Integration Problems in Telephone Feature Requirements

Paul Gibson (NUI, Maynooth, Ireland),pgibson@cs.may.ie
Geoff Hamilton (DCU, Dublin, Ireland),hamilton@compapp.dcu.ie
Dominique Méry (Université Henri Poincaré, Nancy, France), mery@loria.fr

November 6, 1998

Abstract

The feature interaction problem is prominent in telephone service development. Through a number of case studies, we have discovered that no single semantic framework is suitable for the synthesis and analysis of formal feature requirements models, and the choice of modelling language has certain knock-on effects on the transformational design steps which lead to implementation.

We initially describe a mixed semantic model approach whilst acknowledging that integration is a major concern. Our method incorporates operational state transition models, temporal logic formulae and object oriented structuring mechanisms. Each of these approaches gives rise to certain advantages and disadvantages, and we propose a complementary integration which allows the client to express their requirements in the way in which they understand their needs, whilst building formal models for transformation and verification during design.

This paper evaluates such a mixed semantic approach in the domain of telephone feature development. It motivates the need for a single formal language which provides a unified semantic framework and incorporates all of the desirable properties of each of the semantic models used in the mixed semantic approach. We propose the hidden algebraic approach as providing just such a suitable semantic framework.

1 Introduction

A common goal for researchers in the area of formal methods is to generalise the software development model to reflect the fact that different languages appear at different stages of development, facilitating different levels of abstraction and different semantic capabilities. Clearly we need a better understanding of *why* and *how* this happens.

Formal methods are becoming increasingly important in the development of rigorous engineering practices in many different problem domains. The research community has given birth to a large number of different methods and languages which are commonly accepted as being formal [41, 16, 1, 5, 33]. To promote the use of formal methods in industry we must produce a more unified model of formal software development [19], onto which individual approaches can be mapped.

Formal languages have a large number of different roles to play. Consequently, choosing a requirements modelling language involves a number of compromises: Should the language be best able to model the abstract or the concrete? Should the language be problem domain specific or problem domain independent? Should the language be oriented towards the client or the engineers? Should the language be informal, rigorous or fully formal? Should the language be state-of-the-art or well-established?

In this paper, we describe a multi-language approach to the specification of telephone features. We find that this approach has a number of disadvantages when compared to that of using a single language – a lack of

conceptual consistency, less potential for rigorous design techniques based on correctness preserving transformations, and no common vocabulary between all participants in the development process.

We propose the hidden algebraic approach [29] as a possible unified semantic framework which will allow telephone features to be specified using a single language.

2 The feature interaction problem

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[1]) does not as a whole satisfy each of its component features individually. We concentrate on the domain of telephone features [10, 7]. Consider the following 3 well known interactions:

- **Call line delivery and call line delivery restriction**
  One user has subscribed to the `call line delivery` feature in order to identify incoming callers. A second user does not wish anyone to see his[2] number when he makes a call and therefore subscribes to `call line delivery restriction`. If the second user calls the first then which feature will work correctly and which will not meet its specification? This feature interaction corresponds to a contradiction between different user's requirements.
- **Call forward and incoming call screening**
  A user never wants to talk to a certain number and so subscribes to `incoming call screening` to stop that caller from ever being connected. The user's friend is coming over for dinner and so forwards his calls (using the `call forward` service) to the first user's phone. The screened caller phones the user's friend and the call is transferred to the user. There is an interaction because the call forward feature 'would like' the call to be connected and the incoming call screening 'would like' to stop the connection. (The formal modelling of this interaction requires the concept of 'would like' to be rigorously defined.)
- **Call forward on busy and call hold**
  This interaction occurs if a user subscribes to both services and is already talking to someone when a third party tries to phone them. Both services 'are triggered' by this incoming call but the incoming call cannot be forwarded and held at the same time. (This example forms the basis of our case study in section 6, where the triggering and timing aspects are formally specified.)

2.1 User requirement types

Intuitively, we can see that the user often wishes to express different types of requirements:

- **Safety requirements** — where the user specifies things that must never happen. These can be state based, sequence based or property based. A state based safety requirement corresponds to the user never wanting to be in a certain concrete state (e.g. talking to his wife and mistress at the same time in a three-way-call). A sequence based safety requirement corresponds to the user never witnessing a sequence of external actions (e.g. putting the phone `on-hook` , lifting the phone `off-hook` and then hearing a `busy signal`). A property based safety requirement corresponds to a state based requirement where the state is specified abstractly over a number of possible states which are not explicitly listed (e.g. never wanting to pay for an overseas call).
- **Liveness requirements** — where the user specifies that something good will eventually happen. These usually correspond to different types of fairness or eventuality needs. For example, it may be important that a telephone exchange which can buffer incoming calls is forced to service the calls according to some fairness policy; or it may be important only that each call will eventually be serviced. Eventuality

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

[2] We apologise to readers for choosing to use the masculine form for representing both males and females.

requirements are common in user-oriented specifications because they help to abstract away from the network.

- **Nondeterministic and consistency requirements** — where the user specifies a number of different behaviours which would be acceptable and may require that the nondeterminism be resolved in a consistent fashion. For example, when we combine `call hold` with `three way calling` then the user may not mind which is triggered when they receive an incoming call whilst already connected to another phone. This can be resolved (internally) by the phone system. The user may, however, require consistent resolution whereby one service is always given *priority* over the other.
- **Compositional requirements** — where the user specifies new needs by 'combining' already existing services. For example, a user with an `answering machine` and `call hold` may wish to let a held caller leave a message while they are waiting, and thus give them the option of abandoning the call if they have to wait too long.
- **Specialisation and extension requirements** — where the user specifies new needs by making refinements to already existing services. For example, a user may require an answering machine which restricts the length of messages that can be left.

Corresponding to these requirements, we have proposed a mixed semantic approach to provide the following three different views:

An object-oriented view: this provides the structuring mechanisms necessary to compose and specialise feature specifications.

A fairness view: this allows the specification of liveness requirements

An invariant view: this allows the specification of safety requirements.

The object-oriented view is formalised using an object-labelled state transition semantics (O-LSTS [18]). Specifications within this view can be animated, allowing validation to be performed. The fairness view is formalised using TLA, the temporal logic of actions [31]. The invariant view is provided by translating O-LSTS specifications into a model-based specification language, and then using the model checker for the language to prove invariants. Invariants can also be proved using the TLA theorem prover (TLP [17]), but we have found it easier to prove invariants by translation to B [2].

2.2 A software engineering perspective

Features are observable behaviour and are therefore a requirements specification problem [42]. Most feature interaction problems can 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. The two key areas are validation of requirements models and verification of design steps. The peculiar nature of the feature development cycle, where having features as the incremental units of development is the source of many of our problems:

- *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.
- *Assumption Problem:* Phone systems have changed dramatically over the past ten years. Many people 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

(POTS). Furthermore, 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, it is difficult to develop new features in parallel. 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 composition problems.

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

2.3 Requirements — the need for 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.

In our formal approach, interactions can occur when requirements of multiple features are *contradictory*, or when design steps introduce contradictory assumptions into our models. The complexity of understanding the problem is thus contained within a definition of *contradiction* in our semantic framework.

2.4 Requirements — the need for integrated models

Many of the problems which arise when features combine (i.e. *feature interactions*) are due to *badly* developed requirements models for individual features [22]. 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. Unfortunately, customers have many different types of requirements and so the problem becomes one of facilitating multiple means of expression and having a formal integration of different points-of-view through verification of their consistency.

2.5 Transformation and Refinement — the need for a unifying model

Refinement and correctness preserving transformations

A transformation can be applied to a specification which reflects some architectural/implementation choice, without altering the external (observable) behaviour of the system. Such a design transformation is dependent on some nonstandard, though not necessarily informal, means of interpreting the internal details of the specification.

Design is the process which transforms an initially abstract (implementation independent) specification of system requirements into a final, more constructive, implementation oriented specification. In theory, it is possible to verify the correctness of any given design step by mathematical means. In practise, the complete formal verification of most design steps is not possible because of combinatorial problems. In these cases, specifications are partly verified by simulation and testing.

A different means of verifying a design is to perform only transformations (design changes) whose correctness has already been proven correctness preserving transformations (CPTs).

> A specification can be said to be *correct* if it fulfils some property. Assume a specification S, a transformation T and define $S' = T(S)$, i.e. S' is the result of applying T to S. T can be said to be correctness preserving with respect to the property P if $P(S) \Rightarrow P(S')$. In other words, the property P is preserved across the transformation T.

This type of formulation raises a number of interesting questions: What sort of properties can be usefully preserved? How can these properties be formalised? Over what domains should T operate? What is the difference between S and S' which makes T a useful transformation for applying during design? Can we specify appropriate transformations to correspond with decisions most commonly taken by designers in practice? These questions should be addressed within a unifying framework, and not just examined within the domain of one particular formal method.

Compositional refinement and parallel development

The object oriented approach complicates the design process even more. Development is neither top-down nor bottom-up, it is middle-out. Different parts of the system are at different levels of abstraction. A design step does not change the level of abstraction of a system, it changes the level of abstraction of a component of the system. Thus the notion of a correct transformation has to be able to be placed in the context of the environment of the class being transformed (i.e. the system in which it is being used). A unifying framework should facilitate the concurrent refinement of different system components so that design steps can be carried out independently.

Integrating mixed model refinements

A mixed semantic approach makes life even more difficult for designers. Transformations that have been proven correct within a single semantic model now have to be extended to be applicable in the other frameworks. A greater problem is that a transformation which maintains correctness in one approach may not maintain correctness in the mixed approach. For example, in a purely operational model with no liveness constraints a standard design decision is to remove a nondeterministic choice. This does not compromise the requirements in any way. However, removing such nondeterminism may mean that some liveness properties of the system are no longer satisfied. A unified model must address the question of how refinements can be defined within each of the earlier views.

3 Telephone requirements modelling

The process of requirements modelling 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 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. We require a method with: good structuring facilities for constructing models, good structuring facilities for reasoning about models, customer-oriented syntax and semantics for animation, means of expressing non-operational always and eventually properties, and a formal basis for design transformation and verification

3.1 Importance of structure

We advocate an object oriented approach to structuring our requirements models. Object oriented methods encompass a set of techniques which have been, and will continue to be, applied in the successful production of complex software systems [38, 11, 12, 6, 35]. The methods are based on the simple mathematical models of abstraction, classification, refinement and polymorphism.

3.2 Importance of customer-oriented animation

The analysis and requirements capture phases of software development should be *customer oriented*: it is generally agreed that customer communication is the most important aspect of the early stages of development [30, 37, 40]. The successful synthesis of a requirements model is dependent on being able to construct a system as the customer views the problem [25]: requirements validation is not possible if the models cannot be communicated to the customer.

The key is our operational object oriented semantics which, through graphical animation, provide support for customers to validate their understanding of their requirements rather than validating their understanding of the models. In an ideal environment the analyst would identify the set of concepts with which the customer understands their needs, map these concepts onto the formal semantics of one (or all) of our modelling languages and let the customer construct their own requirements model. In practice it is more feasible to expect the customer and analyst to work together during the construction and refinement of requirements (as well as at the testing stages).

3.3 Always and Eventually

Although we have listed a large number of different types of requirements, [26] shows that the notions of always and eventually are central to telephone users' models of their needs. Always properties are simply stated as invariants to be proved and eventually requirements are integrated into our models as fairness assumptions on objects which serve concurrent requests.

3.4 The importance of composition

In [23], we have shown the importance of formalising different composition mechanisms in the domain of telephone requirements models and emphasised the need for different formal methods. The final goal is a feature interaction algebra which can be used within any formal development method, independent of language. Using algebriac techniques we hope to provide a library of composition mechansims where re-use of analysis is prominent — instead of analysing the (potential) interactions between given features, we analyse the (potential) interactions between classes of features and identify the re-usable composition mechanisms which guarantee absence of interaction.

4 Our mixed semantic method: an overview

As an aid to comprehension we limit our study to interactions which deal only with the user's point of view. We also restrict our level of abstraction: problems due to sharing of resources, information hiding, interface realisation, etc ... are primarily 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: an object oriented view, a liveness view, and an invariant point of view.

4.1 Object Oriented state transition model

Labelled state transition systems are often used to provide executable models during the analysis and requirements stages of software development [13, 14]. In particular, such models play a role in many of the object oriented analysis and design methods [6, 12]. 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 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.

We adopt a simple object-labelled state transition system semantics (O-LSTS) which regards an object as a state transition machine [21]. The true advantage of the O-LSTS approach is in the initial requirements modelling phase of development. It improves the communication with the customer and aids in synthesis, analysis and validation. However, we have found the inability to express liveness requirements a main weakness when applying the O-LSTS approach. Without liveness we can specify only what cannot happen (i.e. safety properties) rather than what must happen. Furthermore, without temporal semantics based on liveness, the nondeterminism in a system can be specified only at one level of abstraction: namely that of an internal choice of events. This can lead to many problems in development.

This state based object view forms the basis on which we build our feature animations and facilitate 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. 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.
- *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 to meet its invariant reqirements when we have large (possibly infinite) numbers of states in our systems. By translating to a more proof-theoretic model we can verify these invariants statically by showing that the external services are closed with respect to the invariant properties.

The O-LSTS semantics have no means of expressing fairness and these must be defined informally as comments in the specification text. These comments can then be formalised when we translate to TLA.

4.2 Temporal Logic

Temporal logic extends classical logic by handling modalities such as fairness and eventuality. Our temporal framework is TLA [32] which is a temporal logic providing a notation for actions, namely a relation between unprimed and primed variables. A specification of a system is a formula stating the initial conditions, the transition relation and the assumptions over actions for strong and weak fairness :

$$Initial \wedge \Box[N]_x \wedge SF_x(A) \wedge WF_x(B) \tag{1}$$

TLA provides a simple and effective means of expressing fairness properties. The semantics incorporate the notions of *always* (represented by the $\Box$ operator) and *eventually* (represented by the $\Diamond$ operator). Using these, we can specify different categories of fairness [27] within the object oriented framework. The ability to model nondeterminism at different levels of abstraction is the key to TLA's utility in requirements modelling. Unfortunately, TLA does not provide the means for easily constructing and validating initial customer requirements. By combining TLA and object oriented semantics we can alleviate these problems.

We are interested in specifications that are machine-closed and the machine closure ensures that the specification corresponds to an implementable system. In equation 1, N denotes the relation between the current state and the next state of the system. In fact, it defines the transition system over states. However, TLA is not very useful as a requirements modelling language and a specification language which includes TLA, namely TLA$^+$, is used to provide basic structuring mechanisms and the means for expressing data in a set-theoretic

framework. Unfortunately, these structuring mechanisms are not as useful as those found in our object oriented model and it is difficult to specify re-usable structured concepts directly in TLA$^+$. Thus we advocate using the object oriented O-LSTS models as the basis from which we derive the *flatter* TLA$^+$ specifications.

Through experience [34] we have found that it is easier to prove invariants by translation to B [2] rather than through direct use of the TLA theorem prover (TLP [17]). The B abstract machine notation provides a very simple way to define a reactive system, when one considers that a service is a set of possibly enabled actions triggered when a guard is true. One can use B for modelling services as abstract machines and we can also use the B notations as a starting point for the expression of assumptions to derive eventuality properties in TLA.

TLP is currently used to prove eventuality properties but we hope to be able to axiomatise the fairness semantics of our fair objects in order to allow for automatic generation of a set of proof obligations to be discharged by the B theorem prover.

5 Integration —proof of concept

Although much of this paper talks about an integrated formal model, we have to emphasise that such integration is not fully formalised. Rather than trying to perform such an integration in one giant step, we chose to attempt some partial integrations in order to identify the problems and motivate a more thorough integration after the problems are better understood. We have identified three interesting integration areas: fair objects, algebraic objects and a unifying model of refinement.

5.1 Fair objects — eventuality in requirements

In [27], we examined how we could combine temporal semantics and object oriented concepts in a complementary fashion. High level re-usable concepts were formalised as different kinds of fair objects, defined by the abstract superclasses from which they inherit their eventuality properties.

5.2 Safe objects — invariants in requirements

Our work, up to now, has integrated the object oriented operational requirements with the proof of invariants through translation to a proof theoretic framework. The work by Mermet [34] shows the ways in which a theorem prover can automatically verify invariant properties and detect interactions as invariants which are broken. The success of this approach leads us to believe that it may be best to use a formal method which allows the object oriented concepts to co-exist with the invariants in one single semantic framework.

5.3 A unifying model of refinement

In [20] we introduced a unifying model for expressing three different types of development step: *specification refinement*, *program refinement* and *transformational refinement*. The model is generally applicable to any pair of *specification* and *implementation* languages, and, furthermore, can be extended to any number of development languages working in one coherent framework. Our current goal is to fit the object oriented paradigm onto this model.

Algebraic approaches are based on bisimulation relationships that allow one to show that terms are equivalent according to some correctness criterium. If we consider a program then we would like to be able to compare it with another program that is *better* with respect to some implementation environment. We have to define a relationship over programs and more generally we have to include specifications in the same framework. Category theory for studying combination of temporal theories, and its techniques of superposition are useful for transforming programs by preserving correctness properties. The seminal work of Back [3, 4] has founded the refinement calculus that is a transformational calculus over action systems in which the transformation preserves total correctness. The UNITY [9] approach is based on a mixed calculus that includes refinement

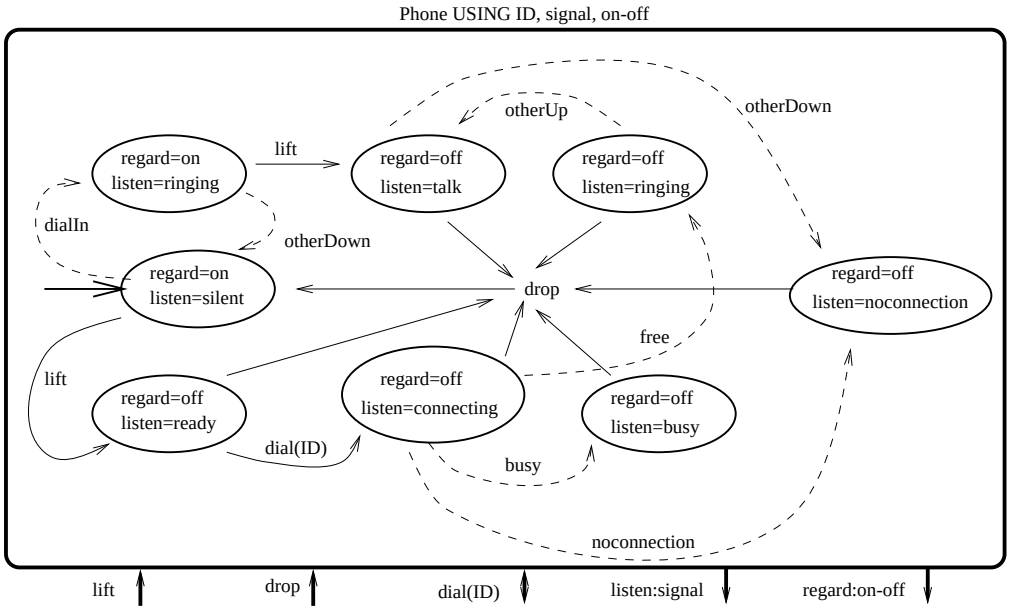

Figure 1: OOLSTS Specifi cation of a Telephone

calculus over action systems under weak fairness and refinement calculus over a temporal specification (the refinement preserves safety properties and eventuality properties under the weak fairness assumption).

6 A feature study —call waiting and call forwarding

Although we work within 3 different views, we do have a form of integration at the syntactic level. The O-LSTS requirements can be used to generate partial specifications in the other two views. Then, the object oriented structure is used to organise the compositional validation and verification processes (in all 3 views). Syntactically, we can intuitvely see that all three views, at their core, incorporate the 'same' state transition system. A more formal integration would require such intuition to be verified using a meta-semantic model. In [36] we provide an overview of the large number of feature specifications which we have developed. In this section, we examine one particular interaction, namely the problem of integrating call waiting and call forwarding.

6.1 The POTS specifications

6.1.1 Object oriented specification

`Telephone` and `POTS` class specifications are given in Figures 1 and 2.

The POTS specification defines the requirements of a network which acts as the interface between any number of telephones. The telephone requirements model specifies the telephone user requirements.

There are two main classes in the `POTS` requirements model, namely: `Telephone` and `POTS`. All other classes, except `Signal` and `Hook` (which are just simple enumeration types), that are used by these two main classes are not specific to the `POTS` problem domain.

6.1.2 TLA specification

The telephone system is modelled as a set of operations acting on calls. The POTS specification is decomposed into several steps. The first step includes definitions for observing calls, users, customers, media, billing statements, etc ... The second step models actions that are executed by the system administrator and by the user. For example, SUBSCRIBE and UNSUBSCRIBE are executed under the control of the system adminsitration but, when a service is subscribed to, the user can decide to ACTIVATE or to INHIBIT the service. We model the system management using TLA$^+$. Users interface with POTS through actions such as OFF_HOOK, ON_HOOK and DIAL, and the switch ensures the connection and the communication through actions. Fairness constraints can be easily added in the TLA$^+$ specification for achieving the basic service of POTS. For example, we specify weak fairness on the connection so that if a call can be continually connected then it will eventually be connected.

The specification of the POTS system is obtained by defining the next relation of POTS, that is a disjunction of actions of POTS and by choosing a suitable fairness constraints over actions: $pots.next = \exists X, Y \in Addresse : User\_To\_Switch(X, Y)$ means that the next relation is defined as the disjunction of actions of the user to the switch. $Spec(POTS)$ is simply $pots.Init \wedge \Box[pots.next]_{pots.var} \wedge pots.WF \wedge pots.SF$. Now, we have to define some properties of the service for the customer. Our specification was giving an operational view in a logical language but we need to express what we are waiting for in the service.

When one considers every service, we would like to express what we would like to do with these services. For instance, if X calls Y and if Y is not busy, then X will ring Y and we express it as follows:

$Spec(POTS) \Rightarrow ([Calling(X, Y) \wedge \Box\neg Busy(Y)] \rightsquigarrow Ringing(X, Y)$

6.2 Call waiting and call forwarding specifications

6.2.1 Object oriented specification

In [23] we have shown that both these services can be specified as simple state refinements of the state where a user is already talking and an incoming call arrives. In POTS, in such a case the incoming call would not be connected as the line is busy. However, with each of our two new features, this is not the case. The incoming call would trigger the creation of a new temporary object which specifies the additional requirements placed by the new service in an encapsulated manner.

Using the process algebra part of LOTOS we wrap our ADT specifications of the POTS, CW and CF objects inside well-defined interfaces as part of process specifications. Then we use the LOTOS synchronisation operators to specify the communication requirements (the triggering, for example).

6.2.2 TLA specification

$Spec(POTS)$ (in appendix A) contains all the requirements of the simple telephone system. However, one would also want to be able to specify services and define additional properties required of the new system. For example, when the service `Call Forward` (CF) is activated for Y, if X has forwarded his calls to Z and if X calls Y, then X will ring Z or if Z is busy, X will hear a busy tone. We express it as follows:

$Spec(CF) \Rightarrow ([Calling(X, Y) \wedge \Box active(CF, Y) \wedge \Box\neg Busy(cf(Y))] \rightsquigarrow Ringing(X, cf(Y))$

One can also write that if the CF service is active for X, then nobody will be able to connect with X.

$Spec(CF) \Rightarrow \Box\,[\forall X, Y \in Addresses : active(CF, X) \wedge cf(X) <> Y \Rightarrow \neg Ringing(X, Y)]$

The expression of properties like these is precisely the definition of our feature requirements. Howver, the new TLA specification must be validated and verified. It means that the additonal requirements must be accepted by the client and the system must be proved with respect to the requirements.

6.3 Validation

6.3.1 Object oriented specification

One of the main advantages of using an object oriented specification is in the area of validation. Since the structure of the problem domain is represented directly in the requirements model, we can perform a compositional validation. (This is not so advantageous with the simple telephone but was useful with POTS, where we validated the behaviour of the components before the whole system was checked.)

The validation of each component was carried out by completely checking all possible states in our state transition system requirements models. The new state of the system after a given transformation is specified by an operation. Term rewriting of ACT ONE expressions is used as the operational semantics for our requirements model. The new state of a system (after a transformation) is validated through the application of accessor operations.

The validation of the composed systems was done as follows. Firstly, we validated the behaviour of POTS with CF. Secondly, we validated the behaviour of POTS with CW. Then, during validation of POTS with CW and CF we found a problem ...as each of these features is triggered by the ame action in the same state, we introduced nondeterminism into the model. This is a type of feature interaction.

6.3.2 TLA specification

The validation of TLA specification is based on the animation of the temporal specification. A tool is under development [8] and has been designed for obtaining an abstract execution of the underlying model of the specification. We have used the Logic Solver of the B environment [39]. Further research is needed to obtain a better interface for the user.

6.4 Verification

6.4.1 Object oriented specification

The step from analyis to design requires an extension to the requirements model to incorporate semantics for object communication and concurrency. A process algebra provides a suitable formal model for the specification of these properties. LOTOS, which combines ACT ONE and a process algebra in one coherent semantic model, provides a means of constructing object oriented design semantics.

The design phase of our development starts with the transfer of an ADT specification into a full LOTOS specification. In this way, we can start to reason in terms of higher level constructs such as communicating processes. Quite deliberately, in our method, high-level design features are abstracted away from during requirements capture. The requirements model says *what* rather than *how*. ADTs do play a major role in LOTOS designs: they maintain the underlying abstract behaviour whilst the process algebra is used to define the more concrete high-level design properties.

Given a set of system requirements specified in ACT ONE, there are a number of different ways in which these requirements can be translated to an initial abstract LOTOS design. Object oriented designers must initially identify the communication aspects of the way in which the underlying object oriented behaviour is to be fulfilled. The designers of a system must decide how the behaviour is to be offered at its external interface (and what this external interface should look like). This simple decision can affect the rest of the design process. Identifying an object oriented communication model and specifying the translation from OO ACT ONE, is not simple. There are a number of alternative models and a number of ways in which these can be specified. Four of these alternatives are examined in the thesis by Gibson[24].

The principle in each case is that a process definition is parameterised by an ACT ONE class specification. Every operation associated at the external interface of the class must be an event which the LOTOS process can always synchronise with. The parameterisation of the event corresponds to the input/output parameters of the operation being requested. The process algebra is then used to coordinate inter-object communication.

6.4.2 TLA specification

The verification of a TLA specification is much more complex than the validation. In fact, one has to derive in a formal way a proof of properties from the specification of the current POTS. Fundamentally, the interaction is detected when we can not derive properly the proof of the system. It means that we have done a proof P of a property as $Spec(POTS) \Rightarrow Property$ and we instantiate a service. We must prove now that $Spec(NewPOTS) \Rightarrow Property$ and if we can not, it means that there is an interaction. We have developed this property with the B abstract machines and the invariant property and we defined the interaction as a violation of invariants. What is new with TLA is that we have to deal with fairness constraints and with well foundedness. Using TLA and the notion of proof, we can view the feature intraction as a violation of proofs.

6.5 Design and Transformation

The design and transformation part of development is concerned with two main issues. Firstly, we have to decide how to resolve interactions which are found during verification. Secondly, we have to move a an abstract system (which is interaction free) towards a concrete implementation. We must not risk intoducing interactions due to these design steps.

6.5.1 Object oriented specification

In [24], object oriented design is driven by correctness preserving transformations. This technique helps us to move towards more structured (concrete) designs by removing nondeterminism in our requirements models. In fact, the major design process is in filling out the details of the communication network (abstractly represented as POTS).

In [23], design techniques for resolving interactions are examined. In the CF-CW interaction, there is a technique which allows the introduced nondeterminism to be removed: either by the engineers (statically) or the telephone users (dynamically).

6.5.2 TLA specification

Proof of refinement in TLA is done by simple logical implication. One system refines another if it implies it. The advantage of this is that the proof framework is very elegant. The disadvantage is that it it is hard to prove systems in a compositional manner. Even when we identify problems due to proof obligations which cannot be proved, it is difficult to say exactly where in the specification is found the root of the problem. In other words, we cannot easily identify the interaction point between features.

7 Towards a unified semantic method

In our search for a single semantic framework which encompasses all three of the specified views, we have found the hidden algebraic approach described in [29] to be very promising. This approach incorporates the object-oriented and invariant views, and we are currently investugating how we can also incorporate the liveness view. The language which we use for specification within this approach is CafeOBJ [15], which incorporates hidden algebra with order-sorted algebra.

Hidden algebra incorporates the object-oriented view, as the initial goal of the approach was to give a semantics for the object paradigm. Order sorted algebra [28] provides a natural way to handle multiple inheritance, subtypes, coercions, overwriting and other object-oriented concepts. Hidden order sorted algebra [29] extends order sorted algebra in that it uses sorts for states in addition to the normal use of sorts for data values. The data value sorts can be used to define the attributes of a class, and the state sorts can be used to define behaviour. The sorts used for states are called *hidden* sorts, so the definition of the behaviour of a class is encapsulated. Methods are therefore defined as operations on these hidden sorts.

Hidden algebra also incorporates the invariant view, and the proving of invariants is made as simple and mechanical as possible. Within a model-based approach, proof obligations concern satisfiability, where we must prove for each operation that an output state exists given that the operation's preconditions have been met and that all output states are valid. Proof obligations within a hidden algebraic specification usually involve structural induction.

7.1 Specification of a telephone

The specification of a telephone is shown in Appendix B. The behaviour of the telephone is represented by the hidden sort Telephone. Users of Telephone objects cannot access this sort. All interaction must be through the attributes and methods. As external events can also affect the behaviour of objects, we also define external events as operations on hidden sorts. Rather than the operations on the object being defined in terms of how they alter the attributes of the object, the attributes are defined in terms of the aequences of operations which have been performed on the object.

7.2 Call waiting and call forwarding specifications

The specifications of call waiting and call forwarding are also shown in Appendix B. These are both defined by inheriting from the Telephone class, and adding any new or modified operations. Call-Wait and Call-Forward are both defined as subsorts of Telephone. Note that for both these features, the DialIn event can cause a normal telephone to be transformed into one in which the feature has been activated. The extension of a telephone to incorporate both these features is also included in Appendix B.

7.3 Validation

Validation of the specification can be performed by showing that a given term can be reduced using term rewriting logic to another term as expected. For example, we can use CafeOBJ to show that:
the term `listen(otherBusy(dial(lift(newTelephone))))` can be reduced to `busy`:

```
TEL-WITH-TWO-FEATURES> reduce listen(otherBusy(dial(lift(newTelephone))))) .
busy : Signal
```

CafeOBJ provides facilities which allow the user to get a trace of the rules which were applied during this reduction, and also to step through the reduction. Obviously, it would be preferrable if the specification could be represented graphically, and validation performed by animating this graphical representation. There is no reason why this could not be done along similar lines to the way in which it was performed for O-LSTS.

7.4 Verification

Verification of the system can be performed by showing that the specification of the composed features is a refinement of each one of the specifications of the individual features. This proof is by coinduction, and involves the following steps:

1. define a behavioural equivalence relation on terms belonging to models of the composed feature specification
2. prove that this relation holds for all the behavioural operators of the composed feature specification
3. prove that all the equations in the specifications of the individual features are satisifed (by replacing the '=' in these equations with the equivalence relation defined above) - if any of these equations cannot be proven, then there is an interaction

In the above example, it can be shown that this is not the case. This is because the state represented by the term:
`dialIn(lift(dialIn(newTelephone)))`

is not behaviourally equivalent within each of the specified features. The composition of these features cannot therefore be a refinement of both of the individual features. There is therefore an interaction between the features. An interaction occurs if the same term can be reduced to different terms within the specifications of each of the interacting features. This corresponds to the problem of name clashes when using multiple inheritance.

7.5 Design and transformation

Interactions can be resolved using methods analagous to the resolution of name clashes when using multiple inheritance. Where there is a clash (or interaction), this can be resolved by renaming one of the operations or by using a priority mechanism for selecting which method (or feature) is activated. This priority can either be set by the system by including it within the specification, or by the user — allowing them to choose which feature they would like activated whenever an interaction occurs.

A system priority mechanism can be achieved by restructuring the inheritance hierarchy to avoid the use of multiple inheritance. One feature must therefore come above the other within the inheritance hierarchy. In this case, whichever feature comes lower in the inheritance hierarchy will have priority when there is an interaction. This corresponds to the overwriting of a method, thus avoiding the problem of method name clashes which may occur when using multiple inheritance. In our example this would be done as follows:

```
mod* TEL-WITH-TWO-FEATURES {
   (Extending CALL-WAIT + CALL-FORWARD)

   [ Call-Wait < Call-Forward ]
}
```

Because `Call-Wait` is defined as a sub-sort of `Call-Forward`, it will have priority whenever there is an interaction.

A user priority mechanism can be achieved by adding new methods which allow the user to choose betwwen the interacting features. In our example, two new methods (`hold` and `forward`) can be added to the telephone which allow the user to choose between the two features as follows:

```
mod* TEL-WITH-TWO-FEATURES {
   (Extending CALL-WAIT + CALL-FORWARD)

   *[ Tel-With-Two-Features < Call-Wait Call-Forward ]* -- hidden behavioural sort

   bop hold : Tel-With-Two-Features -> Call-Wait        -- new method
   bop forward : Tel-With-Two-Features-> Call-Forward   -- new method
   bop dialIn : Telephone -> Tel-With-Two-Features      -- modified event

   var T: Tel-With-Two-Features

   -- modification of hook attribute

   eq hook(hold(T)) = hook(T)
   eq hook(forward(T)) = hook(T)

   -- modification of listen attribute

   ceq listen(dialin(T)) = choose, if listen(T)==talk
   ceq listen(hold(T)) = talk-hold, if listen(T)==choose
   ceq listen(forward(T)) = talk, if listen(T)==choose

   -- modification of ID attribute

   eq ID(hold(T)) = ID(T)
   eq ID(forward(T)) = ID(T)
   }
```

The priority between the features is therefore specified by the user by applying the appropriate method.

8 Conclusions

We have shown that different viewpoints must be considered when trying to perform formal specification within the domain of telecom feature development. We first of all considered the use of a mixed semantic model, in which different formal languages are used for each of these viewpoints. None of these languages could cover all of the required viewpoints; for example TLA has very poor structuring mechanisms and O-LSTS cannot be used to specify liveness. There are also a number of disadvantages associated with a mixed semantic approach; a lack of conceptual consistency, less potential for rigorous design techniques based on correctness preserving transformations, and no common vocabulary between all participants in the development process.

Hidden order sorted algebra is a very promising approach which could allow all the viewpoints to be included within a single semantic framework. This already incorporates object-oriented structuring mechanisms and safety. We can also use similar techniques to those which were used for O-LSTS to animate specifications, and thus perform validation. The only viewpoint which is missing is the liveness viewpoint. We have already made some progress in incorporating this into hidden order sorted algebra, and work is continuing in this area.

References

[1] J.-R. Abrial. A formal approach to large software constructions. In J. L. A van de Snepscheut, editor, *Mathematics for Program Construction*, pages 1–20. Springer-Verlag, june 1989. LNCS 375.

[2] J.-R. Abrial. *The B-Book*. Cambridge University Press, 1996.

[3] R. J. R. Back. On correct refinement of programs. *Journal of Computer and System Sciences*, 23(1):49–68, 1979.

[4] R. J. R. Back. Correctness preserving programs refinements: proof theory and applications. Mathematical Centre Tracts 131, Mathematical Centre, Amsterdam, 1980.

[5] D. Bjorner and C. B. Jones. *Formal Specifi cation and Software Development*. Prentice-Hall, 1982.

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

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

[8] D. Cansell and D. Méry. Interprétation de spécifications temporelles à l'aide d'un outil de preuve. In N. Lévy and Y. Ledru, editors, *AFADL'98*, septembre-octobre 1998.

[9] K. M. Chandy and J. Misra. *Parallel Program Design A Foundation*. Addison-Wesley Publishing Company, 1988. ISBN 0-201-05866-9.

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

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

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

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

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

[15] R. Diaconescu and K. Futatsagi. *CafeOBJ Report*. World Scientific, 1998.

[16] A. Diller. *An Introduction To Formal Methods*. John Wiley and Sons, 1990.

[17] U. Engberg. *TLP Manual-(release 2. 5a)*-PRELIMINARY. Department of Computer Science, Aarhus University, May 1994.

[18] J.-P. Gibson. *Formal Object Oriented Development of Software Systems Using LOTOS*. PhD thesis, Stirling University, August 1993. Tech. report CSM-114.

[19] J.-P. Gibson and D. Méry. A Unifying Model for Multi-Semantic Software Development. Rapport Interne CRIN-96-R-110, Linz (Austria), July 1996.

[20] J.-P. Gibson and D. Méry. A Unifying Model for Specification and Design. Rapport Interne CRIN-96-R-110, CRIN, Linz (Austria), July 1996.

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

[22] J. Paul Gibson. Feature requirements models: Understanding interactions. In *Feature Interactions In Telecommunications IV*, Montreal, Canada, June 1997. IOS Press.

[23] J. Paul Gibson. Towards a feature interaction algebra. In *Feature Interactions In Telecommunications V*, Lund,Sweden, September 1998. IOS Press.

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

[25] Paul Gibson. An object oriented requirements capture and analysis environment. Technical Report CRIN-98-R-010, CRIN, January 1998.

[26] Paul Gibson and Dominique Méry. Always and eventually in object models. In *ROOM2*, Bradford, June 1998.

[27] Paul Gibson and Dominique Méry. Fair objects. In *OT98 (COTSR)*, Oxford, May 1998.

[28] J Goguen and E. Diaconescu. An oxford survey of order sorted algebra. *Mathematical Structures in Computer Science*, 4:363–392, 1994.

[29] J. Goguen and G. Malcolm. A Hidden Agenda. Report CS97-538, Dept. of Computer Science and Engineering, University of California at San Diego, April 1997.

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

[31] L. Lamport. A Temporal Logic of Actions. *ACM Transactionson Programming Languages and Systems*, 16(3):872–923, May 1994.

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

[33] P. A. LINDSAY. A survey of mechanical support for formal reasoning. *SEJ*, January 1988.

[34] B. Mermet and D. Mery. Detection of service interactions: An approach with b. In *AFADL97*, Toulouse (France), 1997.

[35] B. Meyer. Re-usability: the case for object oriented design. *IEE Software Engineering*, March 1987.

[36] B Mermet P. Gibson and D. Méry. Specification of services in a compositional temporal logic. Rapport de fin du lot1 du marche no 961B CNET-CNRS CRIN, CRIN, 1997.

[37] D. T. Ross. Structured analysis (SA): A language for communicating ideas. In *IEE Transactions on Software Engineering*. IEE, 1977.

[38] James Rumbaugh et al. *Object oriented Modeling and Design*. Prentice-Hall, 1991.

[39] Steria Méditerrannée. *Atelier B, Version 3.2, Manuel de Référence du Langage B*. GEC Alsthom Transport and Steria Méditerrannée and SNCF and INRETS and RATP, 1997.

[40] K.J. Turner. SPLICE I: Specification using LOTOS for an interactive customer environment — phase 1. University of Stirling SPLICE Internal Technical Document, 1992.

[41] K.J.T. Turner. *Using FDTS: An Introduction To ESTELLE, LOTOS and SDL*. John Wiley and Sons, 1993.

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

A TLA Specification

```
------------------- MODULE telephone --------------
(* user_to_switch(UID,otherUID) defines the set   of events available
for   the user in the case of the basic call model telephone.
In the TLA we have commented out the aspects which are
relevant only to the global state  and are there only to connect the
telephone to the network of other phones *)

user_to_switch(UID,otherUID) ==
{LiftToOff_hook_ready(UID),
LiftToOff_hook_talking(UID),
DropToOn_hook_first(UID),
LiftToOff_hook_ringing(UID),
DropToOn_hook_first(UID), DropToOn_hook_last(UID),DropToOn_hook_after_dial(UID),
DropToOn_hook_after_busy(UID), DropToOn_hook_during_wait(UID),
Dial(UID, otherUID)}

EXTENDS basic,invariant

LiftToOff_hook_ready(UID) ==
 /\ UID \in ADDRESSES
 /\ phones \in [ADDRESSES -> PHONES_STATES]
 /\ phones[UID] = "ready"
 /\ phones' = [phones  EXCEPT ![UID] = "silent"]
 /\ UNCHANGED << {(* some global system state is unchanged here *) >>

LiftToOff_hook_ringing(UID) ==
 /\ UID \in ADDRESSES
 /\ phones \in [ADDRESSES -> PHONES_STATES]

(* Global system state updates *)

IN /\ phones' = [[phones  EXCEPT ![UID] = "talking"] EXCEPT ![y] = "talking"]
       /\ calls' = (calls \ {c}) \cup {nc}
  /\ UNCHANGED << (* system state *) >>

DropToOn_hook_first(UID) ==
 /\ UID \in ADDRESSES
 /\ phones \in [ADDRESSES -> PHONES_STATES]
 /\ phones[UID] = "talking"

(* Global system state updates *)

    IN /\ phones' = [phones  EXCEPT ![UID] = "ready"]
       /\ calls'= (calls-{c}) \cup {nc}
       /\ recorded_calls' =  recorded_calls \cup {c}
  /\ UNCHANGED << (* global system state *)>>

DropToOn_hook_last(UID) ==
 /\ UID \in ADDRESSES
 /\ phones \in [ADDRESSES -> PHONES_STATES]
 /\ phones[UID] = "disconnecting"
(* Gobal system state updates *)
{\it
 /\ \E c : /\ (c \in calls)
           /\ (c.state = "completion_state")
           /\ \/ (c.calling_party = <<UID,"disconnecting">>)
              \/ (c.called_party = <<UID,"disconnecting">>)
           /\ (c.state = "established_state")
 /\ LET c == CHOOSE ac : /\ (ac \in recorded_calls)
```

```
                        /\ \/ (ac.calling_party = UID)
                           \/ (ac.called_party = UID)
                        /\ (ac.state = "completion_state")
       nc == CHOOSE ac : /\ (ac \in (CALLS-recorded_calls)-calls)
                         /\ (ac.state = "completion_call")
                         /\ (ac.called_party = c.called_party)
                         /\ (ac.calling_party = c.calling_party)
                         /\ (ac.state = "terminated_call")
                         /\ (ac.stime = c.stime)
                         /\ (ac.ftime = c.ftime)
                         /\ (ac.paying_party = c.paying_party)
    }
    IN /\ phones' = [phones  EXCEPT ![UID] = "ready"]
       /\ calls' =  calls-{c}
       /\ recorded_calls' =  recorded_calls \cup {nc}
  /\ UNCHANGED << (* global system state *)>>

DropToOn_hook_after_dial(UID) ==
 /\ UID \in ADDRESSES
 /\ phones \in [ADDRESSES -> PHONES_STATES]
 /\ phones[UID] = "dialling"
 /\ phones' = [phones  EXCEPT ![UID] = "ready"]
 /\ UNCHANGED << (* global system state *)>>

DropToOn_hook_after_busy(UID) ==
 /\ UID \in ADDRESSES
 /\ phones \in [ADDRESSES -> PHONES_STATES]
 /\ phones[UID] = "busy"
 /\ phones' = [phones  EXCEPT ![UID] = "ready"]
 /\ UNCHANGED << (* global system state *)>>

DropToOn_hook_during_wait(UID) ==
 /\ UID \in ADDRESSES
 /\ phones \in [ADDRESSES -> PHONES_STATES]
 /\ phones[UID] = "ringing"

(* global system state updates *)

IN /\ phones' = [phones EXCEPT ![UID] = "ready"]
       /\ calls' = calls \ {c}
       /\ fail_calls' = fail_calls \cup {c}
       /\ UNCHANGED << (* system state *) >>

waitingatDial(UID,otherUID)  ==
 /\ UID \in ADDRESSES
 /\ otherUID \in ADDRESSES
 /\ phones \in [ADDRESSES -> PHONES_STATES]
 /\ phones[UID] = "dialling"
 /\ phones' = [phones EXCEPT![UID] = <<"dialling",otherUID>>]
 /\ UNCHANGED << (* global state *) >>
```

B Algebraic specification

The folowing is a specification of a TELEPHONE using CafeOBJ:

```
mod* TELEPHONE {
   protecting(UID + Hook + Signal)

   *[ Telephone ]*                              -- hidden sort for telephone behaviour
```

```
   op newTelephone(ID) : -> Telephone            -- initial state
   bop hook : Telephone -> Hook                  -- attribute
   bop listen : Telephone -> Signal              -- attribute
   bop ID : Telephone -> UID                     -- attribute
   bop lift : Telephone -> Telephone             -- method
   bop drop : Telephone -> Telephone             -- method
   bop dial : Telephone -> Telephone             -- method
   bop dialIn : Telephone -> Telephone           -- external event
   bop otherBusy : Telephone -> Telephone        -- external event
   bop otherFree : Telephone -> Telephone        -- external event
   bop otherChangeHook : Telephone -> Telephone  -- external event
   bop noConnection : Telephone -> Telephone     -- external event
   op TelephoneEXC : -> Telephone                -- for errors

   var T : Telephone

   -- Definition of hook attribute

   eq hook(newTelephone(ID)) = on
   eq hook(lift(T)) = off
   eq hook(drop(T)) = on
   eq hook(dialIn(T)) = hook(T)
   eq hook(dial(T)) = hook(T)
   eq hook(otherBusy(T)) = hook(T)
   eq hook(otherFree(T)) = hook(T)
   eq hook(otherChangeHook(T)) = hook(T)
   eq hook(noConnection(T)) = hook(T)
   eq hook(TelephoneEXC) = TelephoneEXC

   -- Definition of listen attribute

   eq listen(newTelephone(ID)) = silent
   ceq listen(lift(T)) = ready, if hook(T)==on and listen(T)==silent
   ceq listen(lift(T)) = talk, if hook(T)==on and listen(T)==ringing
   eq listen(drop(T)) = silent
   ceq listen(dialIn(T)) = ringing, if hook(T)==on and listen(T)==silent
   ceq listen(dial(T)) = connecting, if listen(T)==ready
   ceq listen(otherBusy(T)) = busy, if listen(T)==connecting
   ceq listen(otherFree(T)) = ringing, if listen(T)==connecting
   ceq listen(otherChangeHook(T)) = silent, if hook(T)==on and listen(T)==ringing
   ceq listen(otherChangeHook(T)) = talk, if hook(T)==off and listen(T)==ringing
   ceq listen(otherChangeHook(T)) = noconnection, if listen(T)==talk
   ceq listen(noconnection(T)) = noconnection, if listen(T)==connecting
   eq listen(TelephoneEXC) = TelephoneEXC

   -- Definition of ID attribute

   eq ID(newTelephone(ID)) = ID
   eq ID(lift(T)) = ID(T)
   eq ID(drop(T)) = ID(T)
   eq ID((dialIn(T)) = ID(T)
   eq ID(dial(T)) = ID(T)
   eq ID(otherBusy(T)) = ID(T)
   eq ID(otherFree(T)) = ID(T)
   eq ID(otherChangeHook(T)) = ID(T)
   eq ID(noConnection(T)) = ID(T)
   eq ID(TelephoneEXC) = TelephoneEXC

}
```

Call waiting can be defined as follows:

```
mod* CALL-WAIT {
   (Extending TELEPHONE)

   *[ Call-Wait < Telephone ]*                   -- hidden behavioural sort

   bop switch : Call-Wait -> Call-Wait           -- new method
   bop dialIn : Telephone -> Call-Wait           -- modified event
   bop drop : Call-Wait -> Telephone             -- modified event
   bop otherChangeHook : Call-Wait -> Telephone  -- modified event

   var C : Call-Wait

   -- modification of hook attribute

   eq hook(switch(C)) = hook(C)

   -- modification of listen attribute

   ceq listen(dialIn(C)) = talk-hold, if listen(C)==talk
   ceq listen(switch(C)) = hold-talk, if listen(C)==talk-hold
   ceq listen(switch(C)) = talk-hold, if listen(C)==hold-talk
   ceq listen(otherChangeHook(C)) = talk, if listen(C)==talk-hold or listen(C)==hold-
talk

   -- modification of ID attribute

   eq ID(switch(C)) = ID(C)
}
```

Call forward on busy can be defined as follows:

```
mod* CALL-FORWARD {
   (Extending TELEPHONE)

   *[ Call-Forward < Telephone ]*                   -- hidden behavioural sort

   bop dialIn : Telephone -> Call-Forward           -- modified event
   bop drop : Call-Forward -> Telephone             -- modified event
   bop otherChangeHook : Call-Forward -> Telephone  -- modified event

   var F : Call-Forward

   ceq listen(dialin(F)) = talk, if listen(F)==talk
}
```

A telephone with both of these features can now be created as follows:

```
mod* TEL-WITH-TWO-FEATURES {
   (Extending CALL-WAIT + CALL-FORWARD)
}
```