Feature Requirements Models: Understanding Interactions

J. Paul Gibson, *CRIN, Nancy, France*[1], gibson@loria.fr

Abstract

This paper states that many of the problems which arise when features combine (i.e. *feature interactions*) are due to *badly* developed requirements models for individual features. With sufficiently *good* requirements models, in which each feature is formally modelled and validated against customer understanding, the feature interaction problem is much more tractable. We analyse some of the *standard* feature interactions and show that, in most cases, the notion of interaction is used to signify that the requirements are not fully understood, properly recorded or rigorously validated. In contrast, we then show how *good* requirements models could resolve the problems that arise when combining features. An interaction is said to occur if and only if requirements are contradictory. The problem which this paper addresses is how to avoid, detect and resolve such contradictions during requirements development.

1 Introduction

The feature interaction problem is stated simply, and informally, as follows: A *feature interaction* is a situation in which system behaviour (specified as some set of features[2]) does not as a whole satisfy each of its component features individually. We concentrate on the domain of telephone features [4, 2]. Figure 1 illustrates this definition within the formal framework which we adopt throughout this paper. Two types of formal models are used. Firstly, we require an executable model (written in LOTOS [12] with an object based style [10]) which is useful for validating the dynamic behaviour. Secondly, we have a logical model (using a mixture of TLA [14] and PVS [15]) which defines a set of properties that can be validated statically. Then, we have a formal verification that the executable model fulfils the requirements of the static model.

The argument put forward in this paper is as follows. Feature interaction is a difficult problem (see section 1.2) made more difficult by the fact that feature combinations cannot be fully understood when individual features are themselves not understood. Formal requirements modeling [13] is the triangular process of gaining understanding, recording understanding in a formal model, and validating the model. By formally developing feature requirements models we can approach the problem of combining features with much more confidence. The problem is still as difficult, but at least it is now well defined. We argue (in section 1.1) that improving understanding is the key to interaction avoidance, detection and resolution.

1.1 Requirements Modeling: An Overview

Requirements capture is the first step in the process of meeting customer needs. Building and analysing a model of customer needs, with the intention of passing the result of such a process to system designers, is the least well understood aspect of software engineering. The process is required to fulfil two very different needs: The customer must be convinced that requirements are completely understood and recorded, and the designer must be able to use the requirements to produce a structure around which an implementation can be developed and tested. However, the process is made

[1]Supported by contract 96-1B CNET-France-Telecom & CRIN-CNRS URA262.

[2]In this paper we make no distinction between a feature and a service.

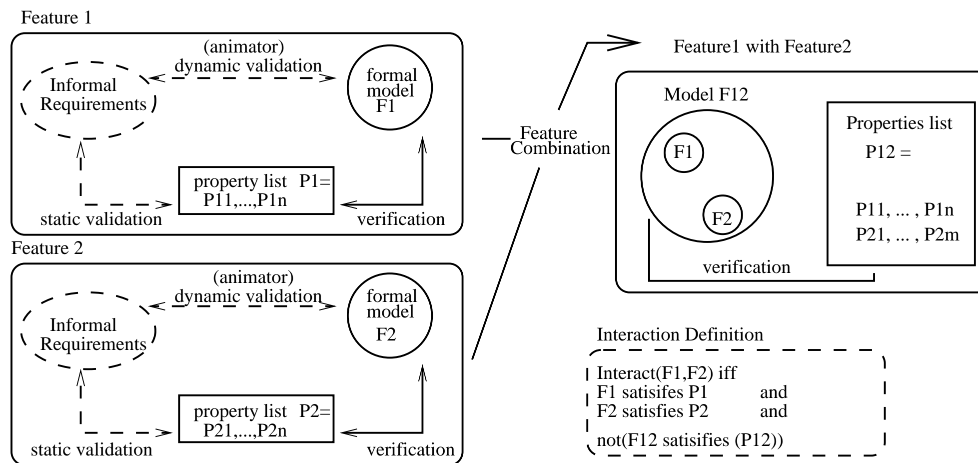

Figure 1: Feature Interaction: A formalisation

easier by the fact that many of the same principles of structure, organisation and method are common to both the problem domains and solution domains. The fundamental principle of requirements capture is the improvement of mutual understanding between customer and analyst and the recording of such an understanding in a structured model.

1.2 Feature Interaction: What's different?

Features are observable behaviour and are therefore a requirements specification problem [17]. Most feature interaction problems can be (and should be) resolved at the requirements capture stage of development. If there are no problems in the requirements specification then problems during the design and implementation will arise only through errors in the refinement process. Certainly the feature interaction problem is more prone to the introduction of such errors because of the highly concurrent and distributed nature of the underlying implementation domain, but this is for consideration *after* each individual feature's requirements have been modelled and validated; otherwise it will not be easy to identify the source of the interaction. Features are requirements modules *and* the units of incrementation as systems evolve. A telecom system is a set of features. A feature interaction occurs in a system whose complete behaviour does not satisfy the separate specifications of all its features. Having features as the incremental units of development is the source of our complexity:

- Complexity explosion: Potential feature interactions increase exponentially with the number of features in the system.
- Chaotic Information Structure In Sequential Development Strategies: The arbitrary sequential ordering of feature development is what drives the internal structure of the resulting system. As each new feature is added the feature must include details of how it is to interact with all the features already in the system. Consequently, to understand the behaviour of one feature, it is necessary to examine the specification of all the features in the system. All conceptual integrity is lost since the distribution of knowledge is chaotic. The arbitrary ordering of feature development has a large effect on the internal system structure.
- Assumption Problem: Already developed features often rely on assumptions which

are no longer true when later features are conceived. Consequently, features may rely on contradictory assumptions.

- Independent Development: Traditional approaches require a new feature developer to consider how the feature operates with all others already on the system. Consequently, we cannot concurrently develop new features: since how the new features work together will not be considered by either of the two independent feature developers. We want to have an incremental approach in which the developers do not need to know anything about the other features in the system. In our approach, it is the system designers who must resolve the integration problems: integration methods arise from an analysis of the features to be integrated. Formal requirements of individual features are required for the integration process to be verified.
- Interface Problem: User controls on traditional phones are very limited and hence the input signals become polymorphic. This is a major problem in requirements specifications as it can lead to the introduction of ambiguities in systems of features. Formal requirements models make explicit the mapping between abstract and concrete actions and our systems can be automatically verified to ensure an absence of ambiguity that could lead to interactions.
- Invalid Plain Old Telephone Service (POTS) Assumptions: Phone systems have changed dramatically over the past ten years. Many people (including feature developers) are not aware of the underlying complexity in the concrete system and, as a way of simplifying the problem, often make incorrect assumptions based on their knowledge of the plain old telephone service.

The feature interaction problem is difficult: having formal requirements models makes it manageable.

1.3 Formal Methods

A formal model of requirements is unambiguous — there is only one correct way to interpret the behaviour being defined. Although the model must still be mapped onto the real world (i.e. validated by the customer), this mapping is in essence more rigorous than in informal approaches. Building a formal model requires a better understanding of the problem domain and a better understanding of how the problem domain is viewed by the customer. A formal model can explicitly model nondeterminism, when choice of behaviour is specified. Another advantage of a mathematical approach is that high levels of expressibility allow the definition of *what* rather than *how*.

In our formal approach, interactions occur only when requirements of multiple features are *contradictory*. The complexity of understanding the problem is thus contained within a definition of *contradiction* in our semantic framework. We shall show that in most of the feature interaction examples found in published texts, there is no formal definition of feature interaction. In fact, most of the interactions which we studied (see section 3) correspond to incomplete and informal requirements models. In other words, if the features were modelled *better* then we would be able to better understand what is and what isn't an interaction.

2 Understanding Features: formalising requirements

As an aid to comprehension we limit our study to interactions which deal only with the

caller's point of view. We also restrict our level of abstraction: problems due to sharing of resources, information hiding, interface realisation, etc ... are design issues and are not examined here. The key to understanding features is the language(s) we employ for communication, verification and validation. Three different, though complementary, views of features play a role in the process of requirements capture. Before we examine specific interaction cases, we summarise the different semantic views and use them to model the POTS requirements of a single phone.

2.1 States and Actions: A dynamic (object) view

Labelled state transition systems are often used to provide executable models during the analysis and requirements stages of software development [7, 8]. In particular, such models play a role in many of the object oriented analysis and design methods [1, 6]. However, a major problem with state models is that it can be difficult to provide a good decomposition of large, complex systems when the underlying state and state transitions are not fully understood. The object oriented[3] paradigm provides a natural solution to this problem. By equating the notion of class with the state transition system model, and allowing the state of one class to be defined as a composition of states of other classes, we provide a means of specifying state transition models in a constructive fashion. Further, such an approach provides a more constructive means of testing actual behaviour with required behaviour. This is explained in more detail in [11] where LOTOS is used to provide a state based view of objects as processes.

The equations of a class specify how all its member objects respond to all valid service requests. In fact, they correspond to labelled state transitions. The equations map a state before the transition to a state after transition (and may be labelled by a value to be returned to the service requester). An object's state can be structured or unstructured. An unstructured state is represented by a literal value. A structured state is represented by a set of component objects. Composition is primarily a relationship between objects. It can, however, be extended to classes: when all class *members* are represented by the same *structure* then the class can, without ambiguity, be said to be *composed* from the classes which parameterise the *structure*.

This state based object view forms the basis on which we build our feature animations and permit behaviour validation in a compositional manner. However, they are not *good* for formal reasoning about feature requirements. For this we need to consider specification of state invariants and fairness properties.

2.2 Defining Invariants: A static view

Invariants are needed to specify abstract properties of a system which must always be true. The object oriented approach permits the definition of three sorts of invariant:

- Typing: By stating that all objects are defined to be members of some class we are in fact specifying an invariant. This invariant is verified automatically by the O-LSTS tools.

- Service requests: Typing also permits us to state that objects in our system will only ever be asked to perform services that are part of their interfaces. This invariant is also verified automatically by the O-LSTS tools.

[3] In fact, in this paper we do not consider such issues such as subclassing, inheritance and polymorphism; thus the view is really object based.

- State Component Dependencies: In a structured class we may wish to specify some property that depends on the state of two or more of the components, and which is always true. This cannot be statically verified using the O-LSTS semantics but can be treated through a dynamic analysis (model check). Unfortunately, such a model check cannot be guaranteed when we have large (possibly infinite) numbers of states in our systems. For this reason we need to adopt a more proof theoretic framework (such as the B tool).

By translating our state invariant requirements into another proof theoretic framework[4] we have been able to statically verify our state component invariants. These invariants are often the key to feature interactions: when features are components and their invariants are not guaranteed by the containing system then we have an interaction.

2.3 Temporal Logics: A fairness view

TLA is a temporal logic introduced by Lamport [14] and based on the action-as-relation principle. A system is considered as a set of actions, namely a logical disjunction of predicates relating values of variables before the activation of an action and values of variables after the activation of an action; a system is modeled as a set of traces over a set of states. The specifier may decide to ignore traces that do not satisfy a scheduling policy as strong or weak fairness and temporal operators as $\Box$ (Always) or $\Diamond$ (Eventually) are combined to express these assumptions over the set of traces. Such fairness is important in feature specification and cannot be easily expressed using our state based semantics. The key is the need for nondeterminism in our requirements models.

Without a temporal logic, nondeterminism in the features can be specified only at one level of abstraction: namley that of an internal choice of events. This can lead to many problems in development. For example, consider the specification of a shared database. This database must handle multiple, parallel requests from clients. The order in which these requests are processed is required to be nondeterministic. This is easily specified in our OO semantic framework. However, the requirements are now refined to state that every request must be eventually served (this is a fairness requirement which we cannot express in our semantic framework). The only way this can be done is to *over-specify* the requirement by defining how this fairness is to be achieved (for example, by explicitly queueing the requests). This is bad because we are enforcing implementation decisions at the requirements level. With TLA we can express fairness requirements without having to say how these requirements are to be met.

2.4 POTS: A mixed model view

To illustrate the mixed-model approach, and to aid understanding of the features in our case studies, we examine the specification[5] of the behaviour of a single phone, within POTS, from the user's point of view. The dynamic behaviour is modelled by the state transition system in figure 2.

A state invariant which defines a relation between the hook component and the signal component will include the following requirement: `(signal = talking) => (hook = off)`. In the simple finite telphone system, this is directly verifiable by checking that

[4] In fact, we have successfully performed the translation to B and PVS.

[5] This specification is much simpler than the real case, but we abstract away from a number of details in order to simplify our reasoning.

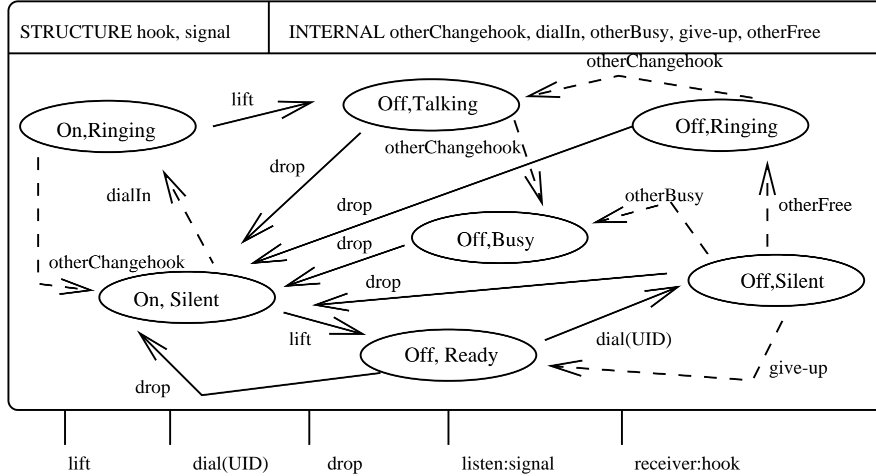

Figure 2: A single POTS phone user model

it is true in all the states. In the network of many different telephones we use the invariant to specify relations between pairs of phones. For example, a simple POTS state invariant requirement is that `only an even number of phones can be talking at the same time`. This is proved by showing that it is true in the initial state (where all phones are off and ready). Then we show that all possible state transitions maintain the required property (if it is true before the action occurs then it is true after the action occurs). This property cannot be checked through an exhaustive search of a system of an unbounded number of phones. It can be checked by proving that all transitions are closed with respect to the invariant.

Nondeterminism in the telephone is specified through internal events. These correspond to actions which are not available to the phone user but which perform state transitions. The simple phone example illustrates the need for fairness (where we specify that something good eventually happens rather than that something bad never happens). Consider a telephone which has just dialled a number and is off hook and silent. The user does not wish to remain in this state indefinitely but can only drop the phone if they wish to instigate a state change. By specifying fairness on the `give-up` action we guarantee that this state transition must eventually happen if the phone stays in the off-silent state (for whatever reason).

3 Case Studies: Feature Pairs

We examine only pairs of features that are generally accepted as *interactions*. Of the six pairs which we chose (more for their repetitive apperances in the available literature than for any other reason) we show that, within our formal semantic framework, there is only one interactions which, if suitable analysis methods are used, can be detected and resolved automatically. We find this to support our argument that formal requirements models are beneficial to the feature development process.

3.1 Call ID and Call ID Restriction

Call ID (CID) offers a means of identifying the caller whilst Call ID Restriction (CIDR) stops outgoing calls from being identified. This is clearly a case of contradictory

requirements ...or is it? We believe that with a *good* requirments model then there is no interaction in this case. Firstly, we must *formalise* the requirements of the two services in order to show that there is no reason for there to be a contradiction. Such formalisation led us to a more thorough understanding of the requirements.

CID Requirements

Consider what the user actually wants from CID. If I subscribe to CID and my phone is ringing then I can see the *identification* of the caller. The type of the *identification* is crucial to the problem. For example, if the identification is a telephone number (TN), then we must consider the assumptions made about what we can do with such data. For example, we can dial a TN and expect to reach a phone, we can add a TN to a black list for incoming calls, and we can call back (if busy) the TN of a caller who couldn't get through. This is the real reason why types are important: they define what we can do with certain things in the system. Problems arise (feature interactions) if we are not careful about the type of the *identification* returned by the CID feature. Above all, we should not make the assumption that the type of the *identification* is a TN.

CIDR Requirements

Consider what the user actually wants from CIDR. If I subscribe to CIDR then I don't want the person(s) I am calling to be able to contact me through the use of CID. In other words, I don't want my *identification* to give them enough (any) information about who I am or where I am. CIDR should thus give me control over what information is carried by my *identification* when I make a call. By default, the *identification* can be set to some *well accepted* value like the TN from which the call originated, or a personalised TN for the caller, or *restricted* for CIDR, or *international* for overseas calls which do not offer CID, or ....

CID and CIDR: No contradiction

A problem was said to arise when the caller has activated the CIDR feature whilst the callee has activated CID. It was expected that the callee can access the TN of the caller and so use it in many other different features like Automatic Call Back (ACB), for example. This led to the feature developers havng to change the original feature specifications to cope with the exception case(s). Problems arose because they made assumptions about what could be done with the value returned by the CID feature; then these assumptions were invalidated by the CIDR feature. This is a classic example of bad requirements modelling: there are hidden assumptions which should be made explicit.

In our formal model we make no such assumptions because we create a polymorphic type system for the *identification* data. A class hierarchy states explicitly the assumptions we make about the data being *shared* and so we avoid an interaction (and can formally prove that no such interaction occurs).

Lessons learned

It is clear that problems can arise in this case because of lack of understanding of the CID requirements. CID is concerned with gaining some information about the incoming call which may be used to identify the caller. We can mistakenly assume that the ID will be a TN but unless such an assumption is made explicit then we do not truly understand the requirements being specified. In this case, if we explicitly state that an ID is a TN then we do have an interaction. Such an assumption can be made explicit by specifying the type of the ID. However, through detailed analysis of the CID requirements, we have avoided a potential interaction by not making this invalid

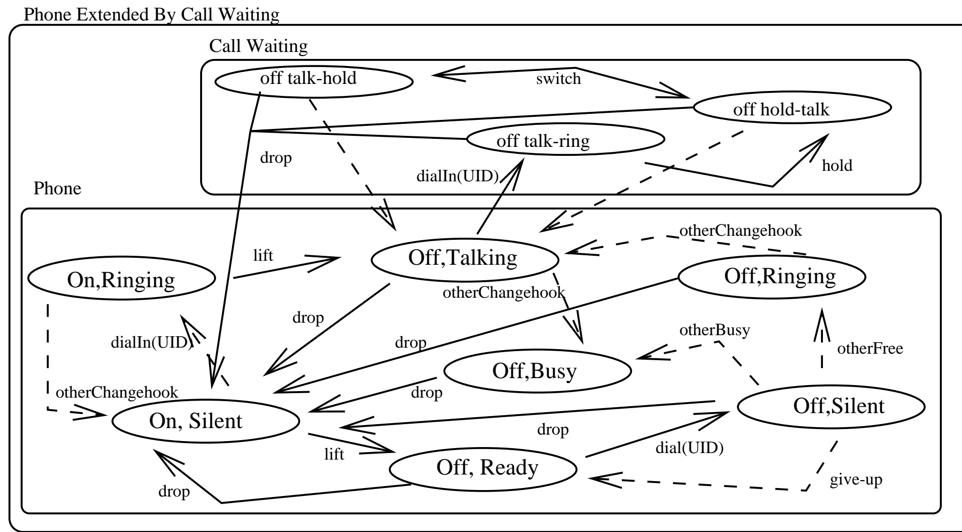

Figure 3: Call Waiting state transition model

assumption.

3.2 Call Waiting and Three Way Calling

This, again, is commonly used to illustrate feature interaction. It is a classic case which frequently appears in the literature. We believe that, with good requirements models, there is no interaction at all. The interaction arises only when invalid assumptions are made about the triggering of the Call Waiting (CW) and Three Way Calling (TWC) features.

CW Requirements

When I activate CW its functionality is enabled only when I am talking to someone else. If I am talking to someone and a third person tries to call me then I can chose to hold them. Now I am talking to one person and holding another. I now have the ability to switch the talking and held persons. I can hang up on both calls at once and become on and silent in the normal POTS state. Similarly, either of the two other parties can hang up and I return to the POTS state off and talking. The state-transition based requirements model is shown in figure 3.

TWC Requirements

If I subscribe to TWC then the feature is enabled only when I am talking to someone and I receive a call from a third party. I then have a new service (connect) which permits me to talk with both callers at the same time. In this new state I can also disconnect either one of the two callers. This is illustrated in the partial state transition diagram in figure 4.

CW and TWC: No interaction now

Traditionally, the argument for these two features interacting is as follows: What happens if both features are activated and we are talking to someone when we recieve an incoming call? There is an ambiguity because the system doesn't know which feature to execute. The problem is then further complicated by the fact that a flash-hook is used as the concrete event for three different abstract events (namely, hold, switch and connect). In this case the mapping between abstract and concrete events is not well defined since the ambiguity cannot be resolved by context alone.

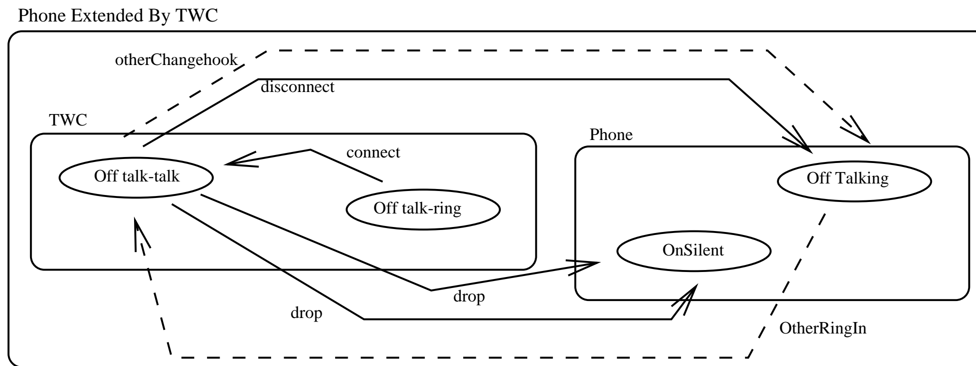

Figure 4: Three Way Calling (partial) state transition model

In our development strategy, neither of these problems exists and thus we do not consider this an interaction case. Quite simply, we cannot identify a contradiction in the requirements. CW allows the caller to be held *if the called chooses to do so.* TWC allows the caller to be connected to a 3-way talk *if the called chooses to do so.* There is an active choice made and the caller cannot be guaranteed to be held or connected. Thus, in the combination of both these features the possibility of either feature executing does not change, and the choice is still made by the person being called. Certainly, in the documented specifications where an interaction is claimed one of the problems seems to be that feature execution is not a choice but an obligation. Consequently, interaction will occur. This, again, is a problem of *bad* requirements models and lack of understanding.

To complete our analysis of this feature combination, we comment on the notion of mapping abstract to concrete actions. The need for this arises from the limited interface functionality of standard telephones. At the telephone we often have to assign different meanings to the same action depending on context (like the flash-hook signal in this example). This requires a mapping from possibly overloaded concrete actions to a set of abstract mappings. Problems occur if this mapping cannot resolve the ambiguity of overloading using internal knowledge of the system state. For example, in the state where I am talking and someone is ringing we should not have flash hook as the concrete action for both connect and hold. The *correctness* of the mapping can be verified statically to guarantee that this doesn't happen. It is really a design detail which should not be of interest in the requirements development stage.

Lessons learned

By avoiding passive entry into feature execution we avoid the obligation of feature execution. In this way we avoid concurrent execute of two features. Furthermore, we must take care when mapping from abstract to concrete actions during design. This mapping can only be made rigorous (and proven to be *correct*) if the individual features have formal requirements models.

Call Waiting and Three Way Calling: An extension

In our formal specification of CW and TWC we have not considered how to go from one feature to the other (without first having to go through a basic POTS state). For example, we may require that when holding a call we could then connect it. This transfers us from CW control to TWC control. Similarly, we may consider allowing a TWC to hold someone, thus transferring control the other way. The temptation is to

try to do this directly as an interaction between CW and TWC. However, this approach does lead to contradictory requirements in each of the distinct features. Consequently, we realise that this extension needs to be developed as something new in its own right. In this way we can re-use the two features as components of the new feature (but the user does not use the two features individually). This is a much more controlled approach to feature development than an ad-hoc approach which attempts to use interactions to provide something more than just the sum total of their two parts (which is where many of the problems originate). Our object oriented approach clarifies this as the difference between inheritance and delegation.

3.3 Wake Up Call and Call Forward No Reply

This combination of features illustrates what happens if requirements modellers make implementation decisions too early in the development process. The Wake Up Call (WUC) can be specified as occurring as the result of an incoming call from some central clock. This leads to problems with Call Forward No Reply because it tries to forward WUC to another number.

WUC Requirements

With WUC I can request my phone to ring so that it wakes me up at a specified time. To stop it ringing, I lift the phone which leaves it in a wake up message state. After that I can only put the phone down. This cancels the alarm.

CFNR Requirements

With CFNR, when I have an incoming call which is not answered then it will eventually be forwarded to another number.

WUC and CFNR: interaction analysis

In the case where both WUC and CFNR are activated the traditional view is as follows: The two features interact because if I don't answer a WUC before a certain number of rings then the call will be forwarded to another number. Thus, I may not be woken up whilst someone who wishes to sleep will be disturbed.

In this case, problems occur because the developers are making the assumption that a WUC leads to the phone ringing because of an incoming call, i.e. the phone rings because the call comes in from somewhere else in the phone network. This assumption is obviously implementation based: if we have a centralised clock then this clock can be shared between all WUC users. Furthermore, it assumes that the call in the WUC (used to start the phone ringing) is the same type as the call used when an ordinary phone calls the user.

In our model, we make no such assumptions. The WUC action that triggers the ringing of our phone is not the same as that of a normal call: we don't even specify whether it comes from the network or from a local clock in the phone. Since it is not a *call* it cannot be forwarded. Thus, there is no interaction.

Lessons learned

This example shows that we shouldn't make implementation decisions too soon. To validate the WUC feature we need to provide an executable specification and the temptation is to re-use the incoming call mechanism. This is a concrete implementation decision which should not appear in a requirements model. We must have the requirements of WUC specified at a level of abstraction which does not oblige us to use the

incoming call mechanism to trigger a WUC.

3.4 Call Waiting and Answer Call

This example is identical to the CW-CFB problem. We mention it because it appears so often in feature interaction papers. The key, as before, is to make the *hold* and *leavemessage* actions actively decided by the users of the telephones. Thus the activation of these features gives the choice to do these things and doesn't oblige their execution.

Lessons learned

This example shows that there is potential for finding a classification of feature compositions. Re-use of requirements analysis is possible, and this re-use represents a useful abstract classification.

3.5 Originating Call Screening and Call Forward

This feature combination is interesting because it illustrates that precision in requirements really does help to avoid feature interactions. In particular, it shows the importance of the validation process in the development of formal models. We identify the need to really understand why Originating Call Screening (OCS) is required. If this reason is not made explicit in the requirements model then problems can arise with other features such as Call Forward (CF).

OCS Requirements

With OCS activated, I cannot make outgoing calls to a number on my screening list.

CF Requirements

With call forwarding, all my incoming calls are forwarded to another number.

The potential interaction

The traditional view is that there is an interaction because I may be able to forward a call to a number on my screening list. Then, when an incoming call arrives we must ask if it is forwarded as CF seems to require or if it is screened as OCS seems to require?

Here, the reasoning behind the OCS feature needs to be examined. Does the user screen the calls because they don't want to pay for calls to the screened numbers (perhaps all international numbers are screened)? Does the user screen the calls because they don't want to allow access to the phones specified. When the requirement is cost based then we have to consider who pays for the redirection (forwarding) when specifying the two features together. For example, if forwarding is paid for by the caller then we don't mind forwarding to a screened phone. If forwarding is paid by the user then we do mind. Although cost is outside the scope of this paper, we can distinguish between the two cases:

- Cost counts: Specify OCS to disallow the dialling of a number (and thus disallow payment)
- Cost does not count: Specify OCS to disallow the ringing of a screened call.

The second specification does not contradict the CF requirements since all calls are forwarded. The first case does lead to a contradiction and is thus representative of a feature interaction. Only through precise communication with the customer of the feature can we decide what the requirements of OCS really are. In this case we need two different OCS features to reflect the different needs (even though, in isolation, the one OCS feature fulfils both needs).

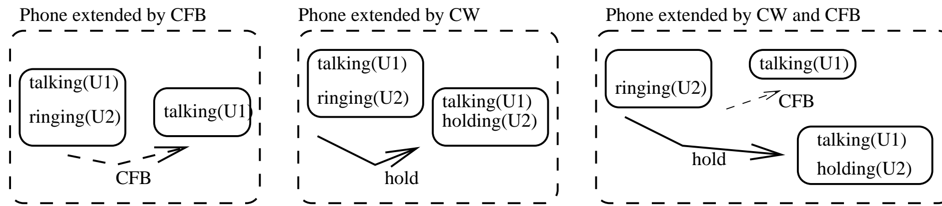

Figure 5: Call Waiting and Call forward on busy

Lessons Learned

The simple lesson here is that we need to understand why a feature is being developed. If there are different reasons then we need to justify using one model (feature) for sets of different requirements. To address the question of why the customer wants a feature requires a good level of communication, together with a rigorous validation process.

3.6 Call Waiting and Call Forward on Busy

Call Waiting (CW) and Call Forward on Busy (CFB) provide us with the first instance of contradictory requirements which can be automatically detected because of the breaking of temporal assumptions in each of the two features.

CW Requirements

The requirements of CW are as explained previously.

CFB Requirements

If I activate CFB then if I am talking to someone and another call comes to my phone it is automatically redirected (forwarded) to another number of my choice.

CW and CFB: An Interaction (At Last)

CW requires that I will *always* be given the choice of holding an incoming call if I am talking to someone when it arrives. However, the CFB feature may be executed *before* this choice can be made and so the CW requirement is contradicted in the system containing both features. Furthermore, a fairness condition in CFB states that *eventually* the incoming call will be forward. However, if CW is executed, the incoming call may be held by the user and never be forwarded. Thus we have another contradiction. This is illustrated in figure 5.

This example illustrates the need for both a state transition model of our requirements and a logical set of requirement that have to be verified against the model. Using just the state transition view, it is difficult to see the contradiction in the requirements. CFB states that *eventually* the phone forwarded to will be dialled. CW states that the choice of holding will *always* be available. The notions of *always* and *eventually* cannot be explicitly expressed in our state transition model. We need a meta-reasoning (based on TLA semantics) to formally detect the contradictory natures of CFB and CW.

Interaction Resolution: Arbitration

Now that the interaction has been identified, we must examine resolution techniques. Clearly, the problem is due to the nondeterministic action which forwards the incoming call to another phone. This action is not actively chosen by the user. If we made the user choose to forward the incoming call then there would be no interaction. However, this is against the spirit of the requirements of CFB: the user wants the forwarding to be done automatically without needing them to know that it is happened. Consequently, we refuse to change the action of forwarding to be active. Two (obvious) choices for

stopping the parallel execution of both features are as follows. Firstly, we could refuse to allow the activation of one service whilst the other is activated. On the other hand, we could ask the user to decide between services as soon as he attempts a double activation. Both these options are a form of prioritisation (or arbitration). The first case is controlled by the network system, the second is controlled by the user. Both these arbitrations are dynamic since the decisions are state dependent.

4 The Feature Case Studies: A Review

A number of principles where identified during our case studies.These are listed below.

4.1 Use Strong Typing

The advantages of typing in all forms of development are well known. Types are not needed in correct systems but they do help to create correct systems. A simple telephone example is that of the concept of a telephone number. Clearly, telephone numbers are polymorphic within a concrete subtyping hierarchy. Without types, it is difficult to handle the different natures of telephone numbers at the requirements stage.

4.2 Use Types as Behaviours or Roles

Here we impose the object oriented principle of classification. The class is used as the fundamental unit of behaviour (and as the means of typing these units). Thus, every feature is a class which plays a specific role. The class hierarchy provides a behavioural benchmark for categorising features. Already some work has been done towards creating a benchmark for feature interactions [3] and we believe an object oriented strategy would complement this research.

4.3 Use Invariants

Invariants are used to define relationships between components of a system that must be true during the lifetime of the system. They are a well understood, formal means of specifying requirements in a compositional manner. Every non trivial component (of a (sub)system), i.e. one with its own components, has an associated invariant and there is one invariant between all components (of a (sub)system). Invariants are the logical glue for putting together systems from component parts.

4.4 Avoid ambiguity in naming of actions

The practice of polymorphic actions arises from the minimilist interface provided by most telephones. For example, a *flash hook* action can signal different things to different features. If these features are requested at the same time then the meaning of a *flash hook* may be ambiguous. Whenever possible try to maintain a clear distinction between *abstract actions/signals* in the requirements model and *concrete actions/signals* in the implementation model. Finding a *correct* mapping between abstract and concrete action names is a design problem and not inherently a feature interaction problem; although, the limited domain of concrete actions may make this more difficult (or impossible) in the case of feature development.

4.5 Features should be explicit not implicit

Typically, features appear in formal specifications only as implicit, derivable properties of the total system. We can verify that a system complies to the requirments of a feature but there is no compositional means of removing the feature and examining it as a single identity. Given a logical landscape as our semantic basis would permit such a

compostional view since the properties required of a feature are exactly its specification. However, in such a logical approach, the actual development of features is much more complicated. We believe that we should have an *explicit* compositional approach in the same spirit as the logical method, whilst avoiding the synthesis and analysis problems that arise from features that are specified logically.

4.6 Arbitration is the key to interaction resolution

An interaction occurs only when feature requirements are contradictory. Arbitration is the means of automatically weakening the requirements of some features to remove the contradiction.

4.7 Choosing Modelling Language

The advantage of a logical approach is self evident: the combination of features is just logical conjunction and the absence of interaction is automatic provided the result is not false. The disadvantage of such an approach is the difficulty in constructing new features and analysing their dynamic behaviour. The principal problem is that of communicating with the clients in such a mathematical model. Contrastingly, more operational (state based) approaches are more powerful with respect to synthesis and analysis. However, it is then much more difficult to say what it means for two features to be contradictory (i.e. have requirements that cannot be met at the same time). The main source of feature interaction problems seems to be exceptions. Using invariants helps to transfer the analysis of exceptions away from the dynamic and towards a purely static approach. An object oriented approach gives us abstraction and generalisation within a compositional user friendly framework. A combination of a number of semantic frameworks seems to be the only option for all our modelling needs. The probelm of feature interaction is so general, with complex, diverse issues, that we cannot expect a single semantic model approach to be satisfactory.

4.8 Operational Requirements are necessary

Concurrent (independent) development of features requires separation of specification from implementation. However, for implementors (designers) to fully understand requirements we expect to be able to animate our specifications. This is also a necessary part of validation. Thus we require operational semantics [16] for our feature specifications (as well as our more abstract logical requirements for testing compatibility).

4.9 The incremental development problem: minimise impact of change

In traditional problem domains (and using state-of-the-art development methods) when new functionality is added to a system it is possible to do this by *connecting it* to only a small subset of the system components. (We will not for now attempt to define the different types of connection.) Additions that are localised (with fewer connections) are easier to make than those which are spread about the system. The addition of a feature is inherently a non-local problem (in the current telephone architectures) because it necessitates connection with most of the other components (the other features) in the system. Hence, each addition has global impact. We are searching for an architecture which supports local incrementation techniques.

4.10 Restrictive Assumption Approach

Restrict the assumptions that a feature developer can make about the behaviour of its environment (other features in the system included). Since new features will be added

later we cannot place any assumptions on them. However, in some cases assumptions must be made. These should be specified as invariant properties that are amenable to static analysis. Our goal is to simplify this analysis by formalising a minimum assumption set that does not restrict our functionality but does eliminate interactions.

5 Conclusions

We have shown that many interactions arise because requirements are badly developed. Furthermore, we argue that good feature requirements models depend on a formal mixed-model semantics for successful development. Contradictory requirements can and should be detected and resolved at the requirements stage of development. Formality is the most powerful tool in the area of feature interactions.

References

[1] G. Booch. *Object oriented design with applications.* Benjamin Cummings, 1991.

[2] L. G. Bouma and H. Velthuijsen, editors. *Feature Interactions In Telecommunications.* IOS Press, 1994.

[3] E. J. Cameron, N. D. Griffeth, Y. Lin, M. E. Nilson, and W. K. Schnure. A feature interaction benchmark for in and beyond. In *Feature Interactions In Telecommunications*, 1994.

[4] K. E. Cheng and T. Ohta, editors. *Feature Interactions In Telecommunications III.* IOS Press, 1995.

[5] P. Coad and E. Yourdon. *Object oriented analysis.* Prentice-Hall (Yourdon Press), 1990.

[6] P. Coad and E. Yourdon. *Object oriented design.* Prentice-Hall (Yourdon Press), 1990.

[7] L. Constantine. Beyond the madness of methods: System structure methods and converging design. In *Software Development 1989.* Miller-Freeman, 1989.

[8] Geoff Cutts. *Structured system analysis and design method.* Blackwell Scientific Publishers, 1991.

[9] T. DeMarco. *Structured analysis and system specification.* Prentice-Hall, 1979.

[10] J. Paul Gibson. Formal object based design in LOTOS. Tr-113, University of Stirling, Computing Science Department, Stirling, Scotland, 1994.

[11] J.Paul Gibson. *Formal Object Oriented Development of Software Systems Using LOTOS.* Tech. report csm-114, Stirling University, August 1993.

[12] R. Guillemot, M. Haj-Hussein, and L. Logroppo. Executing large LOTOS specifications. In *Proceedings of Prototyping, Specification, Testing and Verification VIII.* North-Holland, 1991.

[13] IEE. *Special Collection On Requirements Analysis.* IEE Transactions on Software Engineering, 1977.

[14] L. Lamport. A temporal logic of actions. Technical Report 57, DEC Palo Alto, april 1990.

[15] S. Owre, N. Shankar, and J. B. Rushby. *The PVS Specification Language.* Computer Science Laboratory, SRI International, CA, February 1993.

[16] Pamela Zave. The operational versus the conventional approach to software development. *Comm. ACM*, 27:104–118, 1984.

[17] Pamela Zave. Feature interactions and formal specifications in telecommunications. *IEEE Computer Magazine*, pages 18–23, August 1993.