Formal modelling of services for getting a better understanding of the feature interaction problem:

A multi-view approach

Paul Gibson[1] and Dominique Méry[2]

[1] NUI, Maynooth, Ireland
email: pgibson@cs.may.ie
[2] Université Henri Poincaré-Nancy 1 & LORIA UMR 7503 CNRS
Campus Scientifique, BP 239,
54506 Vandœuvre-lès-Nancy, (France)
email: mery@loria.fr

Abstract. We report results of a joint project with France Telecom on the modelling of telephone services (features) using formal methodologies such as OO ACT ONE, B and TLA$^+$. We show how we formalise the feature interaction problem in a multi-view model, and we examine issues such as animation, validation, proof and verification.

1 Introduction

In this section we briefly introduce the need for formal methods in software engineering, the use of formal methods to help resolve the feature interaction problem, and the particular formal methods we adopt in our mixed-semantic model.

1.1 Formality

Many software engineers do not acknowledge the value of formality. In 1993, a major study [13] concluded by stating: *"... formal methods, while still immature in certain important respects, are beginning to be used seriously and successfully by industry to design and develop computer systems ..."* We believe that formal methods are, five years later, *just about* ready for transfer to the industrial development of telephone features. Like all forms of engineering, one must always compromise between quality and cost. In telephone systems, it appears that the cost of resolving interactions between features at the implementation stage is now (or will soon be) greater than the cost of developing formal features requirements models and eliminating many of the potential interactions before implementation begins. Formal methods in this domain should be regarded as an investment for the future.
There are a wide and varied range of definitions of *formal method* which can be found in the majority of texts concerned with mathematical rigour in computer

science. The most common methods used for telephone feature specification are reviewed in [42]. For the purposes of this paper we propose the following definition: *A formal method is any technique concerned with the construction and/or analysis of mathematical models which aid the development of computer systems.* Formal methods are fundamentally concerned with *correctness*: the property that an abstract model fulfils a set of well defined *requirements*. In this paper, we are concerned with the construction of such requirements models.
A formal model of requirements is unambiguous — there is only one correct way to interpret the behaviour being defined. Although the model must still be mapped onto the real world (i.e. validated by the customer), this mapping is in essence more rigorous than in informal approaches. Building a formal model requires a better understanding of the problem domain and a better understanding of how the problem domain is viewed by the customer.
A major problem when using formal methods in software engineering is that much of the recent research places emphasis on analysis rather than synthesis. The means of constructing complex formal models is often overlooked in favour of techniques for analysing models.
Re-usable analysis techniques will automatically arise out of re-usable composition mechanisms. Formal method engineers need to learn techniques for building very large, complex systems. Such techniques have been followed, with various degrees of success, by programmers. In particular, object oriented programmers have evolved techniques which have been successfully transferred to the analysis and design phases of software engineering. Where better then to look for aid in the construction of large formal models?

1.2 Feature Interactions

A *feature interaction* is a situation in which system behaviour is specified as a composition of some set of features: each individual feature can meet its requirements in isolation but all features cannot meet their requirements when composed.
The problem of feature interaction is a major topic in telecommunications where formal methods have been usefully applied. There is no single technique which addresses all the aspects of the problem, but the most common approaches that have been used to tackle the problem, at the requirements stage, are: SDL [29, 30], LOTOS [18, 7, 17], state machine and rule based representation [19], and temporal logic[4, 3, 11], .

1.3 Our formal models

In our formal approach, interactions occur only when requirements of multiple features are *contradictory*. The complexity of understanding the problem is thus contained within a definition of *contradiction* in our semantic framework. We have argued that in most of the feature interaction examples found in published texts, there is no generally accepted standard formal definition of feature interaction[25, 43, 6, 9, 15]. In fact, most of the interactions which we studied correspond

to incomplete and informal requirements models. In other words, if the features were modelled *better* then we would be able to better understand what is and what isn't an interaction.

LOTOS (Language Of Temporal Ordering Specifications), see [40, 28], is a wide spectrum language, which is suitable for specifying systems at various levels of abstraction. Consequently, it can be used at both ends of the software development spectrum. Its natural division into ADT part (based on ACT ONE [16]) and process algebra part (similar to CSP [26] and CCS [37]) is advantageous since it provides the flexibility of two different semantic models for expressing behaviour, whilst managing to integrate them in a relatively coherent fashion.

LOTOS provides an elegant way to specify services and to detect interaction among services; it allows the user to specify services in a compositional manner and it provides a set of tools such as LITE from the project LOTOSPHERE[1], to assist in service engineering. Questions regarding fairness cannot be easily expressed or solved in LOTOS: modeling fairness requires us to state properties on traces, or a scheduling policy, and LOTOS has not yet integrated fairness constraints.

We have used LOTOS in our project and compared the expressivity of different languages such as B, TLA$^+$ and OO ACT ONE LOTOS and on the availability of practical development environments for B and LOTOS. The style of specification plays a very important role and the approach of Gammelgaard [19] is automaton-oriented; their approach uses a specification language based on transition systems as predicates. The weakness of their solution relies on the partial view of details whereas a sound and semantically complete reasoning system is required. The solution using TLA [31, 24] borrows the initial idea from their model, but TLA has the advantage of a very carefully equipped proof system. Finally, as the temporal framework can be very expressive, we need a computer-aided proof environment and more generally applicable software environments based on these formalisms.

Blow et Al. [3] and Middelburg [36] investigate the use of temporal logic for specifying services; Blom uses a temporal logic integrating the reactive and the frame parts for services. Middelburg introduces a temporal logic of branching time and restricts its expressivity to obtain a TLA-like logic.

In fact, the integration of very different formalisms such as TLA, B and LOTOS is a way to improve service engineering. B is simple and a tool helps the user in developing specifications: we do not claim that B will solve the entire problem but it is very helpful in the building of requirements models for telecommunication services. As we emphasize B as a tool for developing services specifications using a theorem prover, another crucial element of B is its animator. Several problems are detected by animation which do not need to be resolved by the prover. We have experimented with B as a tool for service engineering, although it was not one of the original goals of the language. Another point is that B and TLA are very close, at least for the action part; we have studied the integration of B and

[1] (see http://wwwtios.cs.utwente.nl/lotos/)

TLA [35] to re-use the B tools for TLA and to extend the scope of B through temporal features.
Our paper is organized as follows. Section 2 describes our mixed model involving different aspects of the formal development. Section 3 introduces service requirements. Section 4 gives details on the way we model services in TLA$^+$; we explain how our mixed views can be checked to be coherent. Section 5 concludes our paper.

2 A Mixed Semantic Model

We have shown the need for a mixed semantic model when specifying telephone feature requirements [22]. Such a model is used to provide three different client views:

- An *object oriented* view which provides the operational semantics used during animation for validation, and the structuring mechanisms which are fundamental to our approach. This view is formalised using an object oriented style of specification in LOTOS [20].
- An *invariant* view which allows the client to describe abstract properties of a system (or component) which must *always* be true. This view is formalised using B and leads to the automatic detection of many interactions [33, 34].
- A *fairness* view which allows the client to describe properties of the system which must *eventually* be true even though they have no direct control over them. A temporal logic provides an ideal means of specifying and verifying such requirements [23].

2.1 Objects and Classes

Labelled state transition systems are often used to provide executable models during the analysis and requirements stages of software development [12, 14]. In particular, such models play a role in many of the object oriented analysis and design methods [5, 10]. 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. Further, such an approach provides a more constructive means of testing actual behaviour with required behaviour.
This state based view forms the basis on which we build our feature animations and permits behaviour validation in a compositional manner. However, such operational models are not *good* for formal reasoning about feature requirements [44]: for this we need to consider specification of state invariants and fairness properties.

2.2 Invariants

Invariants are used to specify properties of a system which must *always* be true for reachable states. Within the object oriented framework we have three kinds of invariant:

- **Typing:** By stating that all objects are defined to be members of some class we are in fact specifying an invariant. These invariants are verified automatically by our object oriented 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. These invariants are also verified automatically by the object oriented 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 invariant. This cannot be statically verified using the object oriented tools, but it can be treated through a dynamic analysis (model check). Unfortunately, such a model check cannot be guaranteed when we have a large (possibly infinite) number of states in our systems. For this reason we need to utilise a less operational framework. By translating our state invariant requirements into B, we have been able to statically verify our state component invariants.

2.3 Nondeterminism and fairness

TLA is a temporal logic introduced by Lamport [31] and based on the action-as-relation principle. A system is considered as a set of actions, namely a logical disjunction of predicates relating values of variables before the activation of an action and values of variables after the activation of an action; a system is modeled as a set of traces over a set of states. The specifier may decide to ignore traces that do not satisfy a scheduling policy such as strong or weak fairness, and temporal operators such as $\Box$ (Always) or $\Diamond$ (Eventually) are combined to express these assumptions over the set of traces. Such fairness is important in feature specification and cannot be easily expressed using our state based semantics. The key is the need for nondeterminism in our requirements models. Without a temporal logic, nondeterminism in the features can be specified only at one level of abstraction: namely that of an internal choice of events. This can lead to many problems in development. For example, consider the specification of a shared database. This database must handle multiple, parallel requests from clients. The order in which these requests are processed is required to be nondeterministic. This is easily specified in our object model. However, the requirements are now refined to state that every request must be eventually served (this is a fairness requirement which we cannot directly express in our semantic framework). The only way this can be done is to *over-specify* the requirement by defining how this fairness is to be achieved (for example, by explicitly queueing the requests). This is bad because we are enforcing implementation decisions at the requirements level. With TLA we can express fairness requirements without having to say how these requirements are to be met.

2.4 Composition Mechanisms

Composition is primarily a question of re-use: given two already specified *components*, how can we create a new *component* from those given? A composition mechanism defines a creation mechanism which is reusable (i.e. can be applied to different sets of components). Clearly, we have to be more precise as to the meaning of a *component*. ¿From the customer's point of view, and hence at the requirements level of abstraction, a component must be some piece of behaviour which can be validated independently. In other words, a component must be able to be seen as a model of behaviour in its own right. We give an overview of the composition techniques from each of our three different view points and argue that a user oriented view would be best during requirements capture:

(1) Object oriented composition in LOTOS:
LOTOS [8] is made up from an abstract data type part [32], and a process algebra part [26]. Clearly there are ways of composing behaviours in each of these models. However, the object oriented composition is at a higher level of abstraction. We do not compose with language operators; rather we compose using object oriented concepts.

(2) Invariant composition (in B):
B [1] is a model-oriented method providing a complete development process from abstract specification towards implementations through step-by-step refinement of abstract machines. An abstract machine describes data, operations and invariant preserved by every operation. Abstract machines are composed by conjunction of its invariants and combination of operations. The resulting abstract machine may either preserve the resulting invariant, or invalidate it. The violation of the invariant is interpreted as an interaction [34] and is in fact an interference between operations: it is a way to detect interaction among services specified as abstract machines. The main advantage of B is that it is supported by a powerful sofware environment, namely the Atelier B [39]. The B method [1] is itself a conceptual tool for specifying, refining and developing systems in a mathematical and rigorous, but simple way.

(3) Fairness composition (in TLA):
The composition of fairness assumptions in TLA is done at a high level of abstraction and is preserved through the composition process. A model for a TLA formula is an infinite trace of states, and a TLA specification is made up of three parts:

- the initial conditions, Init,
- the relation over variables, $\mathrm{Next}(x, x')$, and
- the fairness constraints, $\bigwedge_{A \in \mathrm{WFA}} \mathrm{WF}_x(A) \wedge \bigwedge_{A \in \mathrm{SFA}} \mathrm{SF}_x(A)$
  (we require that $A \Rightarrow \mathrm{Next}(x, x')$, for all A in WFA or SFA to ensure the machine-closure property).

Fairness constraints remove models or traces that do not satisfy them. A service is characterized by a set of flexible variables, initial conditions, a next relation over variables and fairness contraints. When combining two services, we increase

the restrictions over traces but we extend the models by adding new variables. TLA provides an abstract way to state fairness assumptions but in our approach this unfriendly syntax is hidden from the customer. We encapsulate fairness within each object as a means of resolving nondeterminism due to internal state transitions. This is a simple yet powerful way for the fairness to be structured and re-used within our requirements models.

(4) Feature composition (user conceptualisation):

In an ideal world, feature composition would be done using concepts within the clients' conceptual model of their requirements. Clients cannot be expected to express themselves using formal language operators. This does not mean that they cannot express themselves formally. It is the role of the analyst to map the clients' composition concepts onto composition methods in the formal model. For now, we are forced to communicate through the object oriented models (which could be argued to be *client friendly*). In the future we hope to develop a modeling language based on client concepts rather than modeling language concepts.

3 Requirements for features

3.1 Requirements Modeling: Customer Orientation

Requirements capture is the first step in the process of meeting customer needs. Building and analysing a model of customer needs, with the intention of passing the result of such a process to system designers, is the least well understood aspect of software engineering. The process is required to fulfil two very different needs: the customer must be convinced that requirements are completely understood and recorded, and the designer must be able to use the requirements to produce a structure around which an implementation can be developed and tested. In this paper, we concentrate on the customers' point of view, whilst noting that the object oriented approach does lend itself to meeting the designers' needs [21]. We advocate such a *customer oriented* approach since it is generally agreed that customer communication is the most important aspect of analysis [27, 38, 41].

The fundamental principle of requirements capture is the improvement of mutual understanding between customer and analyst, and the recording and validation of such an understanding in a structured model. The successful synthesis of a requirements model is dependent on being able to construct a system as the customer views the problem. [2, 25] illustrate this point with respect to feature models.

3.2 Feature Interaction: What's new?

We concentrate on the domain of telephone features, where the problem has been acknowledged for many years![6, 9]. Figure 1 illustrates the problem within the formal framework which we adopt throughout this paper. We note that the means by which features are composed is not specified.

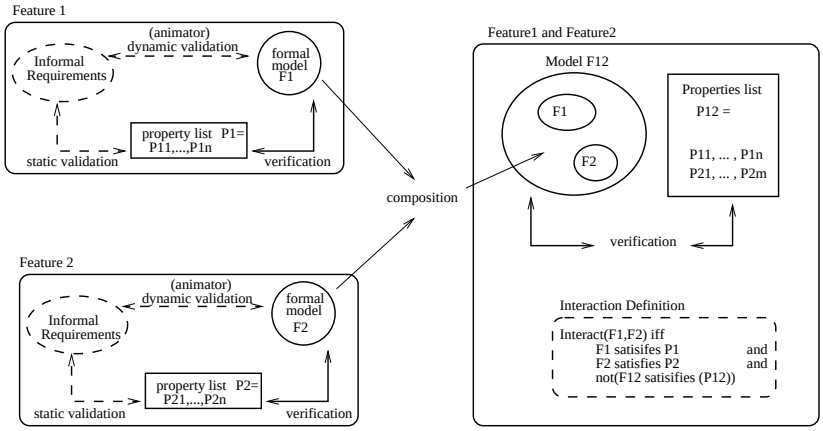

Fig. 1. Feature Interaction: A formalisation

Features are observable behaviour and are therefore a requirements specification problem [45]. Many feature interaction problems can be resolved through communication with the customer during requirements capture. Given a feature requirements specification which is not contradictory, interaction problems during the design and implementation will arise only through errors in the refinement process. Certainly the feature interaction problem is more prone to the introduction of such errors because of the highly concurrent and distributed nature of the underlying implementation domain, but this is for consideration *after* each individual feature's requirements have been modelled and validated. We have extended the work given in [25], where the composition of features was done in an ad-hoc fashion, by identifying and formalising re-usable composition mechanisms. The configuration of multiple features will be shown to depend on the way in which individual features are composed with POTS (plain old telephone service).

Features are requirements modules *and* the units of incrementation as systems evolve. A telecom system is a set of features. Having features as the incremental units of development is the source of our complexity. An understanding of feature composition helps us manage the four main sources of this complexity —

(1) State explosion:

Potential feature interactions increase exponentially with the number of features in the system and traditional model checking techniques cannot cope with the complexity. The fundamental problem is that analysis cannot be done compositionally. We argue that compositional (re-usable) analysis depends on having a formal understanding of the composition mechanisms. This is the main goal of this work.

(2) Chaotic Information Structure In Sequential Development Strategies:

The arbitrary sequential ordering of feature development is what drives the in-

ternal structure of the resulting system. As each new feature is added the feature must *potentially* include details of how it is to be configured with all the features already in the system. Consequently, to understand the behaviour of one feature, it is necessary to examine the specification of all the features in the system. All conceptual integrity is lost since the distribution of knowledge is *potentially* chaotic. At the moment this is certainly true. However, we believe that we can control the distribution of this *configuration knowledge* by containing it within a re-usable set of configuration mechanisms.

(3) Implicit Assumption Problem:
Already developed features often rely on assumptions which are no longer true when later features are conceived. Consequently, features may rely on contradictory (implicit) assumptions. This is a great source of interactions. We propose forcing the specifiers to formalise their (explicit) assumptions, by forcing them to use a certain set of configuration mechanisms.

(4) Independent Development:
Traditional approaches require a new feature developer to consider how the feature operates with all others already on the system. Consequently, we cannot concurrently develop new features: since how the new features work together will not be considered by either of the two independent feature developers. This problem is amplified if feature developers can configure features in any way that they wish.

4 Feature Interaction: An incremental development view

In figure 2, we take POTS as one requirement model. We note that to extend this base requirement with a new feature we must define a means of composing POTS with this feature, or, as illustrated in the diagram, use a previously defined mechanism. Unfortunately, for two different features there is no guarantee that we can use the same composition mechanism. Furthermore, for each composition we may require an additional restriction (called the composition invariant) on the way in which the parts are configured in order to gurantee that individual requirements are met.

Given such a composition technique we must now address the problem of integrating `Feature1` and `Feature2` in the same set of requirements. In figure 3, we see that an interaction occurs if the invariants introduced by the two features and/or the two composition mechanisms are contradictory. Properties are required to be preserved through the composition process; the multi-view approach allows us to integrate the view of invariants (using B) and the view of fairness (using TLA).

We note that there are many different ways in which we may wish to compose the three components. The four most obvious structures are:

- `Compose1(comp1(POTS,feature1), feature2)`, where we compose the `feature2` with the component which results from a composition between `POTS` and the `feature1`.

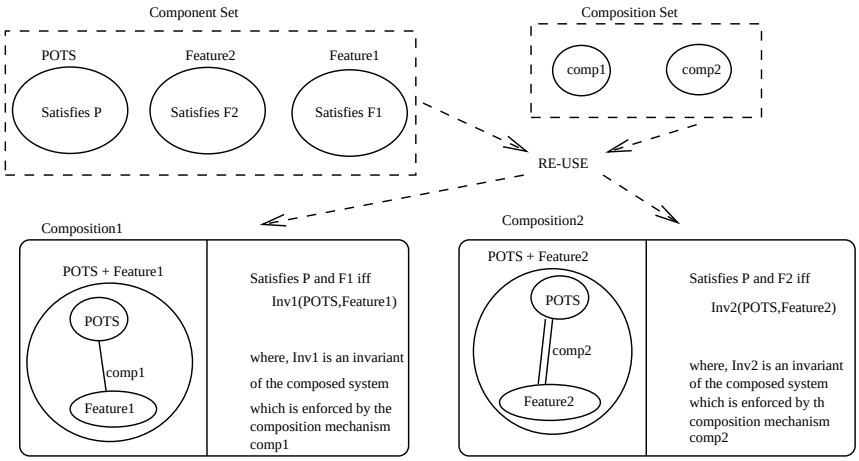

Fig. 2. Incrementing POTS

- `Compose2(comp2(POTS,feature2), feature1)`, where we compose the `feature1` with the component which results from a composition between `POTS` and the `feature2`.
- `Compose3(POTS, comp3(feature1,feature2))`, where we first compose the two features and then compose this new component with `POTS`.
- `Compose4(POTS, feature1, feature2)`, where we define a new composition mechanism which acts on all three components.

The feature composition problem is certainly difficult (even when there are only 2 features); now we argue that having formal requirements models makes it manageable, but we need to develop a methodology for composing features

4.1 Modelling services

A service is an extension of POTS, - the basic service - , providing functionality to the customer for interacting with the switch and the billing system. The modeling of services is based on the view of services as processes altering a set of *calls*. The current state of a service is characterized by an invariant over calls. A call is a structure that manages and describes the current parameters as the caller, the callee, the call state, the paying party ...However, a call may be extended into another call by operations over calls such as fusion, completion etc. This means that calls are central concepts in our modelling but this makes the modelling more flexible. More generally, a call is a structure recording the current participants, the connection, the state, the billing. We use the TLA$^+$ syntax for writing service specifications, as follows:

$COEF \triangleq 0..100$ used to define the percentage for contributing in the billing

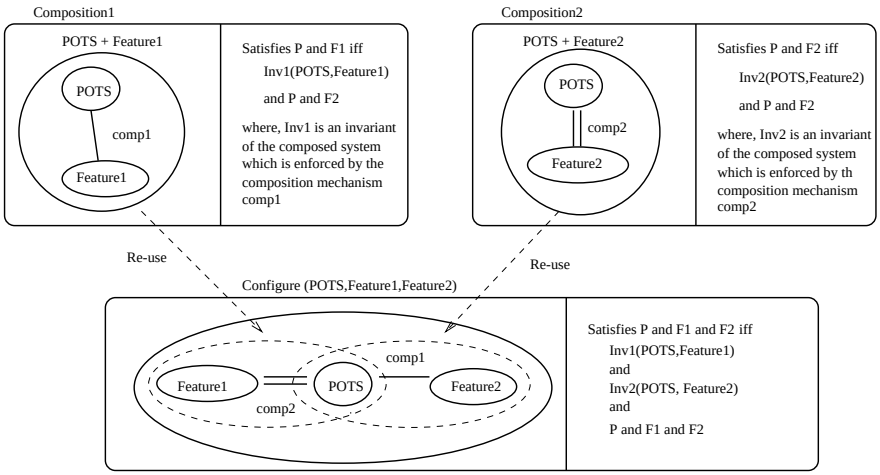

Fig. 3. Integrating Two Features

$$
\begin{array}{l}
CALLS \triangleq \\
\quad [party \ : \ \text{SUBSET} \ USERS, \\
\quad \ linkcall \ : \ \text{SUBSET} \ (USERS \times USERS), \\
\quad \ paycall \ : \ \text{SUBSET} \ \{USERS \times USERS \times USERS \times COEF \times TIME \times TIME) \\
\\
\quad \ com \ : \ \text{SUBSET} \ (USERS \times USERS), \\
\quad \ state \ : \ CALLSTATES]
\end{array}
$$

Variables such as calls, phones, tones, messages, billings, services are typed according to the following typing invariant. We define it and operations or actions which have to preserve it.

$$
\begin{array}{l}
Typing\_Variables\_Invariant \triangleq \\
\quad \wedge \ calls \ \in \ CALLS \\
\quad \wedge \ phones \ \in \ [USERS \ \rightarrow \ PHONESTATES] \\
\quad \wedge \ tones \ \in \ [USERS \ \rightarrow \ TONESTATES] \\
\quad \wedge \ messages \ \in \ [USERS \ \rightarrow \ \text{SUBSET} \ STRING] \\
\quad \wedge \ billings \ \in \ USERS \times COEF \times USERS \times COEF \times TIME \times TIME \\
\\
\quad \wedge \ services \ \in \ USERS \ \rightarrow \ \text{SUBSET} \ SERVICES
\end{array}
$$

Now, we can incrementally add new operations that are either activated by users or customers, or by the telecom systems. The basic service, called POTS, provides the following operations :

– off-hook

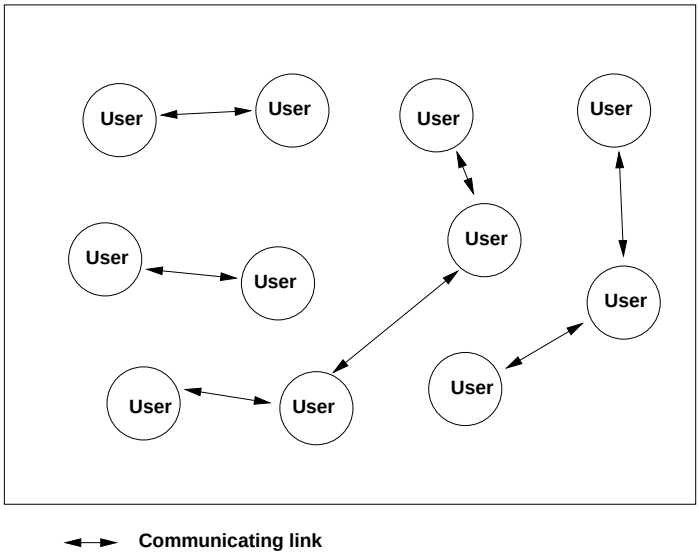

Fig. 4. View of services through calls

A user can off hook the phone because he/she wants to call somebody
somebody else is calling him/her. The switch will reply either by
sending a dialtone or by starting the communication.

$$
\begin{aligned}
&OFFHOOKCALLING(Xcaller) \triangleq \\
&\quad \wedge\ phones' = [phones\ !\text{EXCEPT}[Xcaller] = \text{"offhook"}] \\
&\quad \wedge\ tones' = [tones\ !\text{EXCEPT}[Xcaller] = \text{"notone"}] \\
&\quad \wedge\ \text{UNCHANGED} < tones, calls, messages, billings, t >
\end{aligned}
$$

Y wants to call somebody, namely X, in the call $Xcall$

$$
\begin{aligned}
&OFFHOOKRINGING(X, Y, Xcall) \triangleq \\
&\wedge\ Xcall \in calls \\
&\wedge\ \{X, Y\} \subseteq Xcall.party \\
&\wedge\ tones[X] = \text{"ringing"} \\
&\wedge\ tones[Y] = \text{"ringbacktone"} \\
&\wedge\ tones' = [[tones\ !\text{EXCEPT}[X] = \text{"notone"}]\ !\text{EXCEPT}[Y] = \text{"notone"}] \\
&\wedge\ phones[Xcalled] \neq \text{"offhook"} \Rightarrow phones' = [phones\ !\text{EXCEPT}[Xcaller] = \text{"offhook"}] \\
&\wedge\ phones[Xcalled] = \text{"offhook"} \Rightarrow phones' = phones \\
&\wedge\ \text{UNCHANGED} < calls, messages, billings, t, services >
\end{aligned}
$$

Xcalled is called by somebody else and Xcalled is ringing;
the operation is done by the user

- on-hook
- dial
- communication

We have added an event which is executed infinitly often to model the time, since we need to specify the starting point of a call and the ending point of a call, for instance.

$$TICTAC \triangleq \land\, t' = t + 1 \\ \land\, \text{UNCHANGED} < tones, calls, messages, billings, phones, services >$$

The global system, called POTS, is operationally defined as a disjunction of relations over primed and unprimed variables (thanks to TLA).

we define the set of possible events of the basic system, called
POTS

$$\begin{aligned} EventsBasicSystem \triangleq\ & \cup\, OffHookCallingEvents \\ & \cup\, OffHookRingingEvents \\ & \cup\, OnHookFirstEvents \\ & \cup\, UpdateCallsEvents \\ & \cup\, FinalUpdateCallEvents \\ & \cup\, OnHookLastEvents \\ & \cup\, DialEvents \\ & \cup\, SendingToneDialevents \\ & \cup\, DialToneEvents \\ & \cup\, CommunicationOkEvents \\ & \cup\, CommunicationDownEvents \\ & \cup\, CommunicationBusyEvents \\ & \cup\, OnHookDownEvents \\ & \cup\, OnHookBusyEvents \\ & \cup\, Clean\_Down\_CallsEvents \\ & \cup\, Clean\_Busy\_CallsEvents \\ & \cup\, Clean\_Completed\_Calls \\ & \cup\, \{TICTAC\} \end{aligned}$$

Now we apply the 'Next' operator to obtain the next relation
for the operational semantics.

$$NextBasicSystem \triangleq Next(EventsBasicSystem)$$

TLA$^+$ requires that we specify the variables of the system.

$VarsBasicSystem \triangleq < messages, calls, phones, tones, billings, t, services >$

Finally, we assume that every event is executed under the weak fairness assumption.

$FairnessBasicSystem \triangleq WF(VarsBasicSystem, EventsBasicSsystem)$

We have defined an operator assigning a formula from a set of formulae; it allows us to get a simpler way to specify, since we have to give the set of possible events and to apply it on the current set of events. Now, the bare basic service is simply specified by the following formulae.

$$
\begin{array}{l}
InitBasicSystem \triangleq \\
\quad \wedge\ calls = \{\} \\
\quad \wedge\ \forall p \in USERS : phones[p] = \text{"onhook"} \\
\quad \wedge\ \forall p \in USERS : tones[p] = \text{"notone"} \\
\quad \wedge\ billing = \{\} \\
\quad \wedge\ \forall p \in USERS : messages[p] = \text{""} \\
\quad \wedge\ \forall p \in USERS : services[p] = \{\text{"basic"}\} \\
\quad \wedge\ t = 0
\end{array}
$$

$$
\begin{array}{l}
SpecificationBasicSystem \triangleq \\
\quad \wedge\ InitBasicSystem \\
\quad \wedge\ \Box[NextBasicSystem]\_\{VarsBasicSystem\} \\
\quad \wedge\ FairnessBasicSystem
\end{array}
$$

The basic system provides the user with the basic functionality required for calling somebody else. At this stage, a user 'X' can call only one user 'Y'; if we increase the calling possibilities, we add functionality related to a new service. Increasing the basic functionalities means that we allow the user additional operations; if N is the relation characterizing the current service, then a new functionality is obtained by adding another relation, namely F, as follows: $N \vee F$. Composing is reduced to logical operations over relations on states, but we may have transformations to do on relations. The user view of the service is like a reactive system. The modules for POTS have a very restricted scope, since the functionality of each is very limited.

4.2 Adding a new service

The user's view deals with operations such as subscribing, unsubscribing, paying, billing, and a service is generally characterized by at least two operations that enable or disable the service, when the user has subscribed; for instance the service, called CCBS, allows the user/subscriber to be informed, when another, whom he is calling and busy, becomes idle.

$CCBS\_activation(X) \triangleq$
$\quad \wedge\ X \in USERS$
$\quad \wedge\ X \notin CCBS\_sub$
$\quad \wedge\ CCBS\_sub' = CCBS\_sub \cup \{X\}$
$\quad \wedge\ CCBS\_heap' = [x \in DOM\ CCBS\_heap \cup \{X\}$
$\qquad\qquad \mapsto \text{IF } x = X \text{ THEN } \{\} \text{ ELSE } CCBS\_heap[x]]$
$\quad \wedge\ \text{UNCHANGED } < list\_of\_unchanged\_variables >$

$CCBS\_inhibition(X) \triangleq$
$\quad \wedge\ X \in USERS$
$\quad \wedge\ X \in CCBS\_sub$
$\quad \wedge\ CCBS\_sub' = CCBS\_sub - \{X\}$
$\quad \wedge\ CCBS\_heap' = [x \in DOM\ CCBS\_heap - \{X\} \mapsto CCBS\_heap[x]]$
$\quad \wedge\ \text{UNCHANGED } < list\_of\_unchanged\_variables >$

We modify the basic service, by strengthening operations of the callee; moreover, CCBS is a very interesting service, since it requires the expression of a fairness constraint. A first step is to analyse what is shared by CCBS and POTS and what is private or local for CCBS. We introduce two variables that will manage the current subscribers of CCBS and the waiting users for re-calling somebody.

VARIABLES

$CCBS\_sub$,	set of users that have subscribed to CCBS
$CCBS\_heap$	function defining heaps

The typing invariant of CCBS declares the role of those variables.

$INVARIANT\_CCBS \triangleq \wedge\ CCBS\_sub \subseteq USERS$
$\qquad\qquad\qquad\qquad \wedge\ CCBS\_heap \in [USERS \to \text{SUBSET } USERS]$

The next step is to define "side-effects" on events of the basic service. CCBS requires an event for dequeing recalls for users having subscribed to CCBS; we call it $CCBS\_Dequeue(X, Y, Xcall)$, and it requires a fairness assumption.

$CCBS\_Dequeue(X, Y, Xcall)$
$\wedge\ tones[X] = \text{"notone"}$
$\wedge\ phones[X] = \text{"onhook"}$
$\wedge\ phones[Y] = \text{"onhook"}$
$\wedge\ tones[Y] = \text{"notone"}$
$\wedge\ X \in CCBS\_sub$
$\wedge\ tones' = [[tones\ !\text{EXCEPT}[X] = \text{"ringing"}]!\text{EXCEPT}[Y] = \text{"ringing"}]$
$\wedge\ Xcall.state = \text{"busyCCBS"}$
$\wedge\ Xcall \in calls \cap CCBS\_heap[Y]$
$\wedge\ \{X, Y\} \subseteq Xcall.party$
$\wedge\ \text{LET } newcall = \text{CHOOSE } c \,.\, \wedge\ c \in calls \cap CCBS\_heap[Y]$
$\qquad\qquad\qquad\qquad\qquad\qquad \wedge\ c.state = \text{"waiting"}$
$\qquad\qquad\qquad\qquad\qquad\qquad \wedge\ c.com = Xcall.com$

$$
\begin{array}{l}
\qquad\qquad\qquad\qquad \wedge\ c.paycall\ =\ Xcall.paycall \\
\qquad\qquad\qquad\qquad \wedge\ c.linkcall\ =\ Xcall.linkcall \\
\text{IN}\quad \wedge\ calls' = calls - \{Xcall\} \cup \{newcall\} \\
\qquad \wedge\ CCBS\_heap' = [CCBS\_heap\ !\text{EXCEPT}[Y] = @ - \{Xcall\}] \\
\wedge\ \text{UNCHANGED} < messages, phones, billings, calls, t, CCBS\_sub >
\end{array}
$$

Now, we modify two events in the specification of the basic service, namely the COMMUNICATION-BUSY, which manages calls when they are busy, and OFFHOOKRINGING, which manages when somebody is called and this phone is ringing. Hence, we modify events of the basic service and add new events.

$$
\begin{array}{l}
CBS\_COMMUNICATION\_BUSY(X, Y, Xcall) \triangleq \\
\wedge\ Xcall \in calls \\
\wedge\ Xcall.state = \text{"waiting"} \\
\wedge\ Xcall.party = \{X, Y\} \\
\wedge\ X \neq Y \\
\wedge\ \exists\ c \in calls\ : \\
\quad \wedge\ c \neq Xcall \\
\quad \wedge\ phones[Y] = \text{"offhook"} \\
\quad \wedge\ tones[Y] = \text{"talking"} \\
\quad \wedge\ Y \in c.party \\
\quad \wedge\ c.state = \text{"active"} \\
\quad \wedge\ X \notin c.party \\
\quad \wedge\ phones[X] = \text{"offhook"} \\
\quad \wedge\ tones[X] = \text{"dialling"} \\
\quad \wedge\ \text{LET}\ newcall \triangleq \\
\qquad\quad \text{CHOOSE}\ c\ .\ \wedge\ c \in CALLS \setminus calls \\
\qquad\qquad\qquad\qquad\quad \wedge\ c.state = \text{"busyCCBS"} \\
\qquad\qquad\qquad\qquad\quad \wedge\ c.party = \{X, Y\} \\
\qquad\qquad\qquad\qquad\quad \wedge\ c.com = \{\} \\
\qquad\qquad\qquad\qquad\quad \wedge\ c.paycall = \{\} \\
\qquad\qquad\qquad\qquad\quad \wedge\ c.linkcall = \{< X, Y >\} \\
\qquad \text{IN}\quad \wedge\ calls' = calls - \{Xcall\} \cup \{newcall\} \\
\qquad\quad \wedge\ CCBS\_heap' = [CCBS\_heap\ !\text{EXCEPT}[Y] = @ \cup \{newcall\}] \\
\qquad\quad \wedge\ tones' = [tones\ !\text{EXCEPT}[X] = \text{"CCBStone"}] \\
\wedge\ \text{UNCHANGED} < phones, messages, billings, t, CCBS\_sub >
\end{array}
$$

$$
\begin{array}{l}
CCBS\_OFFHOOKRINGING(X, Y, Xcall) \triangleq \\
\wedge\ Xcall \in calls \\
\wedge\ \{X, Y\} \subseteq Xcall.party \\
\wedge \vee\ tones[X] = \text{"ringing"} \\
\quad \vee\ tones[Y] = \text{"ringing"} \\
\wedge\ < X, Y > Xcall.linkcall \\
\wedge\ X \in CCBS\_sub \\
\wedge\ phones[X] \neq \text{"offhook"}\ \Rightarrow\ \wedge\ phones' = [phones\ !\text{EXCEPT}[X] = \text{"offhook"}]
\end{array}
$$

$$
\begin{array}{lll}
 & & \wedge\ tones' = [tones\ !\text{EXCEPT}[X] = \text{"notone"}] \\
\wedge\ phones[X] = \text{"offhook"} & \Rightarrow & \wedge\ phones' = [phones\ !\text{EXCEPT}[Y] = \text{"offhook"}] \\
 & & \wedge\ tones' = [tones\ !\text{EXCEPT}[Y] = \text{"notone"}]
\end{array}
$$

$\wedge$ UNCHANGED $< calls, messages, billings, t, CCBS\_sub, CCBS\_heap >$

Now, events of CCBS are defined as follows:

$$
\begin{array}{l}
CCBS\_events \triangleq \\
\quad \cup\ UNION_1(CCBS\_activation, USERS) \\
\quad \cup\ UNION_1(CCBS\_inhibition, USERS) \\
\quad \cup\ UNION_2(CCBS\_Dequeue, USERS, USERS, CALLS) \\
\quad \cup\ UNION_2(CCBS\_OFFHOOKRINGING, USERS, USERS, CALLS)
\end{array}
$$

However, POTS is modified by the service CCBS, by restricting COMMUNICATION events when the called user is busy; in fact, it leads to an enqueueing of the busy called user. We define a restriction of the POTS service which is modified and then we define a way to instantiate a system, defined by a set of events.

$$
\begin{array}{l}
CCBS\_Restriction(System\} \triangleq \\
\quad System - CommunicationBusyEvents \\
\quad \cup\ UNION_3(CCBS\_COMMUNICATION\_BUSY, USERS, USERS, CALLS)
\end{array}
$$

$$
CCBS\_instance(System) \triangleq CCBS\_Restriction(System\} \cup CCBS\_events
$$

Properties of CCBS tells us that when somebody (X) calls somebody else (Y) and, if Y is busy, then when Y is put onhook, the system will recall X and Y. X and Y will ring together, when fairness constraints are ensured.

$$
\begin{array}{l}
CCBSPlusBSEvents \triangleq CCBS\_instance(EventsBasicSystem) \\
SpecCCBSPlusBS \triangleq \\
\quad \wedge\ InitBasicSystem \\
\quad \wedge\ InitCCBS \\
\quad \wedge\ \Box\ [Next(CCBSPlusBSEvents)]\_ < VarsBasicSystem, VarsCCBS > \\
\quad \wedge\ WF(VarsCCBS \cup VarsBasicSystem, CCBSPlusBSEvents)
\end{array}
$$

THEOREM $SpecCCBSPlusBS \Rightarrow \Box\ INVARIANT\_CCBS$

---

if 'X' calls 'Y', while 'Y' is busy and 'X' has subscribed 'CCBS',

then eventually 'Y' is appended to the waiting heap for 'X'

THEOREM

$$
\begin{array}{l}
SpecCCBSPlusBS \Rightarrow \\
\quad (\Box\ (Calling(X, Y) \wedge (X \in CCBS\_sub \wedge Busy(Y)) \rightsquigarrow (X \in CCBS\_heap[Y]))
\end{array}
$$

if 'X' is in the waiting heap of 'Y', and if 'Y' has subscribed CCBS,

while 'X' is infinitly often busy, then eventually 'X' and 'Y' will

ring both

THEOREM

$$SpecCCBSPlusBS \Rightarrow$$
$$(X \in CCBS\_heap[Y] \wedge \Box\,(Y \in CCBS\_sub) \wedge \Box \Diamond \neg\, Busy(X) \rightsquigarrow (RingingBoth(X,Y))$$

We have expressed the formal modelling of the basic service and of CCBS; now, we have to verify theorems and to validate the specifications.

4.3 Coordinating views

Our model of services in TLA$^+$ can be verified and validated using the Atelier B toolkit. This means that we can verify invariants using a coding of our TLA specifications in B. Services can be viewed as abstract machines or as TLA$^+$ modules. The coordination of views means that properties that are observed in each model are not contradictory. Our model of services in TLA$^+$ can be verified and validated using the Atelier B toolkit, since our TLA$^+$ specifications are made up of imperative actions; these actions are written $x' = f(x)$ where $f(x)$ is an expression codable in B. Services in TLA$^+$ can be viewed as B abstract machines, but this leads us to forget fairness issues. However, it means that we do get a framework for animating and verifying the B view of a TLA$^+$ specification. It is clear that our approach is based on the use of a theorem prover but one can also use a model-checking-based tool.

4.4 Validation and Verification

We give a graphical representation of our formal models. The graphical syntax is informally explained and, where appropriate, we comment on how the formal meaning is captured using LOTOS, B and TLA. The semantics are clearly based on a state transition model and, as such, are easily communicated to the client through a process of animation.
We have specified a simple (POTS) client-oriented model of phone behaviour. This is sufficiently complex to illustrate the graphical syntax, in figure 5, being employed to communicate the formal semantics with the client.
The following aspects of the specification should be noted:
The header
The name of the class (`Phone`) being specified is given first in the header of the diagram. The other classes which are used in the specification of the new class are listed after the `USING` keyword: the `Phone` uses classes `signal` and `on-off`.

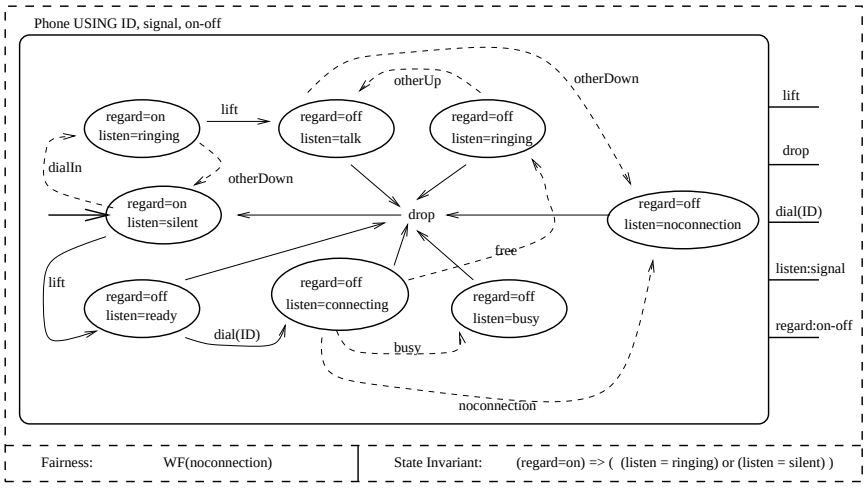

Fig. 5. The Phone

The interface

The interface to the class is represented by the connections at its boundary. Each connection corresponds to a service. In this case there are 5 services, namely: `lift, drop, dial, listen` and `regard`. `Lift, drop` and `dial` correspond to *transformer* services. When requested they result in a state transition. `Listen` and `regard` correspond to *accessor* services. When requested they return a value to the service requester. The type of the value returned is identified by a class name: `listen`, for example, returns a `signal` value. Services can be parameterised by a set of *input classes*: `dial`, for example, is parmeterised by an `ID` value. Services can be polymorphic on their input classes. In other words, a class can have two different services of the same name provided they can be distinguished by the types of their input parameters. The user of the class sees the class as a *black box*. The internal state of the class is encapsulated by its interface. The only access to state information is through the *accessors*. (There is one more type of service which is not illustrated by the `Phone`: the *dual* service is a combination of a *transformer* and an *accessor*: it returns a value *and* results in a state transition.)

Communication

Communication between an object *server* and its environment of *clients* is taken, unless otherwise specified, to be synchronous. This may lead to situations in which services are requested but are not enabled by the server. We note that *accessor* services are always enabled. *Duals* and *transformers* may not always be enabled: if a client requests a service which is not enabled then it is the client's responsibility to avoid a potential deadlock situation. One role of *fairness* in our models is to gurantee that services will be *eventually* enabled.

The operational semantics
There are eight states in the `Phone` class. Thus every `Phone` instance (object) must be in one of these eight states. These states are represented as nodes in the inside of the class boundary. For each state, each of the *accessor* values must be defined. To aid compositional specification techniques, and to facilitate the specification of classes with large (potentially infinite) numbers of states, we can define a class to be *structured* as a set of *component* classes. Then, these *internal* state values can be used to define the *external accessor* values. This provides a degree of implementation freedom and emphasises that *internal* details are hidden to the outside. In the `Phone` example, there is no *structure* definition as the number of states is manageable without one. The initial state of an object on creation is specified by a bold pointer which does not originate from another state. Hence, a `Phone` always starts `on` and `silent`. The state transitions which occur in response to an external service request are represented by solid pointers from old to new states.

Invariants
State invariant properties define restrictions on the possible sets of component values. For example, as is shown in figure 5, we may require that when `on`hook the `Phone` *must* be `ringing` or `silent`. These properties are verified, for more complex cases, using B: by checking that all transitions are closed with respect to the invariant it is not necessary to examine every single reachable state (which we can do directly with the simple `Phone` model). Note that the state invariants specified in this way are explicit requirements of the client that must be respected by the model. A specification where the invariants are not true is said to be *contradictory*.

Nondeterminism
Nondeterminism is formalised as internal state transitions that may occur independent from external service requests. These are represented by (possibly labelled) dotted pointers from old to new states. For example, when `off` and `connecting` the `Phone` user has no control over whether the number they are trying is `busy, free` or if `noconnection` is possible. These three cases are specified using internal actions (labelled appropriately). The difference between internal and external actions specifies a *point-of-view* onto a class (and the objects of the class). In this paper, our models specify the `Phone` user's point of view (or abstraction). The way in which the telephone network interacts with the `Phone` is abstracted away from in the form of nondeterministic transitions. Certainly, it is necessary to specify other points of view when modelling the whole telephone network. Our modular approach lets us work with different abstractions and then helps us to integrate these abstractions into a complete specification. This is beyond the scope of this paper, which concentrates on user requirements.

Fairness
Liveness conditions can be specified on the nondeterministic events in the model. For example, we may require that when `off` and `connecting` the user does not wait forever for a state transition if they refuse to drop the phone. This must

be specified in a separate TLA (temporal) clause. In figure 5, we specify *weak fairness* on the `noconnection` action.

(In)finite processes

A `Phone` is an *infinite* process. In later examples we specify finite behaviours which `EXIT` after some specific behaviour is fulfilled. A `Phone` is said to be of type `NOEXIT`.

A new feature: black list

The `Black List` feature has a similar function to originating call screening, but restricts incoming rather than outgoing calls. The idea is that you can store a list of numbers that you know you do not wish to talk with and then your phone does not ring when such numbers are the source of an incoming call. Our specification of this feature is illustrated in figure 6.

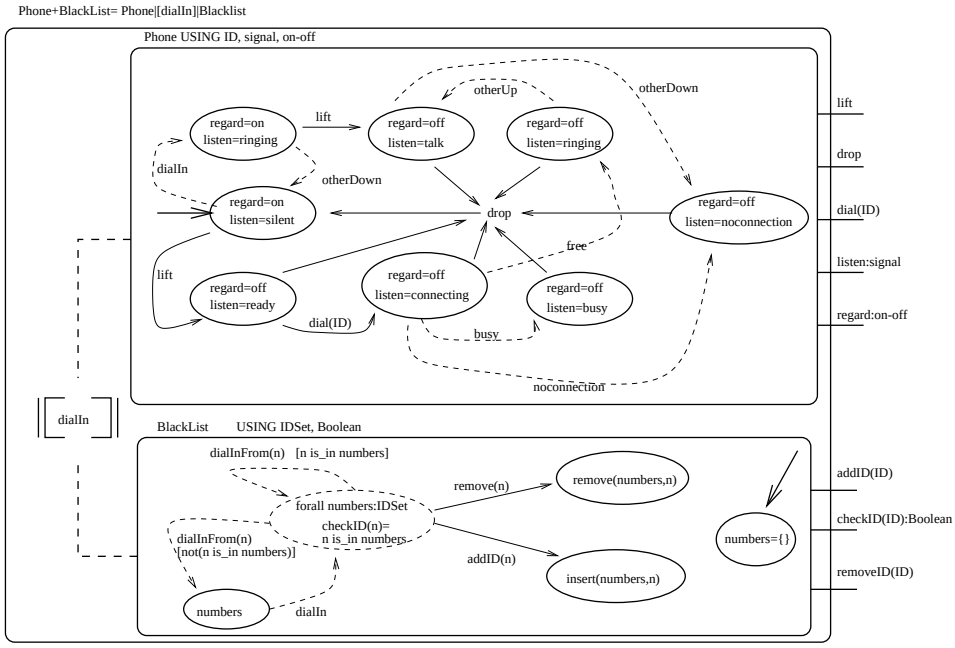

Fig. 6. Black list

Again, we have some comments to make with regard to this feature model:

Composition Re-Use

The composition is precisely that seen for a similar, better known, `CallerID` feature: there is internal synchronisation on the `dialIn` event and the system depends on an action refinement in the network to carry the new identification data using a `dialInFrom` action.

Phone refinement
Unlike `CallID`, not all `dialInFrom` actions result in a `dialIn` action: the blacklist filters out all incoming dials which are stored in the list of numbers in its state. However, like `CallID`, from the point of view of the user the new system is a refinement of the old phone — the only difference is the resolution of some of the nondeterminism in the original phone model.
Weak fairness guarantees eventuality
We require *weak fairness* on the `dialIn` event in the `BlackList` component. In the `BlackList` component we see that after a `dialInFrom` event, the external services `removeID` and `addID` may not be enabled until a `dialIn` action is performed, in the case where the number is not black listed. However, weak fairness on `dialIn` guarantees that this transition will eventually occur. Thus, we guarantee that the telephone user will not be deadlocked if they wish to `add` or `remove` a number from the blacklist because of an incoming call.
Localisation
At first glance, this feature seems to be *local*. All other users of the telephone system can remain unaware of this particular feature at any given phone. However, we have abstracted away from an implementation detail which has *global* effect: what signal should a caller hear if they telephone someone who has black listed them? There are a number of choices:

- A new type of signal telling them that they have been blacklisted.
  This may not be acceptable from a social point of view — do you really want someone to know that you don't wish to talk to them.
- A noconnected signal.
  This may not be acceptable since the caller may misinterpret the signal as saying that the number they are dialling is impossible to connect. Furthermore, an intelligent user may realise *why* they are unconnected, which brings us back to the first problem.
- A busy signal.
  This may be unacceptable since the caller may continue dialling because they think the person they are trying to contact will be available as soon as their current call is completed.
- A ringing signal.
  This seems to be the most acceptable choice, and in our network model we specified the feature in this way. Thus the blacklist service required only *local* change to the telephone user which requested this service. All other users retain their original behaviour.

It is only through animation that a user can be expected to understand such choices and help the designers to resolve the nondeterminism.

5 Conclusion

The problem of telephone feature interaction is just a particular instance of a general problem in software engineering. The same problem occurs when we consider inheritance in object oriented systems, sharing data in distributed systems,

multi-way synchronisation in systems of concurrent processes, etc .... However, the problem is particularly difficult in telephone systems because features are the increments of development.
We have shown the importance of re-usable composition mechanisms. Although our work is targeted towards the client during requirements capture, we believe that the same models could be used during design and at the network level. We support the principle of developing re-usable analysis techniques based on re-usable synthesis mechanisms. The object oriented approach can be extended to include a classification of feature types and we hope to map this onto a formal algebra for feature development.
We have used a graphical notation for communicating with the customer. However, our graphics are based on formal notations of languages, which may be difficult for the customer to understand. This work was very helpful in studying the complementary nature of different formalims. Logical formalisms such as B or TLA are really suitable for logical analysis of services based on proof techniques. Animation is made easier by automata-based representations.
This work is dependent on the different view points and the different semantic models. The integration of these semantics and the development of user-oriented tools is the most important element of our current, and future work. Finally, the integration of refinement-based reasoning is an important point to develop, with experiments in other domains.

References

1. J.-R. Abrial. *The B book - Assigning Programs to Meanings.* Cambridge University Press, 1996.
2. J. Blom. Formalisation of requirements with emphasis on feature interaction detection. In *Feature Interactions In Telecommunications IV*, Montreal, Canada, June 1997. IOS Press.
3. J. Blom, B. Johnsson, and L. Kempe. Automatic detection of feature interactions in temporal logic. In K. E. Cheng and T. Ohta, editors, *Feature Interactions in Telecommunications Systems*, pages 1–19. IOS Press, 1996. [9].
4. J. Blom, B. Jonsson, and L. Kempe. Using temporal logic for modular specification of telephone services. In L. G. Bouma and H. Velthuijsen, editors, *Feature Interactions in Telecommunications Systems*, pages 197–216. IOS Press, 1994.
5. G. Booch. *Object oriented design with applications.* Benjamin Cummings, 1991.
6. L. G. Bouma and H. Velthuijsen, editors. *Feature Interactions in Telecommunications Systems.* IOS Press, 1994.
7. R. Boumezbeur and L. Logrippo. Specifying telephone systems in LOTOS. *IEEE Communications Magazine*, 31(8):38–45, 1993.
8. Ed. Brinksma, Giuseppe Scollo, and Chris Steenbergen. LOTOS specifications, their implementation and their tests. In *Sixth International Symposium on Protocol Testing, Specification and Verification*, Montreal, June 1986.
9. K. E. Cheng and T. Ohta, editors. *Feature Interactions in Telecommunications Systems.* IOS Press, 1996.
10. P. Coad and E. Yourdon. *Object oriented design.* Prentice-Hall (Yourdon Press), 1990.

11. P. Combes and S. Pickin. Formalisation of a user view of network and services for feature interaction detection. In L. G. Bouma and H. Velthuijsen, editors, *Feature Interactions in Telecommunications Software System*, pages 120–135. IOS Press, 1994. [6].
12. L. Constantine. Beyond the madness of methods: System structure methods and converging design. In *Software Development 1989*. Miller-Freeman, 1989.
13. Dan Craigen, Susan Gerhart, and Ted Ralston. An international survey of industrial applications of formal methods. Nistgcr 93/626, U.S. Department of Commerce, Technology Administration, National Institute of Standards and Technology, Computer Systems Lab., Gaithersburg, MD 20899, 1993.
14. Geoff Cutts. *Structured system analysis and design method.* Blackwell Scientific Publishers, 1991.
15. P. Dini, R. Boutaba, and L. Logrippo, editors. *Feature Interactions in Telecommunications Newtworks IV*, Montreal, 1997. IOS Press.
16. H. Ehrig and Mahr B. *Fundamentals of Algebraic Specification I.* Springer-Verlag, Berlin, 1985. EATCS Monographs on Theoretical Computer Science (6).
17. M. Faci and L. Logrippo. Specifying features and analysing their interactions in a lotos environment. In L. G. Bouma and H. Velthuijsen, editors, *Feature Interactions in Telecommunications Software System*, pages 136–151. IOS Press, 1994. [6].
18. M. Faci, L. Logrippo, and B. Stepien. Formal specification of telephone systems in lotos : constraint-oriented style approach. *Computer Networks and ISDN Systems*, 21:53–67, 1991.
19. A. Gammelgaard and J. E. Kristensen. Interaction detection, a logical approach. In L. G. Bouma and H. Velthuijsen, editors, *Feature Interactions in Telecommunications Systems*, pages 178–196. IOS Press, 1994.
20. J.-P. Gibson. *Formal Object Oriented Development of Software Systems Using LOTOS.* Tech. report CSM-114, Stirling University, August 1993.
21. J.-P. Gibson. Formal object based design in LOTOS. Tr-113, University of Stirling, Computing Science Department, Stirling, Scotland, 1994.
22. J.-P. Gibson. Feature Requirements Models: Understanding Interactions. In *Feature Interaction Workshop 1997, Montreal, Canada*, Feature Interaction Workshop. IOS Press, June 1997.
23. J.-P. Gibson, B. Mermet, and D. Méry. Feature interactions: A mixed semantic model approach. In Gerard O'Regan and Sharon Flynn, editors, *1st Irish Workshop on Formal Methods*, Dublin, Ireland, July 1997. Irish Formal Methods Special Interest Group (IFMSIG), Springer Verlag. http://ewic.springer.co.uk/.
24. J.-P. Gibson, B. Mermet, D. Méry, and Y. Mokhtari. Spécification de services dans une logique temporelle compositionnelle. Rapport de fin du lot1 du marché n^{o}96 1B CNET-CNRS-CRIN, Centre de Recherche en Informatique de Nancy, décembre 1996.
25. J.-P. Gibson and D. Méry. A unifying framework for multi-semantic software development. In Max Mühlhäuser, editor, *Special Issues in Object-Oriented Programming.* Dpunkt, 1997.
26. C.A.R Hoare. *Communicating Sequential Processes.* Prentice-Hall International, 1985.
27. IEE. *Special Collection On Requirements Analysis.* IEE Transactions on Software Engineering, 1977.
28. ISO. LOTOS — a formal description technique based on the temporal ordering of observed behaviour. Technical report, International Organisation for Standardisation IS 8807, 1988.

29. B. Kelly, M. Crowther, and J. King. Feature interaction detection using sdl models. In *GLOBECOM. Communications: The Global Bridge. Conference Record*, pages 1857–61. IEEE, 1994.
30. B. Kelly, M. Crowther, J. King, R. Masson, and J. Delapeyre. Service validation and testing. In K. E. Cheng and T. Ohta, editors, *Feature Interactions in Telecommunications Systems*, pages 173–184. IOS Press, 1996. [9].
31. L. Lamport. A temporal logic of actions. *Transactions On Programming Languages and Systems*, 16(3):872–923, May 1994.
32. B. Liskov and Zilles S. Programming with abstract data types. In *ACM SIGPLAN Notices*, volume 9, pages 50–59, 1974.
33. B. Mermet and D. Méry. Incremental specification of telecommunication services. In M. Hinchey, editor, *First IEEE International Conference on Formal Engineering Methods (ICFEM)*, Hiroshima, November 1997. IEEE.
34. B. Mermet and D. Méry. Safe combinations of services using b. In John McDermid, editor, *SAFECOMP97 The 16th International Conference on Computer Safety, Reliability and Security*, York, September 1997. Springer Verlag.
35. D. Méry. Requirements for a temporal B : Assigning Temporal Meaning to Abstract Machines ... and to Abstract Systems. In A. Galloway and K. Taguchi, editors, *IFM'99 Integrated Formal Methods 1999*, Workshop In Computing Science, YORK, June 1999.
36. C. A. Middleburg. A simple language for expressing properties of telecommunications services and features. Technical report PU-94-356, KPN Research, Network and Service Control department, 1994.
37. R. Milner. *A Calculus of Communicating Systems.* Springer-Verlag, 1980.
38. D. T. Ross. Structured analysis (SA): A language for communicating ideas. In *IEE Transactions on Software Engineering.* IEE, 1977.
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. Kenneth J. Turner. *Using Formal Description Techniques - An Introduction to ESTELLE, LOTOS and SDL.* John Wiley, New York, January 1993.
41. K.J. Turner. SPLICE I: Specification using LOTOS for an interactive customer environment — phase 1. University of Stirling SPLICE Internal Technical Document, 1992.
42. K.J.T. Turner. *Using FDTS: An Introduction To ESTELLE, LOTOS and SDL.* John Wiley and Sons, 1993.
43. P. Zave. Feature interactions and formal specifications in telecommunications. *Computer*, August 1993.
44. Pamela Zave. The operational versus the conventional approach to software development. *Comm. ACM*, 27:104–118, 1984.
45. Pamela Zave. Feature interactions and formal specifications in telecommunications. *IEEE Computer Magazine*, pages 18–23, August 1993.