9)}}, \mathbf{t_{(2)}}$) = the set of all overlapping interleavings between $t_{(9)}$ and $t_{(2)}$. Note that the output of $t_{(2)}$ is the input to $t_{(9)}$. The longest prefix of $t_{(9)}$ in $t_{(2)}$ is the sequence $(forward\_call)$, occurring at position 2 in $t_{(2)}$. There is only one $t \in oi2(t_{(9)}, t_{(2)})$, namely:

$t = concat(^+(i\_alert), f(1,1,t_{(9)},t_{(2)}), int(^-(o\_inform, \text{“Call Forwarded”})))$

$= concat(^+(i\_alert), ^-(forward\_call)^+(forward\_call), ^-(o\_inform, \text{“Call Forwarded”}))$

$= ^+(i\_alert)^-(forward\_call)^+(forward\_call)^-(o\_inform, \text{“Call Forwarded”})$

Example 9

oi2($\mathbf{t_{(3)}}, \mathbf{t_{(8)}}$) = the set of all overlapping interleavings between $t_{(3)}$ and $t_{(8)}$. $t_{(3)}$ and $t_{(8)}$ have a common input. The longest prefix of $t_{(3)}$ in $t_{(8)}$ is the sequence $(dial)$, occurring at position 3 in $t_{(8)}$. There are several $t \in oi2(t_{(3)}, t_{(8)})$, as we shall show:

$$\begin{aligned} t &= concat(^+(offhook)^-(announce, \text{“PIN?”}), f(1,1,t_{(3)},t_{(8)}), \\ &\qquad int(^-(announce, \text{“wrong PIN”})^+(onhook), ^-(announce, \text{“screened”})^+(onhook))) \\ &= concat(^+(offhook)^-(announce \text{“PIN?”}), (dial), \\ &\qquad int(^-(announce, \text{“wrong PIN”})^+(onhook), ^-(announce, \text{“screened”})^+(onhook))) \end{aligned}$$

As int can return four different values, we obtain four different traces t that are overlapping interleavings of the two input traces. All four traces have the common beginning as shown, their respective endings are:

$^-(announce, \text{“wrong PIN”})^+(onhook)^-(announce, \text{“screened”})^+(onhook)$

$^-(announce, \text{“wrong PIN”})^-(announce, \text{“screened”})^+(onhook)^+(onhook)$

$^-(announce, \text{“screened”})^-(announce, \text{“wrong PIN”})^+(onhook)^+(onhook)$

$^-(announce, \text{“screened”})^+(onhook)^-(announce, \text{“wrong PIN”})^+(onhook)$

Finally we consider the general *overlap* function, operating on n traces. We distinguish two cases:

$$
\begin{array}{lll}
a) \text{ if } n = 2 & overlap(t_1, ..., t_n) = oi2(t_1, t_n) \\
b) \text{ otherwise} & overlap(t_1, ..., t_n) = oi2(overlap(choice(n-1, t_1, ..., t_n)), choice(1, t_1, ..., t_n) \\
 & \qquad\qquad\qquad\qquad \cup\, overlap(choice(n-1, t_1, ..., t_n)
\end{array}
$$

The function $choice(k, t_1, ..., t_n)$ shall not be described here in detail; the idea is that it chooses k out of the provided n traces (without replacement). We assume that the trace $choice(1, ...)$ is not in the set $choice(n-1, ...)$.

6.1 Implementation

An implementation of this process has been developed using the Python[3] programming language.

In example 10 we show some sample output. This example shows the complete solution space for the Teenline (TL) and Terminating Call Screening (TCS) features defined earlier. Traces 11-13 are the original traces from the TL feature, traces 14-15 are from the TCS feature, trace 0 is a combined trace of the two features which incidently is identical to a trace from the Teenline feature.

Example 10

```
       +------------ output without destination and origin (for brevity)
       |       +---- the feature manager as defined
       |       |
>>> printshort(fm("offhook", fiwtcs, fiwtl))
These features interfere, finding resolution ...

    0 :  ( + , offhook , - ) ( - , announce , PIN? ) ( + , dial , n ) ( - , tau )
    1 :  ( + , offhook , - ) ( - , announce , PIN? ) ( + , dial , n ) ( - , announce , wrong PIN ) ( - , tau )
         ( + , onhook , - )
    2 :  ( + , offhook , - ) ( - , announce , PIN? ) ( + , dial , n ) ( - , announce , wrong PIN ) ( + , onhook , - )
         ( - , tau )
    3 :  ( + , offhook , - ) ( - , announce , PIN? ) ( + , dial , n ) ( - , tau ) ( - , announce , wrong PIN )
         ( + , onhook , - )
    4 :  ( + , offhook , - ) ( - , announce , PIN? ) ( + , dial , n ) ( - , announce , screened ) ( + , onhook , - )
         ( - , tau )
    5 :  ( + , offhook , - ) ( - , announce , PIN? ) ( + , dial , n ) ( - , announce , screened ) ( - , tau )
         ( + , onhook , - )
    6 :  ( + , offhook , - ) ( - , announce , PIN? ) ( + , dial , n ) ( - , tau ) ( - , announce , screened )
         ( + , onhook , - )
    7 :  ( + , offhook , - ) ( - , announce , PIN? ) ( + , dial , n ) ( - , announce , wrong PIN ) ( - , announce , screened )
         ( + , onhook , - ) ( + , onhook , - )
    8 :  ( + , offhook , - ) ( - , announce , PIN? ) ( + , dial , n ) ( - , announce , wrong PIN ) ( + , onhook , - )
         ( - , announce , screened ) ( + , onhook , - )
    9 :  ( + , offhook , - ) ( - , announce , PIN? ) ( + , dial , n ) ( - , announce , screened ) ( - , announce , wrong PIN )
         ( + , onhook , - ) ( + , onhook , - )
   10 :  ( + , offhook , - ) ( - , announce , PIN? ) ( + , dial , n ) ( - , announce , screened ) ( + , onhook , - )
         ( - , announce , wrong PIN ) ( + , onhook , - )
   11 :  ( + , offhook , - ) ( - , announce , PIN? ) ( + , onhook , - )
   12 :  ( + , offhook , - ) ( - , tau )
   13 :  ( + , offhook , - ) ( - , announce , PIN? ) ( + , dial , n ) ( - , announce , wrong PIN ) ( + , onhook , - )
   14 :  ( + , dial , n ) ( - , tau )
   15 :  ( + , dial , n ) ( - , announce , screened ) ( + , onhook , - )
```

7 Pruning

Solution spaces as defined above can be very large sets (of traces). They need to be reduced into a set of meaningful (or correct) resolutions. Ideally, the reduced set contains only traces that cannot be discriminated with the available information. Nevertheless, when the call progresses the relevant information will be collected (e.g. by user input) and the appropriate "best" traces can be chosen.

[3] http://www.python.org

Some rules for pruning the solution space have been identified based on observations of examples and common sense. Note, they are a starting point and will be replaced and/or completed by others as this work progresses. Some prototype rules are:

1. Announcement messages that occur after each other in a trace without an input message inbetween do not provide any meaningful behaviour to the user.
2. Tones and announcements played to a user after an *onhook* is received will never be received.
3. The unobservable message τ can be removed from traces.
4. Duplicates of traces can be removed (note that this was already done in the example – hence trace 0).

Example 11

We can now analyse example 10 with respect to these rules. The traces that can be removed (traces that can be removed with one rule are not tested for the others rules) are:

- apply rule 1 to remove *traces 7 and 9*.
- apply rule 2 to remove *traces 8 and 10*.
- apply rule 3 to *traces 4, 5 and 6*. Note that the resulting traces are equal.
- apply rule 3 to *traces 1, 2 and 3*. Note that the resulting traces are equal to *trace 13*.
- apply rule 4 and remove traces 1, 2, 3, 4 and 5

Example 11 shows that we can reduce the solution space from 16 to 7 traces using the four pruning rules. *Trace 6* should be removed as well, the behaviour is not consistent: the user dials the PIN and gets an announcement that the call is screened. However, we have not yet formulated a general rule which would capture this. The remaining traces can be returned to the final step (extraction); it should be obvious that they fulfil the previously mentioned criteria for the ideal set.

8 Discussion

Using our formalisation, the solution space for the Do Not Disturb and Call Forwarding Unconditional Features results in 4 traces (the trace of each feature and 2 combined traces). These 4 traces are identical to those reported by Marples and Magill [MM98], hence we can claim that our approach accurately models their achievements.

To understand the additional capabilities of our work we need to consider examples 8 and 9.

In example 8 we constructed the $MultipleFeatures$ part of the solution space for Inform about Forwarding and Call Forwarding Unconditional, which turned out to be a single trace. When we compute the complete solution space, it turns out that it consists of three traces: the trace from example 8 and the two traces describing each feature.

Example 9 is very similar, although here we have used the Terminating Call Screening and Teenline features which are themselves more complicated. The solution space consists of 16 different traces: two from Terminating Call Screening, 4 from Teenline and the 4 traces shown in the example 9. The remaining 6 traces result from overlappings of the traces t_3 with t_5, t_6 and t_7 as well as t_4 with t_5, t_6, t_7 and t_8.

These results show that it is not just the number of features involved but also the number of traces of each feature, as well as their length, that can have an impact on the size of the solution space.

In summary, our method is able to reproduce the results of Marples and Magill and in addition, can deal with features that are represented by longer traces, containing multiple inputs and outputs. Features are handled as black boxes, only represented by their observable behaviour.

By outlining the full solution space, we have shown the complexity of the same. This result would be discouraging, if we had not at the same time shown how this complexity is introduced. This fact leads to clear ideas for resolution mechanisms, as now we have formed a sound understanding of the nature of *all* possible solutions and at the same time identified that several generated traces are unsuitable to resolve interactions.

So far, we can identify all possible traces, or in Marples and Magill's words, *branches.* Further, we have suggested techniques, for removing unsuitable branches, leaving only a small choice for the *best* trace. This opens new questions: *How can we evaluate the "quality" of a trace?* and *Is there always a best trace?*. In answering the first question, we should certainly consider that we want as many features as possible to cooperate; hence traces involving several features are preferable to those involving just one feature. Our intuitive feeling for the second question is, that we will have classes of equally good resolutions rather than just one best resolution.

9 Further Work

As we have shown in the previous section, the solution space is quite large, but as indicated, we are able to prune this space significantly. This pruning will be done by the function *Prune*, which is not specified in detail yet, and thus remains further work.

The initial idea of pruning based on the removal of paths leading to dead- and live-locks is not suitable, as those concepts do not have a meaning in our approach. However, specified properties of the system, such as event x has to occur immediately before y, will allow to reduce the space further (as was shown in example 11). During the process of developing a more formal background to the *Prune* function, we hope to detect general scenarios about sequences of features, which form the basis for the challenging task of integrating *Prune* and *Construct*. This will lead to the proposed solution where we can identify unsuitable traces early in their construction process and hence remove them without spending more effort on exploring them in full. The reader should think about this as a dovetail of *Prune* and *Construct*, reducing the complexity of both significantly.

Further we hope to be able to show that there is a certain constant number of features, say c, where an interaction detected between c or more features means that we can already detect an interaction between $c-1$ or fewer features. This would reduce the complexity in systems with many features ($\gg c$) dramatically as we only need to consider branches with up to c features interleaved.

As the whole process is iterative, the knowledge gained through this work can be fed back and better solutions can be sought. Nevertheless, this first iteration will provide algorithms that are to be implemented in the test-bed and in a prototype feature manager in an operational switching environment.

10 Conclusions

The final result of this research will provide a theory of features and interactions that can be used to guide an online feature manager. This theory is fundamental for the design of algorithms to resolve interactions with minimal knowledge about the internal states of the system. The feature manager can handle any black-box system, as long as the components can be encapsulated in transactional cocoons. The feature manager does not require design information about any of the components – all that is required is a description of the message format.

Although the current construction algorithm is very time and space inefficient, the integration with pruning will reduce this complexity significantly and thus provide an invaluable basis for constructing more efficient algorithms.

Working together with Marples and Magill, we are able to ensure that our work, although rather theoretical, does indeed fill the gap in their current work. That is, we are able to provide strong resolution mechanisms, based on the theoretical work discussed here, that smoothly integrate with their transactional detection approach.

Acknowledgements

The author thanks Evan Magill and Dave Marples for several stimulating discussions about their approach. I would also like to thank the reviewers for pointing out shortcomings of the initial version of this paper, which hinted at unclear expression of several major points. Finally, I thank Muffy Calder for her invaluable help in phrasing ideas and making me explain my ideas often enough to spot mistakes in the detail.

References

[AC97] I. Aggoun and P. Combes. Observers in the SCE and SEE to detect and resolve feature interactions. In [DBL97], pages 198–212.

[Cal98] M. Calder. What use are formal design and analysis methods to telecommunications services? In [KB98], pages 23–31.

[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.

[CR99] M. Calder and S. Reiff. Modelling legacy telecommunications switching systems for interaction analysis. 1999. To appear in *Systems Engineering for Business Process Change*, Springer Verlag.

[DBL97] P. Dini, R. Boutaba, and L. Logrippo, editors *Proceedings of the 4^{th} International Workshop on Feature Interactions in Telecommunications Systems*, Montreal, Canada, June 1997.

[KB98] K. Kimbler and L. G. Bouma, editors *Proceedings of the 5^{th} International Workshop on Feature Interactions in Telecommunications and Software Systems*,Lund, Sweden, September 1998.

[HFI] HFIG. The Hybrid Feature Interaction Group, University of Glasgow and University of Strathclyde: `http://www.dcs.gla.ac.uk/research/hfig`.

[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.

[MTMS95] D. Marples, S. Tsang, E. H. Magill, and D. G. Smith. A platform for modelling feature interaction detection and resolution techniques. In K. E. Cheng and T. Ohta, editors, *Proceedings of the 3^{rd} International Workshop on Feature Interactions in Telecommunications Systems*, pages 185–199, Kyoto, Japan, October 1995.

[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.