# A taxonomy for triggered interactions using fair object semantics

Paul Gibson, *NUI Maynooth, Ireland*
Geoff Hamilton, *DCU, Dublin, Ireland*
Dominique Méry, *Université Henri Poincaré, Nancy, France*

**Abstract.** We formalise the notion of *triggered features* and show how the resulting model can be used to classify a subset of common interactions. Our underlying *fair object* formal framework, based on the integration of object-state machines and temporal logic, can be exploited to provide support for re-usable analysis. We test our theoretical results in the construction of object oriented feature requirements models, where each feature is triggered by the same `dialIn` event. Our results support the view that the development of a feature interaction algebra is not just a theoretical proposition but is also a practical engineering possibility.

## 1 Introduction

### *1.1 Overview of issues to be addressed*

This paper reports on the continuation of our work, introduced in [10], towards formulating a feature interaction algebra. In the conclusions of the previous paper, we identified the crux of the problem as being able to perform a re-usable analysis of feature interactions based on feature classes. The weaknesses to this introductory work were that the classification hierarchies were treated informally, and the analysis that arose was based on a small number of very different features.

In this paper, we aim to address these weaknesses. Firstly, we decided to focus attention on a subset of the many possible features. We agree with [5], where it is claimed that *the feature interaction problem is not one problem but rather a multitude of problems, some of which are understood, but others remain puzzling.* We chose to concentrate on triggered interactions, which have been identified in many different papers (e.g. [19, 2]). The goal was to better model an understanding of this small, yet very important, subset of features — this forms the main body of this paper. Secondly, having previously argued for the formal integration of logical and operational requirements[12], we wanted to apply the resulting *fair object*[10] semantics in the domain of telephone features. Thirdly, we wanted to test our integrated tool support[15, 11] on the new requirements models that we built. Finally, we wanted to address two important issues of *failure* within the feature interaction community as a whole: the *use* of formal methods and the *missed-use* of modularity[4, 21].

### *1.2 Background and related work*

In [5], it was stated that for progress to be made in the feature interaction domain then there are a number of important research areas which need to be targetted. This greatly motivated our research because we are addressing many of these issues concurrently: the *application of formal methods* is fundamental to our approach[9]; the *classification towards a taxonomy* corresponds to our re-usable analysis based on abstract classes[10]; the *experimentation with different methods, architectures, platforms, etc ...* corresponds to the way in which we have developed our mixed semantic approach [12]; and the *consistent and coherent tool support throughout development* is provided by our incremental approach to integration of animators for validation and provers for verification[11].

The work presented in this paper is complementary to the *plug-and-play* approach in [20] which is based on treating features as *first class citizens*. We support this view, but are concerned with the claim that the central point to features is to add functionality to the system which was not considered when the system was designed. We believe that this *paints a very bad picture*. In our approach, as with all object oriented methods, the goal is to build a set of abstract classes which provide a stable foundation upon which changes can be incorporated. In this way, we may not be able to predict the exact functionality being specified by a new feature's requirements but we should be able to predict how it may be incorporated into the old system.

The use of foreground/background models in [16] inspired us to re-evaluate our specification of an interaction as a simple breaking of invariant properties. As we, too, had experience of such a model leading to many spurious interactions, we tried to find a new way of specifying properties which were *like invariants but not quite as strong*. This required the notion of eventuality and the use of temporal logics. The notion of stability which arose out of our work has just started to help us resolve some of the central issues in the feature interactions seen in this paper. This is work in progress but we return to it (in section 7) as we believe it motivates continuing with our *fair objects*-based method.

### *1.3 Paper structure*

The paper is structured as follows. In section 2, we introduce the *fair object* semantic model upon which our formal reasoning is based. In section 3, we illustrate how telephone feature requirements can be modelled as *fair objects*. In section 4, we introduce the notion of *triggered features* through the simple example of an answering machine. In section 5, a taxonomy for triggered types is proposed. In section 6, we examine the utility of our taxonomy through the analysis of a set of features which are all triggered by the `dialIn` event. In section 7, we discuss re-usable detection and resolution techniques for triggered feature interactions. Section 8 concludes with a review of how the work presented in the paper addresses some of the issues raised with respect to the use of formal methods and the lack of modularity.

## 2 Fair Objects

### *2.1 The need for fairness in requirements models*

An operational approach to requirements specification offers many advantages: the customer can use the requirements model as a type of prototype which they can execute (animate). This improves the communication between the customer and analyst, and aids in synthesis, analysis and validation. However, we have found the inability to express *liveness* requirements a main weakness when applying an operational approach:

we can specify only what cannot happen (i.e. safety properties) rather than what must happen. In many cases, it is an explicit requirement of the customer that something *good* must happen rather than something *bad* never happening. The specification and verification of such *liveness* requirements is not possible in the classical operational (state transition system) models.

Liveness requirements often manifest themselves as making some sort of *fair choice* between internal state transitions. In other words, in nondeterministic operational models the *eventuality* of something good happening depends on the nondeterminism being resolved *fairly*. We choose to adopt temporal logic as a suitable means for expressing and reasoning about such fairness properties.

## *2.2 Fairness — a liveness property in TLA*

TLA [18] provides a simple and effective means of expressing fairness properties. The semantics incorporate the notions of *always* (represented by the box operator $\Box$) and *eventually* (represented by the diamond operator $\Diamond$). Using these, we can specify different categories of fairness for guaranteeing the eventuality requirements:

- **Weak fairness** states that if an action is continually enabled then it will be eventually carried out.
- **Strong fairness** guarantees the eventual execution of an action when that action is enabled an infinite number of times (though not necessarily continuously).
- **Possible Fairness** deals with the case when it is always possible for an action to be enabled (by following a certain sequence of internal actions) yet the action cannot be guaranteed to be executed through the use of strong fairness.

The ability to model nondeterminism at different levels of abstraction is the key to TLA's utility in requirements modelling. It provides a means of specifying and verifying eventuality requirements as different types of fairness property. Unfortunately, TLA does not provide the means for easily constructing and validating initial customer requirements. By combining TLA and operational (object-oriented) semantics we can alleviate these problems.

## *2.3 Fair objects: fair servers, progression and politeness*

In [14], we proposed the concept of *fair objects*, which combine temporal semantics and object-oriented concepts in a complementary fashion. The underlying semantic model is that of an object as a labelled state transition system (O-LSTS), as first modelled[8] using LOTOS. This work identified three temporal modalities which provide useful high-level specification mechanisms for requirements modelling: *fair servers*, *progression* and *politeness.*

A *fair object* is one which cannot introduce deadlock (or livelock) into a system through continually refusing to execute a method request. More specifically, a *fair object* has the property that all its external services are always eventually enabled.

We have also formulated the notion of *progression*, the need for which arises from the way in which concurrent processes are modelled through an interleaving of events. We can view such interleaving as if there is a scheduler which randomly chooses which process to be executed at any particular time. In such systems we specify a *progression* requirement to guarantee that that each of the component processes is *fairly scheduled.*

Finally, although safety properties, specified as invariants on the state components, are preserved under composition in our object-oriented approach through encapsulation, liveness properties rely upon the cooperation of the environment. In [13], we examined the effect of object composition on the liveness requirements of the objects being composed. When following an object-oriented approach to specifying the behaviour of concurrent systems, we would like to be able to specify the liveness properties of each object. However, an object must be viewed as an open system which relies on its environment to ensure that its liveness properties are satisfied. When these objects are composed, the resulting composition must be checked to ensure that the liveness properties specified for each object are preserved. If this is the case, we say that these objects are *polite* [3, 10].

### *2.4 The need for client-server eventuality classes*

Different forms of *politeness* arise when maintaining the *eventuality* properties in the new composed system depends on some form of fairness in the other component(s) involved in the construction. Clients may, in order for the system to function correctly, require that the services they request are carried out *eventually.* We have, during the development of telephone services based on the fair object concepts, identified five different types of *client eventuality requirements* which provide high level reusable concepts:

- **Immediately Obliged** — A client may require that a service request be serviced immediately.
- **Eventually Obliged** — A client may require that a service is carried out eventually.
- **Immediately Conditional** — A client may wish a service to be performed immediately but if it cannot be done without delay then it must be informed so that it can attempt to do something else.
- **Eventually Conditional** — The service is required eventually but if it cannot be guaranteed in a finite period of time then the client must be informed so that something else can be done.
- **Unconditional** — The client wants the service but places no eventuality requirements on when (or if) the service is performed.

(The compositional specification and verification of these *eventuality classes* is on-going research which is beyond the scope of this paper.)

### *2.5 The novelty of our approach*

One solution [13] to ensuring that the liveness properties of objects are preserved under composition is to allow only a weak form of object composition in which the liveness properties of the objects are guaranteed to be preserved. This is the approach which is taken in [1] and [6], where only parallel composition of objects (with no communication) is allowed. The liveness properties of the composed object in this case are easy to represent in TLA as the logical conjunction of the liveness properties of each object.

In general, however, we require stronger forms of object composition in which there is communication (through synchronisation) between the composed objects. This is the approach which is taken in [17] and [7]. In this case, separate proofs are required to show that the liveness properties of the composed objects are preserved under composition.

The approach which we take is to define only *local* liveness requirements on each object. Each object can perform a number of actions, which may be either internal or external. The local liveness requirements for the object specify the liveness of the external actions of the object over all its possible action sequences independent of the environment. We then show how additional liveness constraints can be placed on the internal actions of an object to ensure that it meets its local liveness requirements.

### *2.6 Using liveness to classify fair object interactions*

We define the following different classes of politeness between two composed (interacting) objects:

- **Independent** - if they do not synchronise on any actions
- **Perfect friends** - if they do synchronise on actions, but the internal liveness constraints do not need to be strengthened to meet the local liveness requirements of the composed object
- **Friends** - if they synchronise on actions, but the internal liveness constraints of actions in one or both objects need to be changed to meet the local liveness requirements of the composed object
- **Politicians** - if the local liveness requirements of the composed object cannot be met by changing the internal liveness constraints in either object unless some additional resolution mechanism is used
- **Enemies** - if the local liveness requirements of the composed object cannot be met by changing the internal liveness constraints in either object or by using any additional resolution mechanism

This terminology corresponds exactly to the informal definitions of feature interaction types given in our introduction to a feature interaction algebra[10].

## 3 Fair Objects for modelling telephone feature requirements

In this section we introduce our method for modelling feature requirements using our fair object semantics. Of course, we start with the specification of a simple POTS telephone and build from there.

### *3.1 A POTS telephone as a fair object*

Consider the state transition diagram for the `Phone` in figure 1. There are three types of requirement to be specified:
**1. The operational requirements**
There are eight states in the system and the initial state, identified by an incoming arrow which is not rooted from another state, is `on-silent`. There are three external state transitions (`lift`, `drop` and `dial`), and six internal (nondeterministic, from the

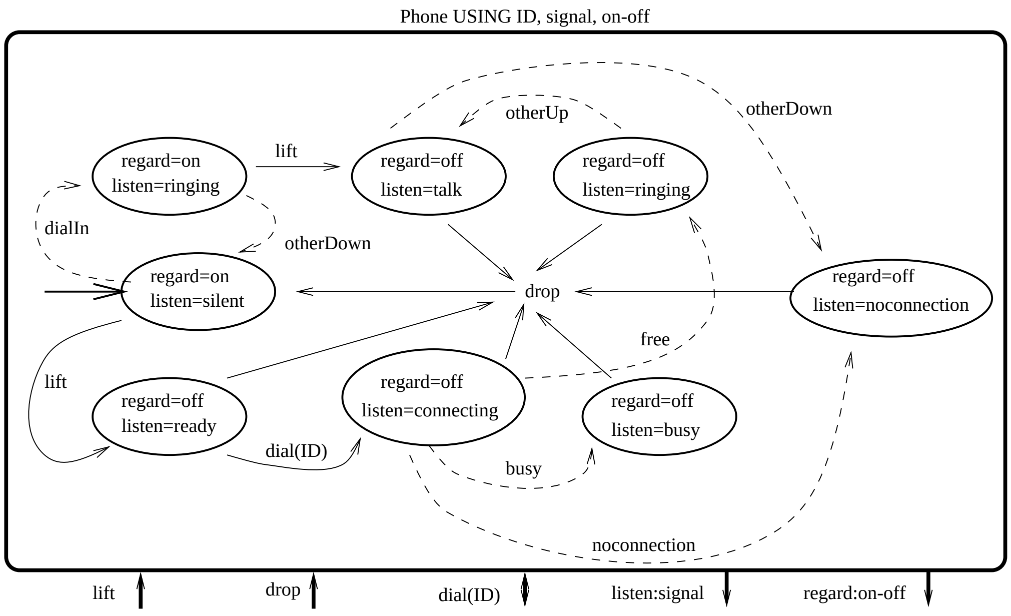

Figure 1: A phone (POTS) Fair Object

point-of-view of the phone user) transitions (`dialIn`, `otherDown`, `otherUp`, `free`, `busy` and `noconnection`). Two accessor operations (`signal` and `regard`) are used to provide information (to the phone user) about the current state of the phone system; these do not result in state changes to the system. Without fairness requirements, this state transition model is considered as a *standard* class; and an object of such a class is specified as being in one of the eight possible states. The methods of the class (object) correspond exactly to the external state transitions and accessor operations.

**2. Safety requirements**

Safety requirements are specified as state invariant properties. The telephone is specified as consisting of two object components: `regard` and `listen`. A state invariant which defines a relation between these components will include the following requirement:

> *Always*, if I am `talk`ing with someone then the telephone must be `off` hook

This can be formalised[1] in TLA as:

```
□ ((listen = talk) => (regard = off))
```

In the simple finite telephone system, this is directly verifiable by checking that it is true in all the states. Since there are only eight states, this is trivial to check.

**3. Fairness requirements**

Even the simple telephone can benefit from the specification of an *eventuality* requirement. In the phone specification it is possible to be in the state `off-connecting` forever. In this state we have just dialled a number and we must wait for the system to tell us if the number is `free`, `busy` or that `noconnection` is possible. A reasonable eventuality requirement is:

> I will *eventually* leave the state `off-connecting` even if I do not force the state transition myself.

[1]We do not give the exact TLA specification, but use a simpler syntax to capture the idea.

Using TLA, this requirement is guaranteed by weak fairness on the `noconnection` event:

$WF(noconnection)$.

Thus, if the network *takes too long* to decide if the requested line is `free` or `busy`, then we will eventually get a `noconnection` signal.

## *3.2 A POTS System*

**The operational object oriented semantics**
The diagram in figure 2 shows how we compose our 3-Phones system using synchronisation between our fair object components. In this case, we show a system with only three telephones which are connected, through synchronisation of their internal actions, to a central `exchange`. (It is beyond the scope of this paper to detail the exchange component's specification. We note only that all three telephones are instances of our `Phone` class and that the exchange component is also specified using the same fair object semantics.)

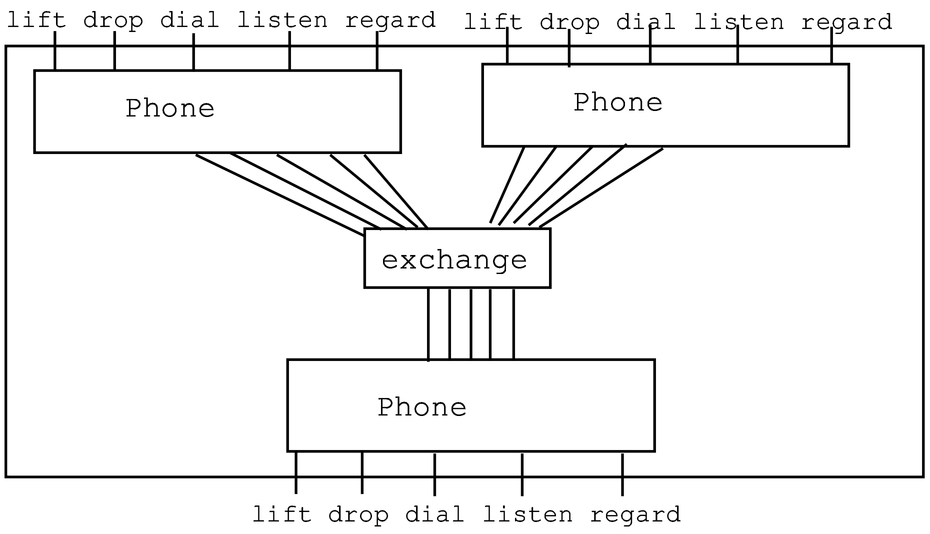
lift drop dial listen regard
lift drop dial listen regard
Phone
Phone
exchange
Phone
lift drop dial listen regard

Figure 2: A 3-phone POTS system

**A safety property**
In the network of many different telephones we can use invariants to specify safety requirements between sets of `Phones`. For example, a simple POTS state invariant requirement is that $\Box$ (the number of phones `talking` is even) 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.
**A liveness property**
Within the telephone network, the POTS `exchange` component controls the synchronisation between pairs of `Phone` connections. When one `Phone` hangs up, for example,

the POTS `exchange` must inform the other `Phone` user by changing the state of their phone. The updating of shared state information must always eventually be carried out. For example:

> When I `dial` the number of a `Phone` which is `free` then it will *eventually* `ring`

These type of *eventuality* requirements can be guaranteed using the notion of *progression.*

## 4 An answer machine as a triggered fair object

Without giving the actual specification of the *service+feature* composition, it should be clear (from the diagram in figure 3) how an answering machine object (`answermachine`) synchronises with a `Phone` object. The standard functionality is for the `Phone` to ring for a finite period of time and then the `answermachine` takes control so that a message can be taken. When the `answermachine` is active, it will return to the `inactive` state when a `lift` or `otherDown` action is forced through synchronisation with the `Phone`.

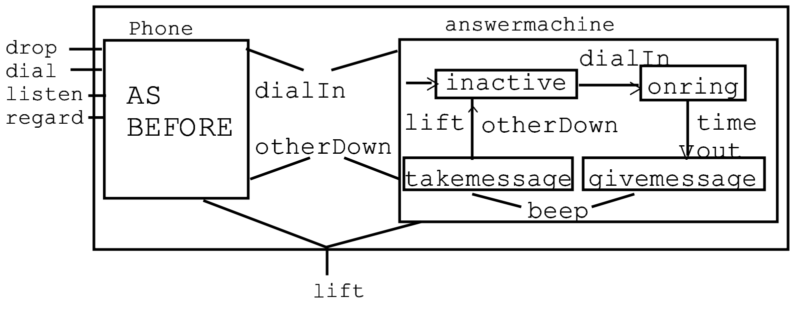

Figure 3: POTS with answering machine

The answer machine can be said to refine the `onringing` `Phone` state into three substates: `onring`, `givemessage` and `takemessage`. These substates are linked by the new (internal) transitions `timeout` and `beep`. When the `Phone` is in the `onringing` state, the `answermachine` must be in one of these three substates. Further, when the `Phone` is not in the `onringing` state then the `answermachine` must be `inactive`. As the `onringing` state can occur only after a `dialIn` event, we say that the `answermachine` is triggered by the `dialIn`.

A reasonable *liveness* requirement is that when I ring someone with an answering machine I will eventually `talk` with them or get to leave a message. This requirement is quite naturally specified by making the answering machine a fair object, which guarantees the eventual execution of a `timeout` and `beep` provided no external actions return control to the `Phone`.

### *4.1 Stability issues*

When we enter the answering machine states, in the new composed system, we no longer preserve the invariant of the system-wide POTS — the number of people `talking` is no longer even. Previously, our analysis (and many others like ours) would register this as an interaction. We propose a solution where some invariant properties are weakened so that they are no longer required to be true in every state of the composed system, but are required to be eventually true from any state of the system. We call these weakened invariants *stability properties*. In section 7, we comment on how stability may offer insight into the resolution of some feature interactions. So, for example, in the answering machine example it is clear that the invariant that an even number of people must be talking at any one time is certainly broken. However, if it is specified as a stability requirement then we can guarantee that the property will always eventually be true, provided that the answer machine is guaranteed to return control to the `Phone` (which it does).

## 5 Triggered features: some analysis

The answer machine seems to capture the essence of a typical triggered feature. However, it is not clear, without further analysis, how the concept of triggering can be formalised. In this section we consider triggering as a general concept, outside of any particular modelling domain.

### *5.1 What is triggering?*

Consider the diagram in figure 4. This represents a triggering which is easily understood because it corresponds, almost exactly, to the standard remote procedure call model which is found in most programming languages. (In the diagram, as with all similar diagrams which follow, we do not name uninteresting states and state transitions which play no role in the interaction between the service and the feature.)

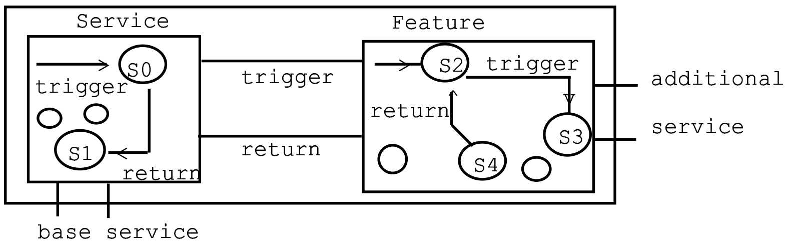

Figure 4: The base trigger class

In this *service+feature* state transition model, we are interested only in fve states and two transitions:

- State S0 is the state of the service in which the feature has just been **trigger**ed (and is henceforth known as an **trigger state**).
- State S1 is the state of the service to which the feature **return**s control to the service (and is henceforth known as a **return state**).
- State S2 is the initial (inactive) state of the feature.

- State S3 is the state in which the feature is initially active after triggering (and is henceforth known as an **entry state**).
- State S4 is the final active state in which the feature can return control to the service (and is henceforth known as an **exit state** of the feature.)

The semantic rules which define this *base trigger class* are as follows:

- There may only be one of each of the states S0 to S4, although they do not necessarily have to be distinct. So, for example, S0 and S1 may be the same state.
- There may only be one trigger action and one return action, although they do not necessarily have to be distinct.
- The service and feature must synchronise on the trigger and the return actions.
- The feature may add additional services (as actions at its external interface) but these must not be found in the base services offered by the service.

This class of feature (as defined by the base triggering mechanism) offers a very limited range of potential functionality. However, it does have the advantage that the new system (service + feature) can be proven to be a refinement of the original service provided that the return event is eventually enabled. This is the role of fairness in our formal reasoning.

## *5.2 Towards a classification*

Let us consider a number of different ways in which we can change the triggering semantics of the base class, above, whilst still retaining the intuitive notion that a feature is being triggered from within a service.

### *Variation 1: multiple trigger and return instances*

The most obvious first step is to generalise the model to allow multiple instances of the same `trigger` and `return` actions. This is illustrated in figure 5, where we model 2 triggers and 2 returns.

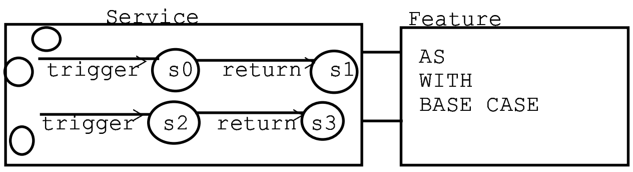

Figure 5: Multiple Trigger and Return instances — variation 1

The semantic rules which define this class of triggered feature are as follows:

- If treated in pairs, the trigger-return combinations all follow the rules for the base case.
- States S0 and S2 must be different but S1 and S3 may be the same.

*Variation 2: multiple return states*

The next variation allows multiple return states in the feature. This is illustrated in figure 6, where we model the fact that the feature can return control to the service from two different states (S4 and S5).

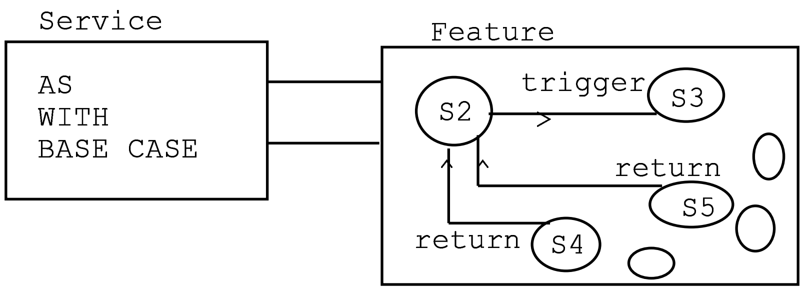

Figure 6: Multiple return states — variation 2

The semantic rules which define this class of triggered feature are as follows:

- If treated in pairs, the trigger-return combinations all follow the rules for the base case.
- States S4 and S5 must be different.

*Variation 3: different triggers and returns*

This variation is the most general and, as such, it is the most difficult to analyse. We allow multiple entry and exit points in the feature and we allow multiple triggers and return states in the original service. This is illustrated by an example in figure 7, where we have two triggers (t1 and t2) and two returns (r1 and r2). Furthermore, we see that the feature now has two inactive states (s7 and s4).

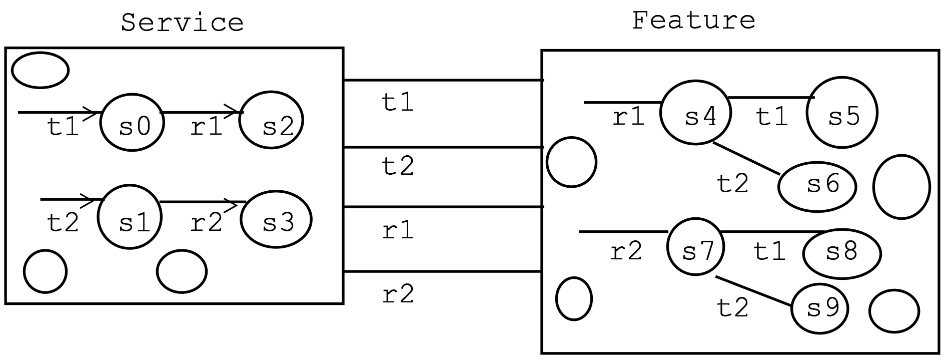

Figure 7: Most general trigger-return case — variation 3

The semantic rules which define this class of triggered feature are as follows:

- If treated in pairs, the trigger-return combinations all follow the rules for the base case.
- All inactive states of the feature class (those which can be reached through a return transition) must have all triggered actions enabled.

There are many different restrictions on this most general case (like the base case, variation1 and variation2). The more restrictions we place on the triggering mechanism, the easier it is for us to reason about the features and potential interactions. Our goal is to identify the most useful set of variations in order for us to formalise a re-usable analysis. The next variation shows that not all triggered features are defined as restrictions of variation3.

*Variation 4: no loss of control*

Here the idea is that the service does not pass its only thread of control to the feature. Instead, it continues with its behaviour as if the trigger feature did not exist. The trigger is used to do one of two things:

- start a flow of execution in a previously passive feature (which was deadlocked due to it waiting for a trigger action to be performed), or
- interrupt/influence a flow of control inside a previously active feature (which was carrying out its behaviour in parallel with the service)

In this variation, there is no need for a return action. Furthermore, the feature can have no influence whatsoever on the behaviour in the service it is triggered from. This is illustrated in figure 8.

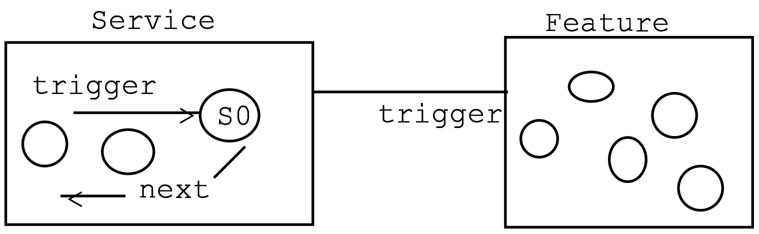

Figure 8: No loss of control — variation 4

Note that we do not connect the trigger to any particular state of the feature because, in the general case, the feature could be in any of its states. As with the trigger-return class, there are many different subclasses to this trigger-only class:

- **strict synchronisation** — the feature must perform a trigger action when the service requests it
- **optional synchronisation** — the feature may perform a trigger action when the service requests it
- **strict asynchronisation** — the feature must eventually perform a trigger action when the service requests it

- **optional asynchronisation** — the feature may eventually perform a trigger action when the service requests it

By default, we choose strict synchronisation; thus we have to ensure that the trigger action must be enabled in all the states of the feature.

### *5.3 Some eventuality analysis*

In [2], the notion of extension is equivalent to a triggered feature; but one where the feature *may* return to the service behaviour. We classify 2 different means of guaranteeing return to original behaviour.

#### *Guaranteeing eventual return*

In all the variations, we can see that it is possible for the behaviour to get stuck in a state of the feature whilst the user of the feature is continually requesting a return to the service. Now, if the feature is specified as a fair object we can guarantee that the return is eventually enabled and that we shall then return to the service behaviour.

A different situation arises when the return is an internal action of the feature. In other words, it is executed independent of the feature's environment. In such a case, we can only guarantee eventual return by specifying a suitable level of fairness on the return action itself (and even then we may require the service user to initiate some sequence of state transitions before this guarantee can be made.)

#### *Guaranteeing immediate return*

In some cases, when the return is an external action of the service, it may be required that the user can return to the service immediately. This can be guaranteed only by enabling such a return in all states of the feature.

#### *Fairness for eventual return in triggered features*

It should be clear that fairness has an important role to play in the classification of triggered features. In the next section we re-examine the fair object semantics, in a particular set of triggered features, so that we have a more formal foundation upon which to start building our analysis towards a triggered feature interaction classification.

## 6 Classification of DialIn triggered features and their interactions

### *6.1 A DialIn triggered feature list*

There were a wide range of features which could be said to be triggered by the `dialIn` action in POTS. We chose not to examine the call forward features (CF no-reply/busy/unconditional) because we assume that they are never passed onto the phone, being caught at the exchange inside the network, and re-routed centrally. Similarly, we did not consider the automatic call back feature. The features that we were left with are: three way calling (TWC), call hold (CH), caller identification (CID), call log (CLOG), answer machine (AM) and incoming call screening (ICS).

## *6.2 The informal classification through experimentation*

Following the variations in the previous section, we arrive at the following informal classification:

- TWC, CH — variation 3 (single trigger, multiple trigger instances, multiple returns)
- CID, CLOG — variation 4
- AM — variation 2 (single trigger, multiple return states, multiple returns)
- ICS — this does not correspond to any of our variations. The problem with this feature is that it appears to filter out some of the allowable traces in the original service behaviour and so the new service+feature model is definitely not a refinement of the old service model.

We note that this classification has arisen out of our limited experimentation (through specification and analysis of the fair objects requirements models for each feature). This is just a first step towards formalising the hierarchy, but our semantic rules do add some much needed rigour.

## *6.3 Classifying triggered interactions*

This is work in progress but we are experimenting in the following way:

> When analysing the interaction between 2 trigger features, F1 and F2. It is necessary only to identify the types of the features and these should be completely sufficient for us to apply a re-usable abstract analysis technique, parameterised by the feature types in question.

Again, the formulation of such abstract analysis can be carried out only after we have more thoroughly experimented with the fair object models and the trigger types. Unfortunately, this work is not yet ready for presentation. However, we can give you an idea of the type of results which we are hoping to achieve through automation in our analysis tools. In particular, we are most interested in identifying cases where interactions are guaranteed not to occur. For example:

- Two variation4 features cannot interact when they share the same trigger action provided they cannot block the trigger action from being executed and they have different return actions.
- Two variation3 features cannot interact on different trigger events provided they are guaranteed to eventually return control to the original service.
- If F1,F2 and F3 are triggered features then they cannot interact when all three are implemented in the same system provided no two of them interact in a *pair-wise* fashion

## 7 Re-usable Analysis for triggered interaction detection and resolution

Many of the classic triggered interaction problems arise when two features are triggered by the same action and the resulting introduction of non-determinism leads to inconsisent requirements. The most common technique to resolve such interactions is to use a priority mechanism. We are currently working on formulating a refinement of a system with two interacting triggered features which will resolve the interaction automatically. The idea is that the first time the non-determinism arises then the user is forced to make the choice. Consequently, any time after this choice has been made, the nondeterminism will be resolved in a manner consistent with the user's initial choice. This type of refinement could be made available as a correctness preserving transformation for use during design: instead of the designer having to resolve the interaction statically, they can gurantee that the resolution will be carried out dynamically. In this case, the designer's goal of having consistent resolution would have automated tool support.

The concept of stability may lead to an alternative solution where the invariants of each feature are weakened so that they have only to be true when that feature has actually been triggered — provided that they will always eventually be true then we do not mind if they are false while another feature is executing. We are currently working on defining new synchronisation mechanisms which incorporate stability into their semantics in order to facilitate the exiting of unstable states by prioritising those states which are stable (when an internal choice of actions is provided then a system which is constructed from the stable synchronisation operators will choose to *move towards a stable state.*) This is very much work in progress, but mention it for future reference as another approach to formalising the notion of resolution through priority. In this case the designer's goal of keeping the system as stable as possible would have automated tool support.

## 8 Conclusions

P. Zave, in [21], identifies the importance of 'failure of modularity'. We agree that we need to: *improve the feature specification and feature composition rules with respect to modularity.* The problem with modularity seems to be the choice between an approach based on information hiding and an architecture-driven approach. In a purely architecture driven approach, it was said that *a pretense of implementation independence is abandoned.* We do not, however, agree that an architectural approach gives rise to software specifications rather than requirements specifications. In all complex systems there is an inherent structure. In requirements, this structure helps to synthesise and analyse the models. It does not force the implementers to follow particular design decisions (although, they are free to do so if they desire).

In [4], a number of *weaknesses* in a formal methods approach are identified. We comment on these with respect to our approach: *Finding appropriate levels of abstraction is very hard* ... but using object oriented modelling facilitates moving between the levels as required. *Formal modelling and analysis is time consuming* ... but the extra time invested does lead to (semi-)automation of re-usable proofs which is a great saving. *New features are usually non-conservative extensions* ... but it is important that our approach can distinguish between those that are and those that are not. There is *little evidence of formal approaches uncovering unknown interactions* ... but using our approach we can predict interactions between non-standard variations on well-known features. *Most techniques are designed for expression rather than for tool support* ... but our formal fair object semantics are developed incrementally in parallel with our tool support for specification, validation and verification. The specifications *should be readable by a wide audience* ... as is supported by an object oriented model which facilitates specification of multi-view mappings to a wide range of different graphical representations. The methods are *targetted at the whole problem* ... clearly our work, to date, has not attempted to address all feature types. In fact, we could almost be accused of doing the opposite by focusing too closely on only triggered features. *They work only in a pair-wise fashion* ... but we believe this paper supports the view that an algebraic approach is the best way to get around this problem.

To conclude, this paper is a review of a lot of different threads of research that are coming together (slowly but surely). As work in progress, it is clear that much needs to be done but, as we stated from the outset, we believe it is best to take a small, steady step than to leap ahead in an uncontrolled fashion. A feature interactiona algebra *is* an ideal but it is an ideal which this paper brings us one step closer to reaching.

## References

[1] M. Abadi and L. Lamport. Conjoining Specifications. *ACM Transactions on Programming Languages and Systems*, 17(3):507–533, May 1995.

[2] A. Aho et al. Sculptor with chisel: Requirements engineering for communications services. In K. Kimbler and L.G. Bouma, editors, *Feature Interactions in Telecommunications and Software Systems V*, pages 45–66. IOS Press, 1998.

[3] N. Barreiro, J.L. Fiadeiro, and T. Maibaum. Politeness in Object Societies. In R. Wieringa and R. Feenstra, editors, *Information Systems: Correctness and Reusability*, pages 119–134. World Scientific Publishing Company, 1995.

[4] M. Calder. What use are formal design and analysis methods to telecommunications services. In K. Kimbler and L.G. Bouma, editors, *Feature Interactions in Telecommunications and Software Systems V*, pages 23–31. IOS Press, 1998.

[5] J. Cameron and F.J. Lin. Feature interactions in the new world. In K. Kimbler and L.G. Bouma, editors, *Feature Interactions in Telecommunications and Software Systems V*, pages 3–9. IOS Press, 1998.

[6] E. Canver and F.W. von Henke. Formal development of object-based systems in a temporal logic setting. In *Formal Methods for Open Object-Based Distributed Systems*, pages 419–436. Kluwer Academic Publishers, February 1999.

[7] J.L. Fiadeiro and T. Maibaum. Sometimes "Tomorrow" is "Sometime": Action Refinement in a Temporal Logic of Objects. In D. M. Gabbay and H. J. Ohlbach, editors, *Temporal Logic*. Springer Verlag, 1994.

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

[9] J.Paul Gibson. Feature requirements models: Understanding interactions. In Boutaba Dini and Logrippo, editors, *Feature Interactions in Telecommunication Networks IV*, pages 46–60. IOS Press, 1997.

[10] P. Gibson. Towards a feature interaction algebra. In K. Kimbler and L.G. Bouma, editors, *Feature Interactions in Telecommunications and Software Systems V*, pages 217–231. IOS Press, 1998.

[11] P. Gibson. Formal object oriented requirements: simulation, validation and verification. In Helena Szczerbicka, editor, *Modelling and Simulation: A tool for the next millenium , vol II, ESM99*, pages 103–111. SCS, 1999.

[12] P. Gibson, G. Hamilton, and D. Méry. Requirements integration problems in telephone feature development. In Andy Galloway Keijiro Araki and Kenji Taguchi, editors, *Integrated Formal Methods conference, IFM99*, pages 129–151. Springer Verlag, 1999.

[13] P. Gibson, G. Hamilton, and D. Méry. Composing fair objects. In *International Conference on Software Engineering Applied to Networking and Parallel/ Distributed Computing (SNPD '00)*, 2000. To be announced.

[14] P. Gibson and D. Méry. Fair objects. In H. Zedan and A. Cau, editors, *Object-oriented technology and computer systems re-engineering.* Horwood Publishing, 1998. Presented at COTSR, Object Technology (OT98), Oxford, UK.

[15] P. Gibson, Y. Mokhtari, and D. Méry. Animating formal specifications — a telephone simulation case study. In Helena Szczerbicka, editor, *Modelling and Simulation: A tool for the next millenium , vol II, ESM99*, pages 139–146. SCS, 1999.

[16] R.J. Hall. Feature combination and interaction detection via foreground/background models. In K. Kimbler and L.G. Bouma, editors, *Feature Interactions in Telecommunications and Software Systems V*, pages 232–246. IOS Press, 1998.

[17] H.-M. Järvinen and R. Kurki-Suonio. DisCo Specification Language: Marriage of Actions and Objects. In *11th International Conference on Distributed Computing Systems*, pages 142–151. IEEE Computer Society Press, 1991.

[18] L. Lamport. A temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994.

[19] Y. Peng et al. Feature interaction detection technique based on feature assumptions. In K. Kimbler and L.G. Bouma, editors, *Feature Interactions in Telecommunications and Software Systems V*, pages 291–298. IOS Press, 1998.

[20] M. Plath and M. Ryan. Plug-and-play features. In K. Kimbler and L.G. Bouma, editors, *Feature Interactions in Telecommunications and Software Systems V*, pages 150–164. IOS Press, 1998.

[21] P. Zave. Architectural solutions to feature-interaction problems in telecommunications. In K. Kimbler and L.G. Bouma, editors, *Feature Interactions in Telecommunications and Software Systems V*, pages 10–22. IOS Press, 1998.