Hybrid Solutions to the Feature Interaction Problem

Muffy Calder[1], Mario Kolberg[2], Evan Magill[2], Dave Marples[3], Stephan Reiff-Marganiec[2]

[1]*University of Glasgow*
Computing Science
17 Lilybank Gardens
Glasgow G12 8RZ, UK
muffy@dcs.gla.ac.uk

[2]*University of Stirling*
Computing Science & Mathematics
Stirling FK9 4LA, UK
{mko, ehm, srm}@cs.stir.ac.uk

[3]*Global Inventures Inc*
Bishop Ranch 2
2694 Bishop Drive, Suite 27
San Ramon, CA, 94583
dmarples@iee.org

Abstract. In this paper we assume a competitive marketplace where the features are developed by different enterprises, which cannot or will not exchange information. We present a classification of feature interaction in this setting and introduce an on-line technique which serves as a basis for the two novel *hybrid* approaches presented. These approaches are hybrid as they are neither strictly off-line nor on-line, but combine aspects of both. The two approaches address different kinds of feature interactions, and thus are complimentary. Together they provide a complete solution by addressing interaction detection and resolution. We illustrate the techniques within the communication networks domain.

1 Introduction

The provision of communications and multi-media services in software is a major growth industry. Increasingly end-users and vendors find software solutions by combining software components, which may or may not be distributed across a network. Typically, such components are added incrementally, at various stages in the lifecycle. When deployed, each component functions well on its own. In an open market components will be provided by different vendors who will have limited ability to ensure that their products are compatible with those of other vendors. Indeed experience suggests that even within one enterprise new components may not be compatible with their existing products [4]. In call control systems components enhancing a basic service are referred to as *features*. Incompatibility between features is referred to as *feature interaction*.

It is important to note that an interaction between two features in call processing systems corresponds to a single *scenario*: which party is subscribing to which particular feature and who is calling whom. Consequently, a *single* pair of features may be involved in a number of *different* call scenarios which cause feature interactions.

As services proliferate with increasing market de-regulation, so too do the problems of feature interactions, particularly in the presence of legacy code and numerous third party vendors. In fact, feature interaction may jeopardise the goal of an open service provision within the telecommunications market and indeed any (distributed) software system employing a variety of software components from a variety of sources.

Within the context of call control software the terms interworking and incompatibility have well understood meanings. Features must *interwork* to share a (communications) resource. The features may interwork *explicitly* through an exchange of information with each other, or *implicitly* through this shared resource. In this second case, the features often have no knowledge that the other exists, however they may be aware of another controlling influence from the behaviour of the resource. Explicit interworking is possible in advanced session control architectures and was first proposed in TINA [14, 17], where interworking was both controlled and expected. Implicit interworking is common in traditional telephony and occurs simply because features attempt to share a resource.

When features interwork to share communication resources, they are *compatible* if the joint behaviour of the resource is acceptable. Typically this requires that the behaviour of the resource does not break the expectations of any controlling feature. Compatibility does *not* refer to simple coding errors, nor to the adherence of interfaces or protocols, but to the adequate behaviour of a resource under the joint control of interworking features.

A substantial body of work [11, 13, 5] exists on dealing with feature interactions. The work embraces both off-line and on-line approaches. Briefly, off-line approaches are those that are applicable at design-time whereas on-line approaches are applicable at run-time. The former being most useful at the early stages of the software lifecycle, the latter during testing and deployment [4]. The literature differentiates between the *detection* of interactions, and the *resolution* of interactions. Traditionally, there has been scarce interest in *avoidance* of interactions because of the enormous architectural changes required.

Off-line approaches are often based on the application of formal methods, and as such require considerable information of each individual software increment. Increasingly, as the market becomes more competitive, this information may be difficult to obtain. Also, as the number of features increases, the work in analysing pair-wise interactions increases with the square of the number of features. With a large number of features in an open market, this will quickly become untenable.

In contrast, on-line approaches simply carry out checks as required. Clearly there are computing resource issues, but the major issue is the paucity of feature information available to on-line approaches. This leads to an ability to *detect* interactions, but a poor ability to *resolve*. Note that off-line approaches at the early stages of development can resolve interactions by indicating how to re-design the features.

Strangely, little attention has been given to combining these approaches. Yet a *hybrid* approach offers the potential to develop a scalable on-line approach, informed by sufficient off-line information to allow not just detection, but also resolution. We suspect this lack of interest to be caused in part by the different academic background required by the two approaches. Another serious problem for hybrid approaches is the completely different representations of information employed in on-line and off-line approaches.

This paper describes two approaches that each combine aspects of on-line and off-line behaviour. The concept of such a hybrid approach was first suggested in [6]. Here, we report on results of a thorough investigation of those ideas, including implementation and experimentation. Preliminary results have been published thoughout the investigation [15, 8, 7, 21]. In particular, we have tried to gain experience and exploit the advantages of both: a high level of abstraction in the approach presented in Section 4, and a lower level abstraction in the approach presented in Section 5. Both approaches are based on the same on-line technique, which is discussed in Section 3. The two approaches address different classes of interactions, and neither provides a complete solution. However, if the approaches are combined they provide a complete solution, as shown by the experimental evaluation (see section 6). The next section contains a discussion of four classes of interaction that occur.

2 Types of Interaction

We can identify four types of interaction based upon a model where an event triggers the system to perform an action. This action may itself trigger the system to perform further actions in a sequence of 'micro-steps'. When there are no further actions pending the system is said to have returned to a relaxed state. Systems of this type were described by Blom [2].

2.1 Shared Trigger Interactions

Shared trigger interactions (STIs) occur when more than one feature wishes to respond to a stimulus. It is possible to detect this type of interaction automatically simply by examining the set of responses from the features to a stimulus. If more than one feature responds, then an STI has occurred.

The "legendary" example of Call Waiting and Call Forwarding Busy is a typical case of an STI: both features are triggered by an incoming call to a busy line. This is the most dangerous type of feature interaction with respect to the stability of the system. There is usually very little protection built into the switch software against the effects that this type of feature interaction can cause and STIs are always considered to be detrimental. Typically an STI could result in a terminal being told to go into the busy state and the ringing out state at the same time; which one the terminal actually reaches being determined by the order in which the messages arrive, that is STIs usually appear as race conditions. It is not uncommon for this to happen in switching software and so it is extraordinarily important that STIs are prevented. Sections 3 and 5 discuss automatic detection and resolution of this type of interaction.

2.2 Sequential Action Interactions

Sequential action interactions (SAIs) occur when the responses generated by one feature cause another feature to be triggered. Thus, although all features do not respond to the same stimulus, from the perspective of the user all of the results occur as a direct result of their action. It is difficult for users to be able to distinguish between SAIs and STIs since the micro-steps inherent in an SAI are hidden from the user.

Call Forwarding Unconditional (CFU) causing Do Not Disturb (DND) to be triggered on a terminal would be a typical example of an SAI. Although SAIs may cause unexpected system operation, they do not usually lead to damage or errant behavior of the system overall because each of the individual features is operating in its 'standard' environment.

SAIs lead to new emergent behaviors of the system which may be desirable or undesirable depending upon the individual use case being considered. Two examples illustrate this point. The interaction between Call Forward Unconditional (CFU) and Call Waiting (CW), where a party that is forwarded to another terminal will receive CW treatment at the second terminal, is probably a desirable new behavior. The interaction between Hot Line (HL) and Terminating Call Screening (TCS), where a HL call from A to B will never be completed if A is on B's TCS list is probably not desirable.

Because some SAIs are beneficial, as in the first case above, it is not sufficient to say that when multiple features respond in the process of handling an original event then this is an interaction. Section 4 introduces an automatic qualification of SAIs to solve this problem.

2.3 Looping Interactions

Looping Interactions (LIs) occur when multiple features, or even multiple instances of a single feature, operate in concert to cause a cyclic sequence of events to occur. LIs are a special case of SAIs. This cycle can be detected since the call never reverts back to a relaxed state. The loop can be detected by simply detecting an event/response chain that is longer than would naturally occur in normal call processing. This is a slightly imperfect solution since it always allows the possibility that a perfectly legal, but long, sequence of events would lead to a false detection as a looping interaction.

2.4 Missed Trigger Interactions

Interactions where the presence of one feature in the system prevents the second feature from operating at all are termed Missed Trigger Interactions (MTIs). Although these interactions cannot be detected during runtime it should be possible, with an extension of the approach presented by Wakahara[26] and Kuisch et al.[18] to apply extended feature specification principles to provide additional information about the trigger and return points for the feature.

3 An On-line Transactional Approach

3.1 On-line Technique

An approach extending the earlier work of Homayoon and Singh[12], Schessel[24] and Cain[3] was developed in which features are represented as finite state automata which only perform state transitions in response to external events. In the absence of these external events they perform no action of their own. Even timers are treated as being external stimuli. Call processing events are offered to all features to see what action they would perform if they were allowed to progress. This is in the spirit of these earlier techniques, but allows application in environments where prior information about the features is not available.

This is an interesting approach but, unfortunately, once a feature has been triggered, the event has been consumed and the internal state of the feature has been updated. It would appear that explicit code must be added into every feature to introduce some sort of ‘proposed’ phase where its responses have been sent back to the system but they have not been accepted for commitment to the hardware. This would require extensive changes to the design of both current and future features.

An alternative to this significant modification to existing architectures is presented by allowing features to be rolled back using the UNIX *fork* mechanism. This allows for an exact copy of the operational feature to be created, with all of the process and state information of the parent. The triggering event can then be passed to the copy. There are now two copies of the feature in memory; an unmodified copy representing the state of the feature before receiving the event, and a modified copy which *has* received the event and responded.

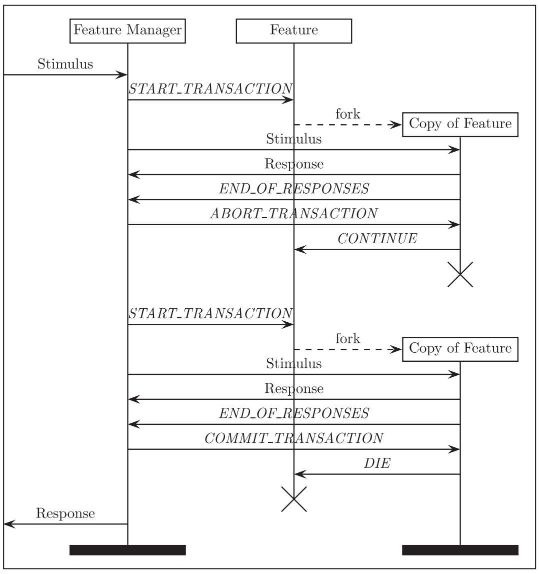

Figure 1: Feature Manipulation using Transactional messages

A feature manager which has performed this splitting process now has a choice: it can commit the responses to the event from the forked feature back to the hardware or it can throw the responses away. If it chooses the former the forked copy of the feature is allowed to continue and the unforked copy is deleted, if the latter then the forked copy is deleted and the feature is returned to its initial state. This is shown in Figure 1.

By using this approach, it is possible to effect rollback control over a feature [27]. Further, since this control is performed at the operating system level, no modification to the feature is required to allow its use. The approach shares similarities with software fault tolerance solutions [20] and error recovery techniques such as documented by Shin [25] and Campbell and Randell [10].

To implement this approach a feature is nested inside a feature controller. The feature is a conventional non-transactional state machine implementation providing call processing functionality and the feature controller is responsible for cocooning this to make it work correctly with the feature manager. We also refer to the controller as a cocoon. This is shown in Figure 2. The use of an all-enveloping cocoon makes it very straightforward to incorporate legacy code into the system since all of the functionality that the feature manager requires can be incorporated into the cocoon itself. Indeed, when using an operating system which supports shared libraries the functionality that the feature manager expects from the feature can be upgraded without even needing to re-compile the feature. This provides a very powerful

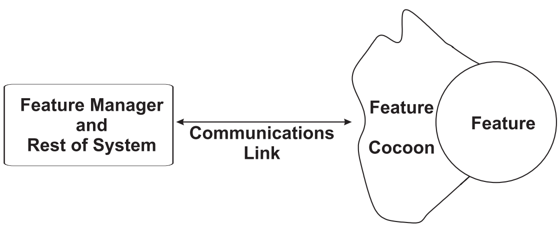

Figure 2: Arrangement of Feature and Cocooning Environment

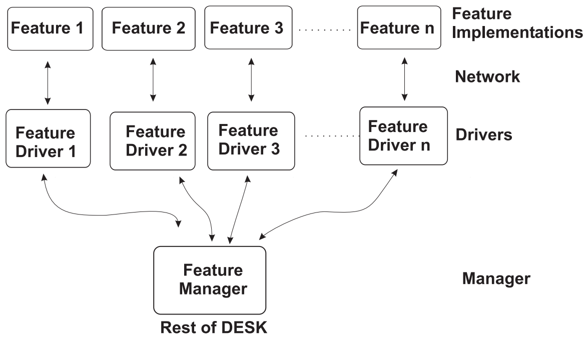

Figure 3: Relationship between the Feature Manager and the Features

upgrade capability.

This rollback technique has been coupled with a conventional feature manager to develop an approach which is capable of using the proposed responses to events to decide which features to allow to progress. The approach presents the stimuli to all of the features in parallel and uses the proposed responses back from the features to detect interaction between them. This can be done during runtime and no prior knowledge about the features is required.

3.2 System Operation

From the point of view of the algorithms that are employed, the structure of the system is as shown in Figure 3. The system consists of the switching and control system which provides stimuli to the feature manager. The feature manager is responsible for distributing these stimuli out to the features via the feature drivers and collecting the responses back again. It is within the feature manager that the feature interaction detection and resolution techniques are implemented. The feature manager has the capability to intercept, block and delay messages as required.

The experimental system is build on top of a switching platform created for the purpose of this work, known as DESK. Initially the system, comprising of the switch (DESK), the feature manager and the features, is in a stable state with no events or responses pending. An event then arrives from the switch, maybe caused by a user having taken one of the terminals off hook. The event is passed to all of the features in the system in parallel. The system has now moved into an unstable state as there are events and/or actions outstanding. All of the features respond to the event with the actions that they wish to perform. These actions are then broadcast back to the features and again they reply with further actions. This process continues until no more actions are sent back from the features and the system returns to a stable state or, alternatively, a loop is detected. In general, we refer to this process of sending out events to the features and capturing their responses as *farming* and in many ways it is analogous to Blom's[2] collection of *micro-steps* forming a *macro-step*. It also shares aspects in common with operating system recovery points as discussed by Rossi and Simone[23].

If in one of these micro-steps more than one feature responds to a stimulus a shared trigger interaction has occurred and requires resolution. If there is no pre-stored resolution available for this interaction the feature manager can try a number of different interaction resolutions to create a graph of all of the possible posterior system states. To achieve this the feature manager rolls all features back to the point just before the arrival of the interaction-causing event before attempting to trigger the responding features in any order (and also only some of those features). This will result in a graph showing all possible behaviours of the system.

Note that this approach is different to other feature manager based techniques in that *the interaction is detected based on the result of passing the triggering event to the features, rather than just on the fact that potentially contesting features are present in the system*. Manual (or in future automatic) study of this graph allows determination of the correct resolution for this interaction, which can then be selected, allowing the system to continue processing. More details about the design and operation of this system are available in [19].

3.3 Strengths and Weaknesses

This work has enabled the differentiation between, and automatic detection of STIs, SAIs and LIs at runtime. It does not require any modification to existing features and the processing overhead for using it is relatively small. Unfortunately, at the point at which this first phase of work was completed, only manual resolution techniques by means of table population were in operation and so further work was required to totally automate the process. Further, SAIs are simply detected by one feature triggering another, which can lead to both desired and undesired interactions. A qualification of SAIs provides a significant improvement.

Once a stable system state has been achieved the messages representing requests for changes to the state of the hardware are committed, the features themselves are committed to the hardware and the system relaxes to await the arrival of the next event.

3.4 Augmenting the On-line Approach with Off-line techniques: Two Hybrid Approaches

As discussed previously, the existing transactional approach is based upon the stimuli and responses of software components. These events are rather “low level”, as one would expect for an on-line approach dealing with live implementations. In communication networks parlance these events can take the form of *off-hook* or *on-hook*, *start-ringing* and the like. Hence the detection mechanism is said to be based at the *technical* level.

In contrast, many of the off-line approaches adopt a *user-centric* view. This is often also a *connection-orientated* view. For example, user A is connected to user B, who forwards the call to user C. A user-centric view is natural as it is the *user* of the network that is affected by feature interaction, and it is the user who judges the efficacy of the network. It also allows a natural high level of abstraction that aids the application of formal methods. Abstractions including network details are prone to complexity issues. Hence for off-line techniques it can be appealing to work at a user-centric level.

We have developed two hybrid approaches, one is technical and the other user-centric. While both approaches originated in the off-line domain, the level of abstraction is very different. That is, the more technical approach (based on [7, 21]) is low-level and requires additional technical-level modelling, whereas the user-centric approach (based on [15]) is more abstract. While there is an overlap between the approaches, essentially the technical hybrid approach is applied to the problems associated with the resolution of Shared Trigger Interactions. The user-centric approach helps primarily with the qualification of Sequential Action Interactions.

As both approaches operate at run-time, they consider only features which are currently active in a call. Other features, which may be implemented, but are not activated or are not subscribed to by the parties involved in a call, are ignored. This helps to cut down the processing overhead incurred by the approaches. Additionally, the two approaches are able to detect interactions between features which have been subscribed to by different parties and consequently may be triggered at different ends of a call.

Furthermore, both approaches treat features as black boxes and assume no detailed knowledge about internal behaviour. This has a positive impact on the applicability of both approaches. Firstly, they can be applied in a live network, not just a captive run-time environment. Secondly, the approaches can be used in a competitive business environment where no detailed technical feature information will be available. And thirdly, the approaches are not limited to a particular network architecture. While in this paper the applicability of the approaches is demonstrated in an IN-like environment, it is believed that the approaches work regardless of whether the features are deployed on an Intelligent Network, on Parlay, or even in a Voice over IP environment using communication protocols such as SIP (Session Initiation Protocol) in combination with e.g. RTP (Real Time Protocol).

The next two sections consider each of the two approaches in detail. For comparison of detection and resolution capabilities, and efficacy with respect to both STIs and SAIs, we choose a single experimental set of features.

4 The User-centric Hybrid Approach

In this approach the selected off-line approach was originally developed as a filtering technique [15], although its detection of problematic scenarios is quite accurate. It has capabilities for detecting interactions which involve features subscribed to by different users; making the approach suitable for the detection of SAIs as these often involve features of multiple users.

The off-line approach is user-centric. In other words, it should detect that certain combinations of features change the behaviour as perceived by a user. This high level view was chosen in order to allow for simple algorithms with good run-time performance. Abstraction to the user level is acceptable, because empirical evidence suggests that all low level problems also have a manifestation at a higher, user perceived level.

The next section provides an overview of the approach and how the necessary information about features is provided. Section 4.2 discusses the interaction analysis in more detail.

4.1 Overview

The approach assumes call control features to be extensions of the Basic Call Service. Feature interactions are problems between different extensions to the Basic Call Service.

Treatments are an important aspect of this approach. Treatments are announcements or tones triggered by the network to handle certain conditions during a call, for example when a call is screened or blocked. Potentially, there may be multiple treatments involved in a call. For example, consider two features invoked during one call. One feature might connect a party to a busy treatment, whereas the other feature connects the (same or the) other party to a network unavailable treatment.

The behaviour of a feature is described in two parts: the triggering party and a connection type. The latter consists of two parts: the original connection to be set up before the feature is activated and the connection set up after the feature has been triggered.

An example should illustrate this. Call Forwarding Unconditional (CFU), which redirects all incoming calls to a predefined third user, can be described as shown in Fig. 4. Assume party A is the originator, B the terminator, and C the party where the call is redirected to. The behaviour has two parts, separated by a semicolon. In the first part, the notation TP.: X indicates that X is the triggering party, in this case it is B because CFU is triggered at the terminating end of a call. In the second part, notation (X, Y) → (U, V) indicates the connection type. (X, Y) is called the original connection and (U, V) is the connection after activating the feature. For each pair (A, B) we refer to A as the source and B as the destination.

TP.: B; (A, B) → (A, C)

Figure 4: Description of Call Forwarding Unconditional

The call starts with A attempting to connect to B. However, because of CFU, A is connected to C instead. So the connection type is (A, B) → (A, C).

4.2 Interaction Analysis

Interaction cases are found by analysing pairs of features. Two feature descriptions are compared according to six rules. If a feature pair fulfills any of the six rules, then the pair is said to interact. We consider each of the rules in turn. Note that while the analysis is pairwise, features are still described in isolation.

Rule 4.1. *Single User - Dual Feature Control*

If both features have the same triggering party and either the original connections or the resulting connections are identical, the pair interacts. Note that, even if the features aim at setting up the same connection, the features may clash as they may be triggered simultaneously. An example is given in Example 4.1. The shaded portions indicate the identical parts of the two behaviour descriptions.

Example 4.1.

CW:	*TP.: B; (A, B) → (A, B)*
CFU:	*TP.: B; (A, B) → (A, C)*

Rule 4.2. *Connection Looping*

The original connection of one feature is identical to the connection after activating the second feature, and the original connection of the second feature is identical to the connection set up after triggering the first feature. As both features are trying to divert the call, a loop occurs (ref. Example 4.2). Again, the shaded portions indicate the identical connections.

Example 4.2.

CFB:	*TP.: B; (A, B) → (A, C)*
CFU:	*TP.: C; (A, C) → (A, B)*

Rule 4.3. *Diversion and Treatment*

Two features interact if one feature establishes a call (*not* to a treatment) and the resulting connection is the original one of a feature, which redirects the call to a treatment. The triggering parties of the features must be distinct. This rule captures the fact that the connection which one feature is trying to establish is prevented by the other (ref. Example 4.3).

Example 4.3.

CFB:	*TP.: C; (A,C) → (A, B)*
OCS:	*TP.: A; (A, B) → (A, Treat)*

Rule 4.4. *Reversing and Treatment*

This rule is very similar to the previous one, but differs as follows: One feature reverses the call and the resulting connection is equal to the original connection of the other feature which connects to a treatment. In contrast to the previous rule, the same users need to be involved in both original connections, although their roles (source, destination) may be swapped. Example 4.4 illustrates this case.

Example 4.4.

AR:	*TP.: B; (A, B) → (B, A)*
OCS:	*TP.: B; (B, A) → (B, Treat)*

Rule 4.5. *Diversion and Reversing*

One feature forwards a call which is subsequently reversed to the originator by the other feature. More precisely, one feature forwards the call (not to a treatment) and the resulting connection is the original one of the other feature which reverses the call. A common manifestation of the problem is the originator of the call is called by a party that was never contacted (see Example 4.5).

Example 4.5.

CFB:	*TP.: C; (A, C) → (A, B)*
AR:	*TP.: B; (A, B) → (B, A)*

Rule 4.6. *Reversing and Diversion*

This rule is closely related to rule 4.5, but the forwarding happens after reversing the call. Consequently, one feature reverses the call and the resulting connection is the original one of the other feature, which then forwards the call to a third party. Example 4.6 illustrates this rule. So, the "new" connection of the first feature is prevented by the second feature.

Example 4.6.

AR:	*TP.: B; (A, B) → (B, A)*
CFB:	*TP.: A; (B, A) → (B, C)*

4.3 Integration into the On-line Environment

As discussed in section 3 the call control software, i.e. the basic call and all features, are encapsulated by cocoons. Signals destined for a feature are therefore passed through the cocoon to the feature proper, whereas control signals destined for the cocoon, are not. These are processed inside the cocoon.

The underlying on-line detection approach detects sequential action interactions because the feature manager monitors the replies sent by the features to an initial event. If two or more features (other than the Basic Call) reply to an event, or to feature responses which have been fed back to features, then an interaction is detected. However, as described in Section 3 this is a very crude detection mechanism. To augment the approach, the feature descriptions, i.e. the connection type and the triggering party, are added to the cocoon. The descriptions are queried using special control messages. The features themselves are unaffected. Recall that the feature manager detects interactions by corresponding with the features but does not commit feature responses to the hardware before the feature interaction check has been carried out.

The addition of the user-centric hybrid approach is a qualification of SAIs. To qualify a detected SAI, i.e. to ascertain if this is a desirable or undesirable interaction, the feature manager prompts both features for their behaviour descriptions. On reception the analysis discussed in Section 4.2 is applied. The feature manager will log the encountered scenario and terminate the call.

For new features, the cocoon is provided on feature deployment. For legacy software the cocoon is provided at the time of introduction of the on-line approach. Hence it is possible to provide the feature descriptions on a *per feature basis*, at feature deployment, assuming the feature provider makes available such high-level information.

5 The Technical Hybrid Approach

We turn our attention to the second hybrid approach, based on a technical level off-line technique. This hybrid approach augments the on-line approach presented in section 3 with the ability to automatically select good (if not best) resolutions.

The advantage of this approach is again an independence from the detailed implementation of a feature, but rather than considering a high level user's view, the feature providers' understanding of contradictory messages is taken into account. This requires a common semantics of messages across different developers and network operators or at least a common classification of messages according to treatment type (e.g. forwarding feature). Such information would be expected, in any event, for minimal interworking.

5.1 Overview

The user centric approach assumes that features extend a basic call model. We make the same assumption here, but do not distinguish between interactions between features and between features and the basic call. The on-line feature manager allows to explore interacting features in different orders and combinations. Here we make precise the concepts of solutions and resolutions as basis for the technical hybrid approach.

For a given set of features, a *solution* is a trace of one or more of those features running concurrently. That is, it is an interleaving of messages generated by a subset of the features. The *solution space* is the set of all traces (i.e. all solutions), for all subsets of a given set of features. It should be noted that the solution space might contain many traces that lead to a violation of required properties (i.e. there might be traces that represent incorrect behaviour). A *resolution* is a trace in the solution space that does not violate any specified properties.

The solution space depends on the granularity of the interleaving. A coarse grained interleaving allows features to be run in any order but does not allow messages of features to be interleaved, (within one macro-step). In this case the solution tree grows exponentially with the number of interacting features. A fine grained interleaving allows individual messages to be interleaved, thus resulting in a solution space growing exponentially in both the number of features and the length of their responses. Note that only features that are actually triggered contribute to the solution space. A fine grained interleaving has been explored in [21, Chapter 5]. Here we assume a coarse grained interleaving, as supported by DESK.

5.2 Resolutions

The main contribution of this hybrid approach is the ability to *resolve* interactions. After the feature manager has constructed the solution space as described in section 3, the space is passed through a pruning and extraction process: that is all branches that lead to undesired behaviour are removed and only the one with the most preferable behaviour is retained.

The operations of pruning and extraction are based on rules determined (off-line) by domain experts. The purpose of the rules is to discriminate between bad and good solutions and also to describe the quality of resolutions.

The offline analysis by domain experts is conducted on the solution space gained by running the feature manager without the extraction and pruning processes. The result of the analysis is a set of rules which are used to prune the branch of the solution space: our ultimate goal is one single branch. The aim of these rules is to describe undesired behaviour in a domain specific, but scenario independent way. An example of such a rule is that two consecutive tones are undesirable. Rules which depend on domain knowledge and the underlying architecture are classified as *message dependent*. We have also identified *message independent* pruning rules. These are more general, as they do not refer to domain specific information. The rules acquired in the offline phase are supplied to the feature manager in order to automatically resolve detected interactions by pruning the solution space. The new solution space can be analysed again to obtain additional resolution rules, and the process iterates, as required.

5.3 Message Independent Rules – Extraction

Message independent rules are used for *extraction* of the best resolution. They are characterised by not requiring any information about the semantics of the messages. Thus, they can be applied to a solution space without any knowledge of the messages occurring therein, assuming that we are able to differentiate between messages from different features (i.e. we know which messages originated from the same feature).

Rule 5.1. *Duplicate subtrees sharing the same parent can be removed.*

The solution space may contain several duplicate branches. Consider an example: assume 3 features (f_1, f_2, f_3) two of which (f_1 and f_2) respond to the same trigger. A branch corresponding to this trigger with all three features active is identical to one with feature f_3 disabled. Furthermore any branch with two features active, one of which is f_3, is identical to the branch with just the other feature active.

Obviously a duplicate branch will not provide a new solution, so all duplicates can safely be removed. This rule is not required explicitly in the DESK environment, as the feature manager implemented there excludes features that do not respond from alternative orderings.

Rule 5.2. *Traces containing messages from the largest possible number of features are preferable (to those containing messages from a small no of features).*

A feature is *satisfied* by a trace when its intended behaviour is exhibited. So, if a feature f_1 responds with messages a_1, a_2 and a_3 in that order, every trace in the tree containing these messages in *that order* satisfies feature f_1. The construction of the solution space maintains the relative order of messages as intended by a feature. However, the messages can be interleaved arbitrarily with responses from other features. If feature f_2 responds with b_1 and

b_2 then some possible traces satisfying both f_1 and f_2 are $a_1a_2a_3b_1b_2$ and $a_1b_1b_2a_2a_3$, but $a_1b_2b_1a_3a_2$ is not acceptable.

While the fine grained interleaving presented in this section is the more general case and not applicable in DESK, the rule is still relevant: the main motivation for the approach is to satisfy as many features as possible.

Rule 5.3. *Traces satisfying features with the highest priority are preferable.*

A simple, but effective method of extraction is prioritising features. We do not possess information about features' identities per se, though we do know their relative position in the network. For example, we know that a message has been received from feature f_i, but we do not know the identity of feature f_i (i.e. whether f_i is *call waiting*, *three way calling* or any other feature). We refer to i as the feature's *connection number*.

A simple precedence scheme allows features with a low connection number to have higher priority. Clearly this scheme could be extended to a system of priorities in which each feature has an associated weight (features with the highest weight are preferable). Each trace has a weight relative to the weights of the features satisfied by that trace. The trace with the highest weight would be preferred. One could define any number of other priority schemes. This provides a mechanism for user preferences.

Rule 5.4. *If there are a number of "best" resolutions, choose one.*

A "best" resolution is not necessarily unique. Suppose that after applying all rules a tree with more than one trace remains. At this point we have found more than one resolution that we would classify as the best, but obviously the system can only commit to one trace. However, if all traces represent behaviour that from a qualitative point of view is indistinguishable, we can simply choose one.

Note that it would be preferable from the user's point of view if this is a deterministic choice. The user is not (and shall not be required to be) aware of the resolution process, so the presented behaviour should be consistent across a number of separate calls.

5.4 Message Dependent Rules – Pruning

Message dependent rules can be seen to be more powerful, but they also require more information. In order for message dependent rules to be useful, a semantic understanding of messages is required. This allows one to develop relations on messages, such as classes of treatments, or billing messages. This understanding enables us to build grammars describing good or bad behaviour. For example "a treatment following an onhook message is not useful unless there was an offhook in between". Message dependent rules are used for *pruning* the solution space.

While it might seem unreasonable to require a semantics of messages, especially in a setting where the internal behaviour of the features is unknown, a requirement for some knowledge about messages is practical because the message set in telephone switching systems is somewhat restricted and such understanding is required for interworking even in the absence of feature interaction.

Messages can be grouped into classes. These include the rather obvious classes such as "billing messages", "user messages" and "system messages". In addition, and more interesting, we can have classes like "announcements", "treatments", "tones", "hookevents". Classes of messages can be overlapping, for example "announcements" is a subclass of "treatments" and "tones" intersects with "treatments" (*ringtone* is a tone, but not a treatment, whereas *busytone* is both a tone and a treatment and *announce* is a treatment but not a tone).

We wish to express rules that describe sequences of messages and also the absence of a message. Often we wish to refer to whole classes in a simple way. Regular expressions are used to express these rules. Empirical evidence shows that regular expressions are sufficient to express any rule required. (We have found no evidence, yet, of the need for context free grammars.) This is not surprising, as messages generally involve call setup, manipulation and tear down. These do not usually require the user to count the occurrence of events, e.g. for an incoming call the third ring denotes a different meaning, but note, such behaviour is possible in the future.

In our case study based on DESK we find that only the following five message dependent rules (5.5.x) are required: (1) connecting a user to two different resources, (2) routing a call to two different locations, (3) routing a call away from a user and still changing that user's state, (4) routing a call away from a user and connecting the user to a resource, and (5) changing a user's state and connecting the same user to a resource. The actions described in these rules map directly onto the message set used by DESK: routing is performed by a *routing* message, a state change is caused by *move to state* and *connect to resource* connects the user to a resource.

A set of resolution rules is considered complete if all interactions can be resolved. However, there is no generic method of identifying a complete rule set (due to the diverse nature of features). This is a drawback, as an incomplete rule set can lead to a trace being identified as a resolution when the trace is not a resolution.

5.5 Integration into the On-Line Environment

The feature manager in the online approach described in section 3 produces a solution space which is passed to a (human) operator for resolution. Here, we pass the solution space to our automated pruning and extraction process.

The pruning and extraction process is essentially the application of the rules defined previously. Pruning is a function that operates on a solution space and removes all undesired traces by checking each trace against the message dependent rules. Thus pruning returns a solution space containing no traces with undesired properties. Extraction operates on a pruned solution space and removes traces according to rules 5.1 to 5.4. The main goal of the extraction process is to identify and extract the most desirable resolution from the possible desirable resolutions. In the run-time environment the two functions are simply added instead of the manual resolution process.

A major drawback of this approach is the complexity: a large number of branches must be constructed that will be discarded. We therefore have also implemented an "on-the-fly" approach. On-the-fly resolution works by trying to apply pruning to the current solution under construction. As soon as a behaviour is identified as undesired, the construction of the current solution is aborted. The branch constructed since the last choice is then removed and construction of another solution is attempted. This will greatly reduce the complexity if many features are active: bad solutions can be identified early in the construction. Extractions, **cannot** be performed "on-the-fly" as there is not enough information about the solution space available before the construction is completed. Obviously, the on-the-fly mechanism requires

	CFU	CW	CFB	OCS	TCS	VMS	AR	ACB	DND	HL
CFU		5.x	5.x, 4.2	4.3	5.x, 4.3	5.x, 4.3	5.x, 4.5, 4.6	5.x	5.x, 4.3	
CW			5.x		5.x	5.x	5.x	5.x	5.x	
CFB				4.3	5.x, 4.3	5.x, 4.3	5.x, 4.5, 4.6	5.x	5.x, 4.3	
OCS							4.4			4.1
TCS							5.x, 4.4	5.x	5.x	4.3
VMS							5.x, 4.4	5.x	5.x	4.3
AR								5.x	5.x, 4.4	5.x
ACB									5.x	
DND										4.3
HL										

Figure 5: Results of the case study.

a deeper embedding into the feature manager, that is the detection process and resolution process are entwined rather than staged, so changes to either become more complicated.

6 Experimentation and Evaluation

The two hybrid approaches were applied to the same case study, for evaluation and comparison. In this section we give an overview of the case study and results.

For the case study ten common features [9, 16] were selected covering a wide spectrum of functionality. The selected features are: Call Forwarding Unconditional, Call Waiting, Call Forwarding on Busy, Originating Call Screening, Terminating Call Screening, Voice Mail System, Automatic Ringback (returns the call to the caller), Automated Callback (allows a caller to automatically recall a busy callee), Do Not Disturb and Hotline.

The case study was performed using a prototype based on the DESK switching system in conjunction with the two off-line techniques presented in Sections 4 and 5.

6.1 Results and Discussion

Between the ten selected features 49 interaction scenarios have been found. Note that a number of feature pairs exhibit interactions in several different scenarios, so the number of pairs of features involved is somewhat smaller. Table 5 provides details of the detected interactions.

Entries are of the form $A.B$. A indicates which approach has been used to detect the interaction and refers to the section number in this paper. Hence *4* refers to the detection of a Sequential Action Interaction using the user-centric approach presented in Section 4, and *5* refers to a Shared Trigger Interaction using the technical approach discussed in Section 5. B refers to a particular rule discussed in the paper. As the technical approach requires the application of multiple rules to resolve an interaction, the notation *5.x* has been chosen.

Of the 49 detected interactions, 28 are Shared Trigger Interactions. All of these could be *resolved* at run-time and the call progressed. With the remaining interactions, of Sequential Action type, the problem was detected and the call terminated.

As can be seen, there is only one case where an interaction was detected using rule 4.1. This is due to the fact that rule 4.1 detects interactions which are triggered by the same party. Very often these are in fact STIs. However, in this project, the technical hybrid approach was used instead to deal with STIs; this approach is preferable, as resolution allows for the call to continue. As a consequence, rule 4.1 is not heavily employed.

A more detailed study of the results of the case study shows that no scenarios which have not been highlighted by any approach contain feature interactions. In other words, no *false negatives* have been found in the case study. This point is fundamental for an interaction approach if its results are to be trusted and hence be useful. On the other hand, all scenarios which are highlighted, actually exhibit problematic behaviour. However, sometimes this is subjective to individual user expectations.

An example is the interaction between Call Forwarding and Voice Mail where a forwarded call is redirected to a voice mail treatment. Some users might see the Call Forwarding feature as a feature to increase their reachability, for instance when they are at a different location. While the voice mail feature does not restrict or affect the behaviour of the Call Forwarding directly, some subscribers to CFB might not expect a voice mail to answer their calls. Also the caller might be surprised to get a voice mail announcement from a party they did not try to reach. There are a few interactions detected which depend on user expectations and user preferences. This point is discussed further in Section 7.

For STIs, the best possible resolutions (based on our understanding of the features) have been found for all detected interactions. The patterns describing undesired behaviour as used by the pruning algorithm have been relatively obvious and only very few such patterns were required. Our understanding of the semantics of the existing messages made the formulation of the rules possible. The quality of the resolutions depends on the knowledge of the message semantics. However, even a general understanding of the messages is sufficient. For any system under consideration, it must be assumed that this general understanding exists, as otherwise enhancement is questionable even in the absence of the feature interaction problem.

Looping interactions are a special case of SAIs, and thus are handled by the user centric approach. Missed trigger interactions cannot be detected by the online technique. However could they be detected, both the user-centric and the technical approach can handle these.

7 Summary and Further Work

In this paper, two separate, complimentary offline techniques have been integrated with a run-time approach to form two hybrid approaches.

Both approaches improve the relatively weak detection (because it lacks any qualification) of SAIs and more importantly resolution capabililties of the run-time approach. The run-time approach detects interactions in two ways: either more than one feature responds to an event (Shared Trigger Interaction) or a feature perceives an event only as a consequence of the behaviour of another feature (Sequential Action Interaction). The two hybrid approaches are complimentary, as broadly each addresses one of the two interaction classes.

The first hybrid approach, the user-centric approach, refines the SAI detection mechanism through an analysis of high-level descriptions of the two features involved. The analysis, based on a set of rules, checks the potential consequences of the two “triggered” features and then classifies them as desirable or undesirable. The latter interactions are then excluded.

The technical approach, refines the STI resolution mechanism through the addition of rules reducing the number of potential solutions and then extract a best resolution. The rules which exclude potential solutions are based upon an agreed semantics of messages, whereas the rules to extract a best resolution are independent of message content.

The two hybrid approaches have been implemented and evaluated in a case study involving 10 features. 49 interactions were detected. Of those, 28 were STIs which were detected and resolved at run-time and the call was allowed to progress. The remaining interactions, all SAI, were detected and the call terminated.

For the first time, Shared Trigger Interactions can be automatically detected *and* resolved at run-time. Previously, only an automatic detection had been possible. Further we have implemented a novel on-line automatic detection and qualification of Sequential Action Interactions; previously, this was a manual task. As a consequence, the difference and the precise meaning of these two types of interactions have been clarified.

With the automatic detection and qualification of sequential action interactions some scenarios have been found which could not be categorised into desirable or undesirable interactions. These cases are dependent on the preferences of particular users. Solving these cases is beyond the work reported in this paper. With the increasing acceptance and hopefully deployment of new emerging architectures, such as SIP new opportunities to solve these cases arise. For example, the expression of caller preferences and the application of feature interaction approaches based on negotiation may be of great value. One such attempt, using policies to express user preferences can be found in [22]. Future work should study these technologies and exploit them in new interaction approaches.

A further valuable area of research is the application of feature interaction approaches into the wider and more general domain of component based systems [1].

Acknowledgements

The two hybrid approaches have been developed as a part of a joint EPSRC-funded project between the Universities of Glasgow and Stirling (EPSRC GR/M03429/01 and GR/M03573/02).

The original work by Dave on classes of interaction was the result of work carried out with the aid and funding of the Royal Commission for the Exhibition of 1851. We would also like to acknowledge that the technical hybrid approach was developed during Stephan's previous period of employment, at the University of Glasgow.

References

[1] L. Blair, T. Jones, and S. Reiff-Marganiec. A Feature Manager Approach to the Analysis of Component-Interactions. In B. Jacobs and A. Rensink, editors, *Formal Methods for Open Object-Based Distributed Systems V*. Kluwer Academic Publishers, 2002.

[2] J. Blom. Formalisation of requirements with emphasis on feature interaction detection. In *[11]*, pages 61–77, June 1997.

[3] M. Cain. Managing run-time interactions between call processing features. In *IEEE Communications Magazine*, pages 44–50, February 1992.

[4] M. Calder, M. Kolberg, E. H. Magill, and S. Reiff-Marganiec. Feature interaction: A critical review and considered forecast. *Computer Networks*, 41(1):115–141, Jan 2003.

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

[6] 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, June 1999.

[7] M. Calder, E. Magill, S. Reiff-Marganiec, and V. Thayananthan. Theory and practice of enhancing a legacy software system. In Peter Henderson, editor, *Systems Engineering Business Process Change 2*. Springer Verlag, London, 2001.

[8] 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.

[9] E. J. Cameron, N. Griffeth, Y.-J. Lin, M. E. Nilson, W. Shnure, and H. Velthuijsen. Towards a Feature Interaction Benchmark for IN and Beyond. *IEEE Communications Magazine*, 31(3):64–69, March 1993.

[10] R. H. Campbell and B. Randell. Error recovery in asynchronous systems. *IEEE Trans. Software Engineering*, SE-12(8):811–826, 1986.

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

[12] S. Homayoon and H. Singh. Methods of addressing the interactions of intelligent network services with embedded switch services. *IEEE Communications Magazine*, page 42ff, December 1988.

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

[14] M. Kolberg and E. H. Magill. Service and feature interactions in TINA. In *[13]*, pages 78–84, 1998.

[15] M. Kolberg and E. H. Magill. A pragmatic approach to service interaction filtering between call control services. *Computer Networks: International Journal of Computer and Telecommunications Networking*, 38(5):591–602, 2002.

[16] M. Kolberg, E. H. Magill, D. Marples, and S. Reiff. Second feature interaction contest. In *[5]*, pages 293–310, May 2000.

[17] M. Kolberg, R. O. Sinnott, and E. H. Magill. Engineering of interworking tina-based telecommunications services. *Proceedings of IEEE Telecommunications Information Networking Architecture Conference*, April 1999. IEEE Press.

[18] E. Kuisch, R. Janmaat, H. Mulder, and I. Keesmaat. A practical approach towards service interaction. *IEEE Communications*, pages 24–31, aug 1993.

[19] D. Marples. *Detection and Resolution in of Feature Interactions in Telecommunications Systems at Runtime*. PhD Thesis, Communications Division, Department of Electrical and Electronic Engineering, University of Strathclyde, 2000.

[20] B. Randell. System structure for software fault tolerance. *IEEE Trans. Software Engineering*, SE-1(2):220–232, 1995.

[21] S. Reiff-Marganiec. *Runtime Resolution of Feature Interactions in Evolving Telecommunications Systems*. PhD Thesis, University of Glasgow, Glasgow (UK), 2002.

[22] S. Reiff-Marganiec and K. J. Turner. Use of logic to describe enhanced communications services. In *Formal Techniques for Networked and Distributed Systems – FORTE 2002, LNCS 2529*, pages 130–145, November 2002.

[23] G. P. Rossi and C. Simone. A multitasking operating system with explicit treatment of recovery points. *Microprocessing and Microprogramming 14*, pages 55–66, 1984.

[24] L. Schessel. Administrable feature interaction concept. *ISS'92*, 2:122–126, oct 1992.

[25] K. G. Shin. Evaluation of error recovery blocks used for cooperating processes. *IEEE Trans. Software Engineering*, SE-10(6):692–700, 1994.

[26] Y. Wakahara, M. Fujioka, H. Kikuta, H. Yagi, and S. Sakai. A method for detecting service interactions. *IEEE Communications*, pages 32–37, aug 1993.

[27] K-L. Wu. Rapid transaction-undo recovery using twin-page storage management. *IEEE Trans. Software Engineering*, 19(2):155–164, 1993.