Towards a Feature Interaction Algebra

J. Paul Gibson, *Université Henri Poincaré (Nancy I),*
LORIA UMR 7503 CNRS, Campus scientifique, B.P. 239
54506 Vandoeuvre-ls-Nancy Cedex, France*,
gibson@loria.fr

Abstract.

The composition (and configuration) of requirements is particularly important in feature specification because the units of incrementation in system development are themselves features. Thus we have requirements models made up of a large number of components, each of which is easy to specify and validate individually, but whose complexity resides in the semantics of composition, and configuration.

We approach the definition of feature composition from the point of view of the client. Through our study of CS-1, we identify different ways in which the client would wish to compose their features with the plain old telephone service (POTS). From this we motivate the development of a feature composition algebra, the foundation of a feature interaction algebra. Quite simply, we hope to be able to perform a meta-analysis of the feature interaction problem using the feature classes rather than the features themselves. In this paper we show the type of meta-analysis that can lead to an algebraic formulation of feature composition and configuration.

1 Overview

In this paper, we examine the composition of telephone feature requirements. Clients often have a compositional understanding of their needs, and requirements are continually evolving in a compositional manner. The key issue is re-use of what has already been defined, together with an integration of what is now required. Problems occur when contradictory requirements are defined. The detection and resolution of these contradictions is important in the analysis and requirements modeling of *feature interactions* (see [8]). In an ideal world, the requirements modellers would be restricted to using a *finite* set of well defined composition mechanisms, and each of these would have a corresponding contradiction *detection and resolution method.* Unfortunately, users always demand new ways of composing their requirements and no such method exists.

We restrict our attention to a set of telephone features as defined by a subset of `Capability Set 1` (CS1) of the `ITU-T`. *Feature interactions* arising out of contradictory requirements are common even in this small subset of telphone capabilities. Through our study of CS-1, we identify different ways in which the client would wish to compose their features with the plain old telephone service (*POTS*). This will form the basis from which we go on to develop a feature composition algebra. Our goal is to have an *algebra of interactions.* Every feature will be classified in a number of fundamental ways, including the way in which it is specified to be composed with POTS, and through this classification we can: create a class hierarchy of features, identify interactions by the types/classes of the features rather than by analysing the features themselves, define re-usable composition mechanisms which can be applied to any given set of classified features, and define re-usable methods of interaction resolution, based on our composition mechanisms.

2 Structure in Object Oriented Feature Specifications

Structure is fundamental to all stages of system development: it provides the framework

upon which already developed parts of a system can be re-used. Structured analysis and requirements capture methods have been successfully applied in many different problem domains during the last twenty years[5, 7]. The figure below illustrates, for a simple example, the symbiotic relationship between structure and re-use: *classification* facilitates re-use of abstractions and relations between abstract behaviours, *composition* facilitates re-use of concrete behaviour, *refinement* facilitates re-use of verification and validation, *configuration* facilitates re-use of composition mechanisms.

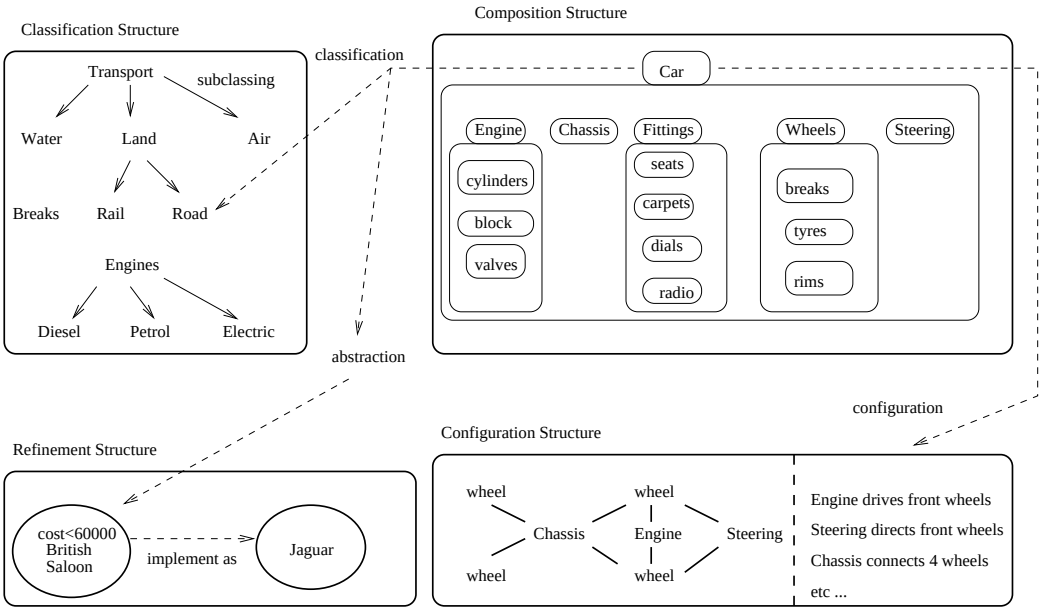

Figure 1: Structure and Re-Use

2.1 Composing Feature Requirements

In this paper we focus on the composition and configuration issues when developing new telephone features. In particular, we examine the different ways in which telephone users may wish to configure their sets of requirements. A *divide-and-conquer* approach to telephone feature development, based on the *has-a* relationship, can be summarised as follows: *If we wish to understand a telephone system then we must first understand the component features, and then understand the way in which the components can be configured.* [8] has examined the means by which feature requirements models can be built and validated by the client. This paper is concerned with the configuration issues. We show that the method by which we compose each feature with the standard telephone behaviour can help us to identify both the means by which features can be configured and the potential interactions between features. There are always different ways of understanding behaviour and such understanding leads to different sets of components. However, in the telephone model of requirements it is quite natural to view the system as a set of features. Given a set of feature requirements then *how* do we put them together into one requirements model? Different users will require their features to work together in different ways — defining different priorities between features, for example. Configuration is thus central to the feature interaction problem.

2.2 Object Oriented Structuring: some motivation

Object oriented concepts were conceived in Simula [19], went through infancy in Smalltalk [13, 14] and could be said to be leaving adolescence, and approaching maturity, in the form of many different languages[6, 22, 18, 21] and methods[20, 4, 18, 2]. In each of these models, we see different forms of structuring which correspond to the ideas expressed in the car example. In this paper we are concerned with composition and configuration and so we could say that

our presentation is object based[23].

(1) Simple Composition

Compositional structure is fundamental to object oriented analysis. A simple *container* view of composition enforces the notion of state encapsulation. The state of an object is defined by the state of all its components. An object's state is encapsulated if it can be changed only through the services offered at its external interface. Consequently, the state of its components can be changed only by the object itself. Composition is a relationship between a container object and its contained parts: it tells us nothing about the relationship between the parts.

(2) Configuration

If `A` and `B` are components of object `C` then we say that `A` and `B` are configured (in `C`) if `C` offers a service which requires the use of both `A` and `B`. In the car, for example, it is clear that the radio and the engine are components of the same object but they are not usually configured! Configurations lead us to define invariant properties between components that must be true for the containing system to behave correctly. In other words, invariants are part of the formal glue for putting requirements components together.

2.3 Features, Interactions and Incremental Development

In figure 2, we consider POTS as one requirements 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.

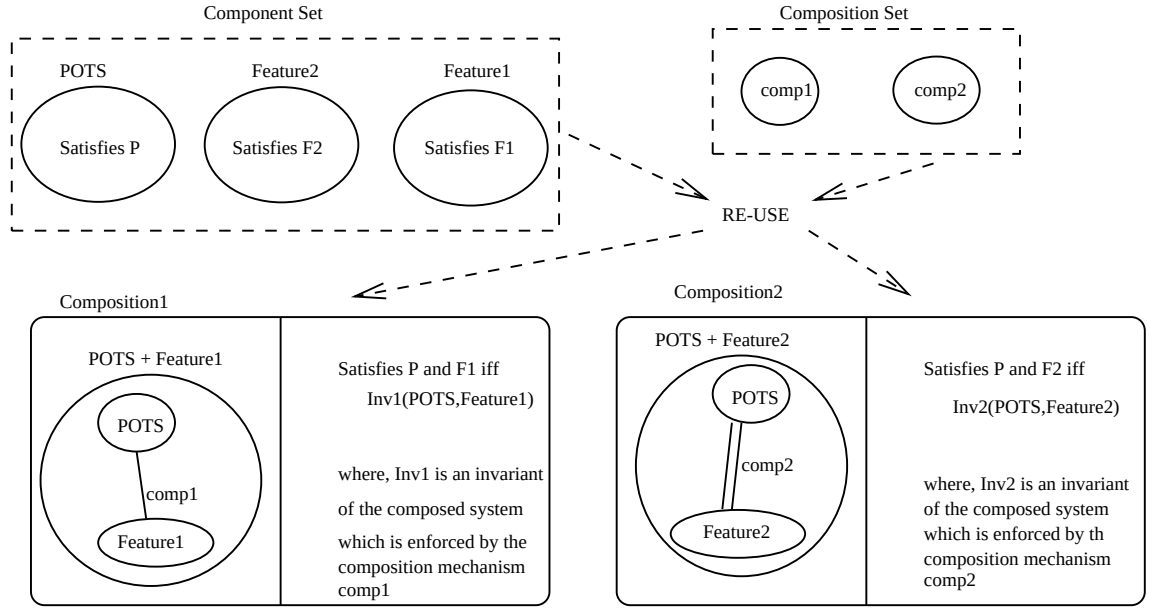

Figure 2: Incrementing POTS

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. In section 4.5 we see that this is one type of interaction which can be detected automatically through using our analysis method.

2.4 The Mixed Semantic (Composition) Approach

We have shown the need for a mixed semantic model when specifying telephone feature

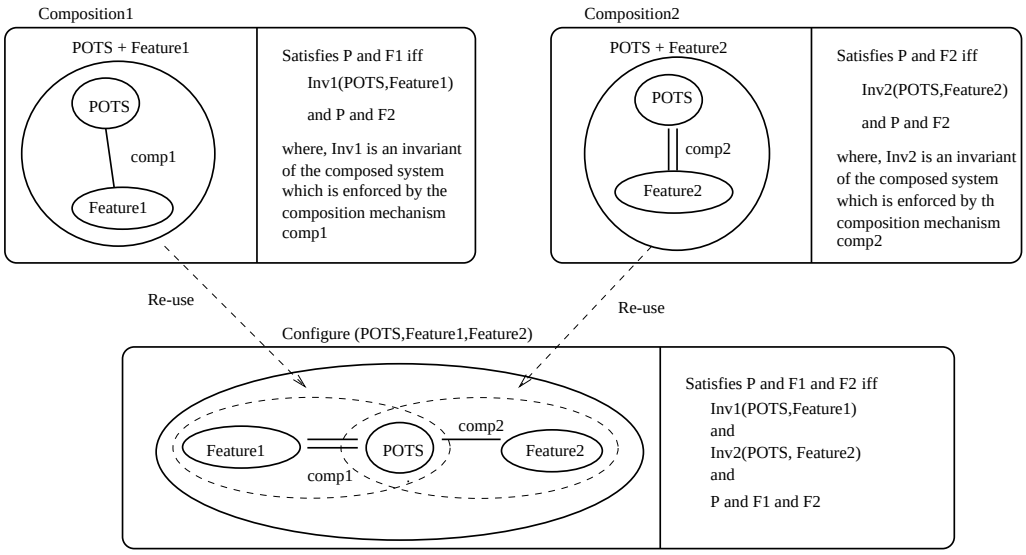

Figure 3: Integrating Two Features

requirements [10]. 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 [9].

- 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 [17].

- 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[11].

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. In section 4.5 we show how our analysis method uses each of these different points of view:

(1) Object oriented composition in LOTOS:
LOTOS [3] is made up from an abstract data type part [16], and a process algebra part [15]. 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 invariant is interpreted as an interaction and is in fact an interference

between operations: it is a way to detect interaction among services specified as abstract machines.

(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, the relation over variables, and the fairness constraints. 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[12].

(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. This paper reports on our attempt to identify such concepts.

3 Telephone Feature Composition: an algebraic *ideal*

Before we introduce our first steps towards developing a feature interaction algebra, we motivate our work by illustrating, in the figure below, an *ideal* development environment based on a composition algebra.

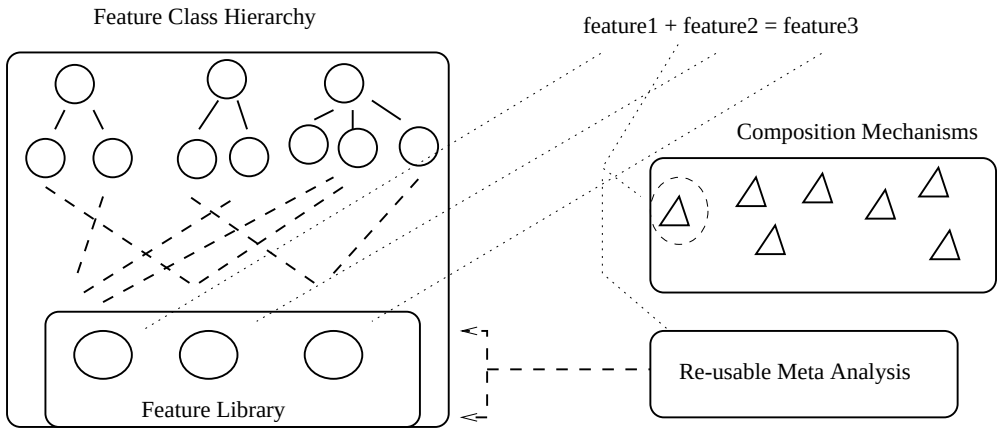

Figure 4: An Ideal Environment

The most important part of our *ideal* environment is a library of features which have already been formally specified and validated. This library will be structured in a class hierarchy where each feature will have many abstract superclasses. The second most important part of our environment is a library of (formally specified) feature composition mechanisms which will operate on a subset of features (depending, of course, on their classification. A re-usable (meta-analysis) will have already identified which classes of features interact and, where possible, will also provide re-usable resolution mechanisms. Creating a new feature will usually require the client combining already existing features in pre-defined ways, resulting in a new feature whose classification is calculated automatically following our algebraic semantics. Some features will require the specification of new concepts within the problem domain

and, as such, cannot be developed using already existing features. These new features will often *fit directly* into our already existing class hierarchy. In such a case, the meta-analysis does not need to be further extended. In the worse case, new abstract classifications (and composition mechanisms) will be needed and hence the analysis will need to be done *from scratch*. Furthermore, the requirements modellers will be responsible for placing this new case within the formal algebra.

4 Telephone Feature Composition: some analysis

In this we give a graphical representation of our formal feature requoirements models[1]. 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, although they are not shown in this paper, we have a means of mapping these specifications onto more client-friendly graphical representations. The graphical specifications which are given correspond to the default diagrams which are produced when no client mappings exist.

4.1 POTS Specification

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.

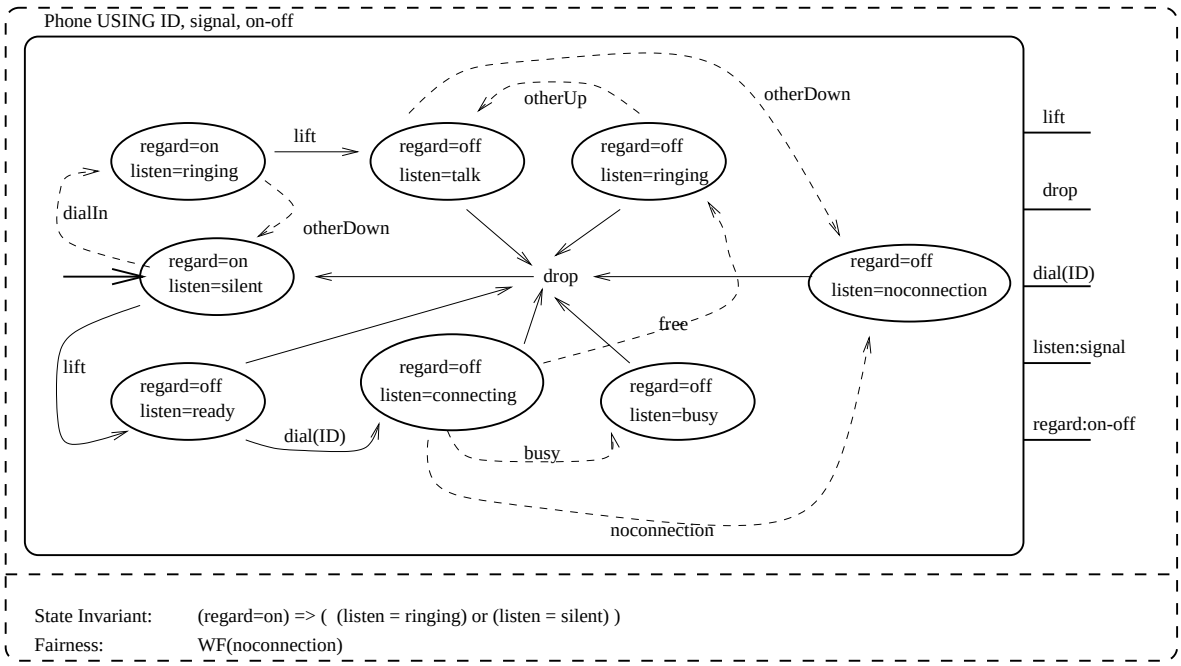

Figure 5: The Phone

4.2 Caller Identification

The first feature which we consider is `caller identification`. This feature is one of the simplest to understand (and specify). In CS1, this feature is known as *Call Number Delivery* (CND). The informal description is as follows:

> CND requires a special telephone that can show the caller's number before a call is answered. The user might decide not to answer a particular caller when the

[1]In fact, due to lack of space, we are restricted to 3 of the simplest features — a more comprehensive list can be found at our web page `http://www.loria.fr/ gibson/features.html`

telephone rings.

The way in which we define a caller identification component and compose this with the telephone component may not be so obvious. There are many different ways in which we could choose to provide call identification to the user For example, we could change the `regard` or `signal` services to provide identification information when in the state `on` and `ringing`, or define a new accessor for the `Phone` which provides identification information (for all `Phone` states, not just when `on` and `ringing`), or define a new state component within the `Phone` specification, or ... We can see that, even with such a simple feature, it is important to be precise about the actual requirements of the user. The behaviour which we finally chose to specify is illustrated in figure 6. Each of the classes is specified by an abstract data type in LOTOS. Then these data types are *wrapped up* into LOTOS processes and their interaction is defined using the process algebra operators. In this example, as with the ones which follow, we leave out the specification of fairness and invariants, and mention them only when necessary.

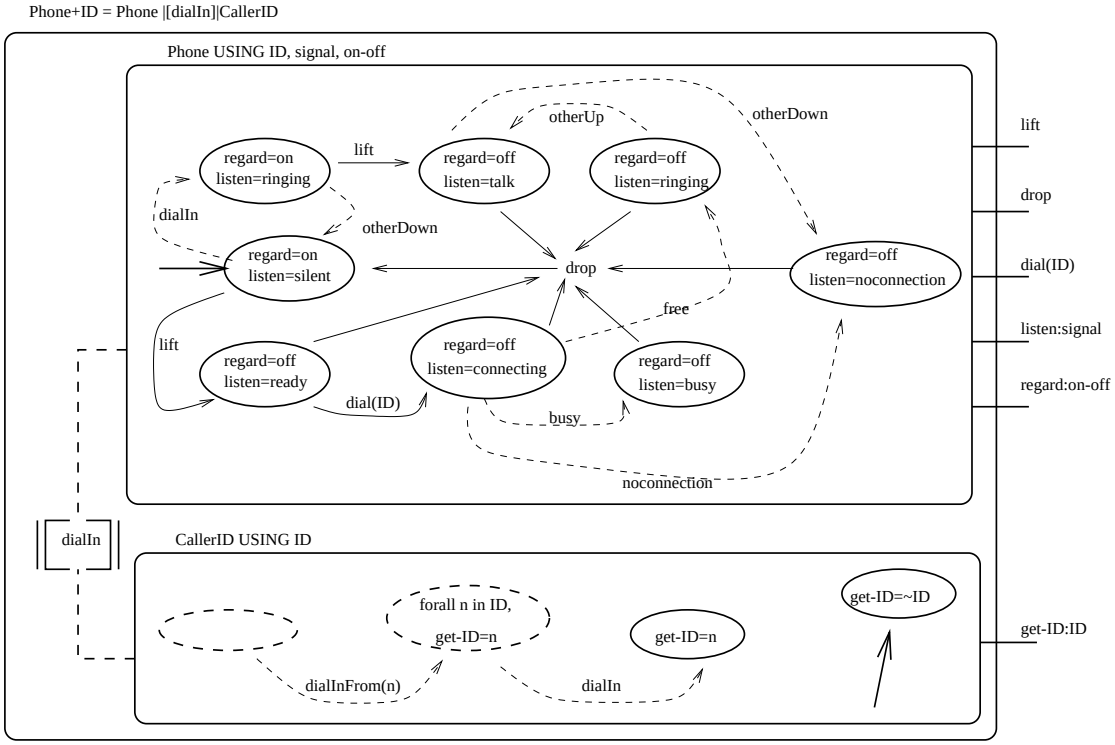

Figure 6: Caller Identification

The following aspects of the specification should be noted:

Compositional Re-Use

The original `Phone` specification is unchanged: we have real compositional re-use where the behaviour of the `Phone` is encapsulated behind it's interface. There is no need to re-validate this component.

Interface Union

All the services offered by the `Phone` are also offered by the new system, defined by the `Phone+ID` class. This is represented graphically by directly connecting the interface of the component to the the interface of its container. The new system also offers a `get-id` accessor, which is is identical to the accessor of the same name as offered by the `CallerID` component. We say that the new system interface is a union of its components' interfaces.

Internal Action Refinement

The `CallerID` component acts as a filter to the telephone network by synchronising on a `dialInFrom` action, which carries the identification of the caller as a parameter, and storing

this identification for the user to access. It then synchronises on a `dialIn` event with the `Phone`. At the network level we have an *action refinement* (and syntactic renaming) on the `dialIn` event which now, as `dialInFrom`, carries additional information needed by the `CallerID` feature.

Internal Synchronisation

Synchronisation between the `Phone` and the `CallerID` feature occurs on the internal `dialIn` event. This is represented by a dotted graphical line between the two components, labelled with the event name. The type of synchronisation is defined to be *strict*, using the standard `|[actionname]|` notation as seen in LOTOS. A *strict* synchronisation is one in which both components must perform the state transition at the same time, otherwise the action cannot be carried out.

Local Fairness Requirements

We require *weak fairness* on the `dialIn` action in the `CallerID` component. This guarantees that after a `dialInFrom` action we do not risk waiting forever for the `dialIn` to occur. This does not imply *weak fairness* on the `dialIn` in the telephone since this would state that if we waited long enough we would always have an incoming call!

The CallerID Component

In the initial state the `get-ID` accessor is undefined. This is represented by the *exception* value `Ĩ D`. The `CallerID` has as many states as there are possible identification `ID` values. Thus, this may not be a finite state machine. A parameterised state (represented by a dotted node) can be used to define sets of state transitions and/or accessor values. Such a set is defined by some state conditions which must be true for the transitions to be valid. When the node is empty there are no such conditions and the transitions are valid *forall* states of the class. We should interpret the diagram as saying that: 'forall states of `CallerID` when a `dialInFrom(n)` action occurs then the new value of `get-ID` is set to `n`'; and: 'forall states a `dialIn` action can occur and it does not change the value of the `get-ID` accessor'.

Localisation

We note that the addition of this new feature can be said to be *local* since it has no effect on the other users of the telephone system.

Extension: A Phone refinement

The new system is said to extend the old phone behaviour: it offers exactly the same behaviour *plus* a new service. This is a form of refinement. A phone user will be completely unaware if a phone is replaced by a phone extension. However, this is not true of the network which is connected to the phone. It must be changed to take into account the action refinement on `dialIn`.

Similar Features: A classification

One of the main goals of this work is to classify features by the way in which they must be composed with the telephone network. The form of composition seen with `CallerID` is also used for two other features which we have developed, namely `CallInHistory` and `BlackList`. These three features interact in the same way when composed with other features: thus we can re-use the same interaction analysis for each of the three requirements models.

4.3 Originating Call Screening

Originating Call Screening (OCS) allows restriction on the types of calls that a user can make. The informal description is as follows:

> OCS prevents calls to certain telephone numbers or area codes. Parents, for example, might wish to stop their children from calling premium rate numbers. In the extreme case, all outgoing calls (except perhaps to the emergency services) may be disabled: the network operator might do this if a telephone bill remains unpaid.

For now, we do not consider *who* is making such a restriction. We require a store for a list of numbers which are to be screened. The simplest functionality is to provide a means of adding a number to the list and removing a number from the list. (More complex means

of updating the list can be built upon these two services.) Then, when the user dials a number, a connection is not made if the number requested is on the restricted list. The formal requirements model is illustrated in figure 7.

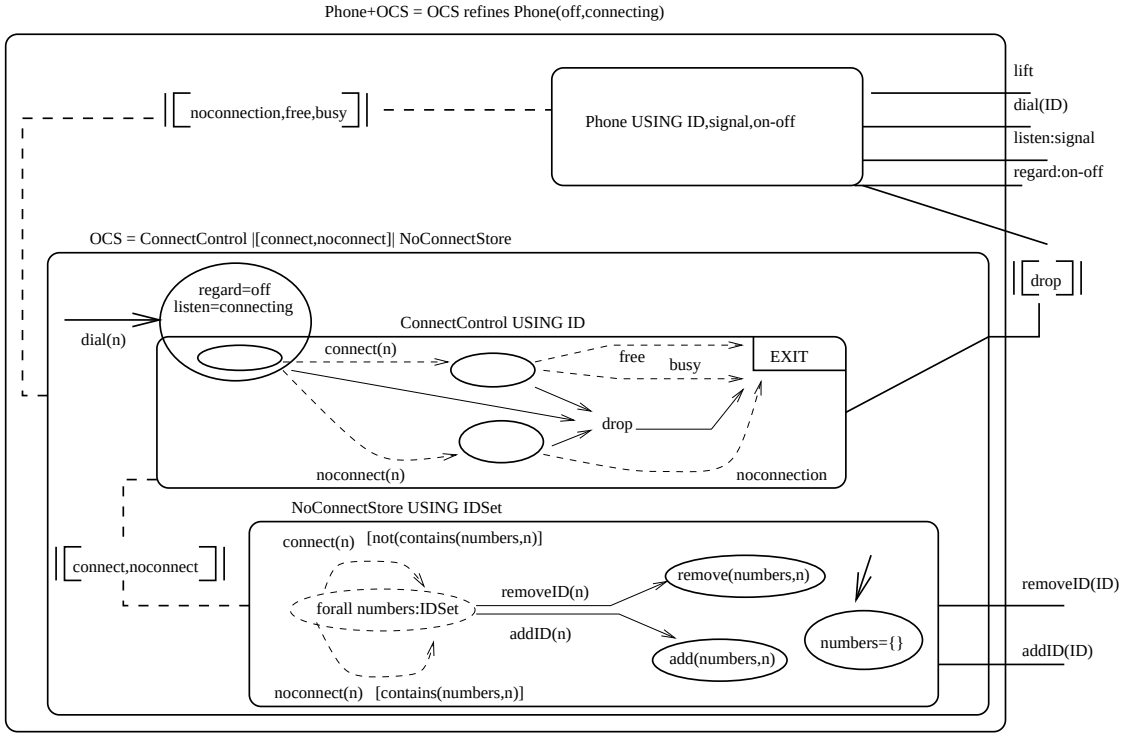

Figure 7: Originating Call Screening

There are a number of interesting points to note about this model:

The OCS Component

The new `OCS` feature is itself composed from two components: the `NoConnectStore` and the `ConnectControl`. `NoConnectStore` stores the list of numbers which have to be screened. This list is set to be empty on the creation of the store. The store always allows the addition and removal of `ID` numbers from the `IDSET` list of numbers to be screened. The store will always be able to do either a `connect(n)` or a `disconnect(n)`, but not both, depending on whether `n` is in the set of numbers to be screened. This is formally specified as a precondition (between square brackets in the diagram) on the action. We require *weak fairness* on the `connect` and `noconnect` actions in `OCS`.

`ConnectControl` is a *state refinement* of `off` and `connecting` in the `Phone`. It is a machine which gets created by the `dial(ID)` action resulting in a `Phone` which is `off` and `listening`, and which `EXITS` when the `Phone` is asked to do any event which would result in a `Phone` state change (in this case `drop, free, busy` or `noconnection`). Note that the number being dialled is available to the `ConnectControl` component on creation. `ConnectionControl` resolves some of the nondeterminism in the `Phone` specification by stating that a `free` or `busy` can occur only after a `connect`, and a `noconnection` can occur only after a `noconnect`.

Phone-OCS Configuration

We have two types of synchronisation, forced on us by the state refinement, between `OCS` and `Phone`. We must have full synchronisation on all actions which can change the state of `Phone` when `off` and `connecting`, and hence we have an *internal* synchronisation on `noconnection, free` and `busy`, and an *external* synchronisation on `drop`. We must be careful with *external* synchronisations because they may restrict the behaviour of the telephone user. For example, `OCS` should never stop the user from being able to `drop` the phone if such a `drop` would be possible in the `Phone` state. In this case, a simple analysis shows that `drop` is always

possible in `OCS` and thus the user's ability to `drop` is not affected. In fact, it is the `refines` composition mechanism which guarantees the preservation of original `Phone` behaviour when composed with `OCS` in this way.

Localisation

This change is again *local* to the user who requests the service. There is no visibility of the service to other users in the system.

Feature Classification

We classify the new feature by the way in which it is composed with the phone system. In this case, the new feature is based on a *state refinement*. Similar features are `CallBack`, `CallHold` and `Answer Machine`.

4.4 Black List

The `Black List` feature has a similar function to originating call screening, but restricts incoming rather than outgoing calls. In CS-1 this service is known as *Terminating Call Screening* (TCS) and its informal description is as follows:

> TCS, also known as incoming call barring, prevents calls from certain telephone numbers or area codes. In the extreme case, all incoming calls may be disabled; for example, a call-box may not be allowed to receive calls.

Our specification of this feature is illustrated in figure 8.

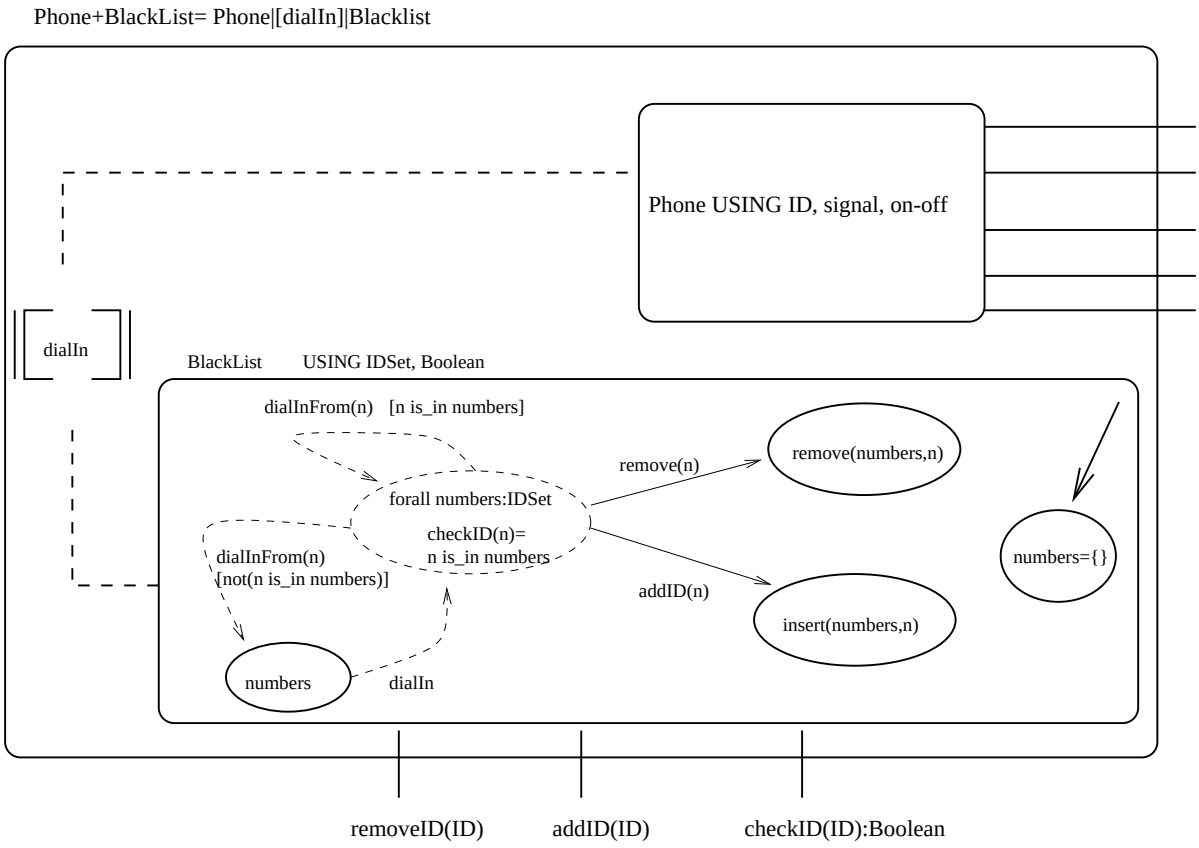

Figure 8: Black list

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

Composition Re-Use

The composition is precisely that seen for the `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? We chose to have 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.

4.5 An Analysis Method

The simple examples, above, give a flavour of our compositional approach. We have used the same method to develop a total of 12 features and 5 composition mechanisms. In this subsection we review 4 important parts of our analysis method:

- **Invariants in B**
  In [17] we see the method by which B is used to verify our invariant properties in a compositional manner. For example, in the network of many different telephones we use the invariant to specify relations between pairs of phones. A simple POTS state invariant requirement is that `only an even number of phones can be talking at the same time`. This is proved by showing that it is true in the initial state (where all phones are `off` and `ready`). Then we show that all possible state transitions maintain the required property (if it is true before the action occurs then it is true after the action occurs). This property cannot be checked through an exhaustive search of a system of an unbounded number of phones. It can be checked by proving that all transitions are closed with respect to the invariant. We can do this quite easily in B. Note that using this technique, the three-way-calling feature will break our invariant and thus tell us that there is an interaction!

- **Nondeterminism in OO LOTOS**
  In [8] we saw how a three-way-calling feature and a call hold feature could be said to interact becuase of the way in which they introduce nondeterminism into the model. In this case the nondeterminism arises becauses the flash-hook action is overloaded and the user cannot say whether it initiates one feature or the other. We have a mechanism for identifying such an interaction automatically which is founded on the classification of feature compositions.

- **Nondeterministic Interaction Resolution**
  The resolution of the introduction of nondeterminism can also be resolved automatically. We have two methods. The first forces the user to resolve the choice at run time, the second forces the specifier to resolve the problem at compile-time by stating a priority between the features.

- **Eventuality Composition in TLA**
  In [12], we show how the notion of *eventuality* can be verified compositionally using TLA.

The example of a telephone answering machine with call forwarding shows a class of feature interaction which repeatedly occurs and should lend itself to automatic detection and resolution (although, this is ongoing work).

The development of such re-usable techniques is fundamental to producing a mature and stable analysis method. We believe that such a method would be improved through the incorporation of feature classifications and meta-analysis methods.

5 Feature Classification

The question of how to classify features is fundamental to our algebraic approach. Through analysis of our specification of the CS-1 services we identify the need for a *multiple inheritance* hierarchy of feature classification. Our first attempt at defining such a hierarchy is illustrated below:

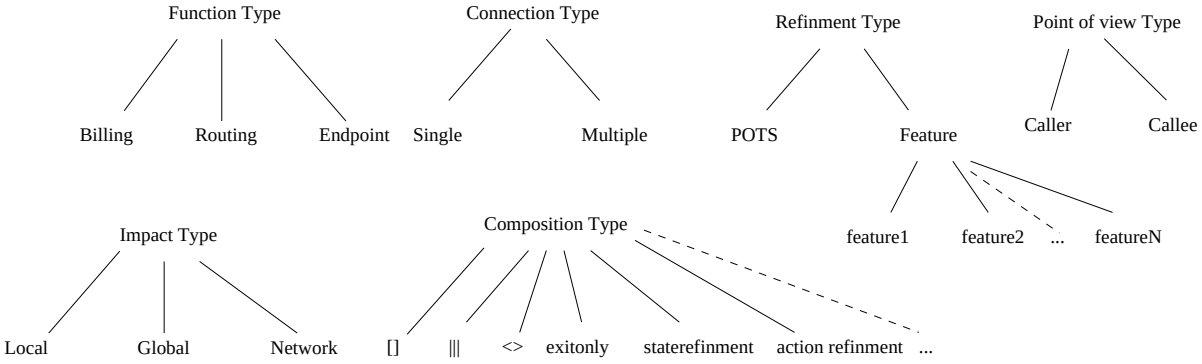

Figure 9: Feature classification hierarchy

Figure 9 shows that we have 6 fundamental classifications:

(1) Function Type

In CS-1 each feature falls into 1 of three function types, corresponding to *billing, routing* and *endpoint*. Billing features are those concerned with how each customer is to be billed. (We are currently developing such features but do not report on them in this article.) The CS-1 billing features are:

> Automatic alternative billing (AAB), Credit card calling (CCB), Freephone (FPH), Premium rate (PRM), Split Charging (SPL), Premium Charging (PRMC), and Reverse Charging (REVC).

Billing was abstracted away from in our case studies and consequently does not appear in our requirements models. Routing features are those which are concerned with setting up connections between users at the network level. The CS-1 routing features are:

> Call Forward (CF), Call Transfer (CT), Call Forward on Busy (CFB), Follow Me Diversion (SF), and Origin Dependent Routing (F).

Routing features are those which require changes to the network model[2] in order for the functionality to be specified. These features cannot be specified by changing the telephone model alone. Such features are called Endpoint features. End-point features are the easiest to specify as they require changes only at the telephone level of the requirements model. Such features are prominent in the case studies that we have carried out:

[2]The development of a *good* network model is one of our next goals. We have not managed to specify an OO network architecture at a *proper level of abstraction*. It seems to us that we are infringing on design decisions whilst we should just be building requirements.

Call Hold, Originating Call Screening, Answer Machine, Do Not Disturb, Call Identification, Terminating Call Screening, Abbreviated Dialling, and Redial.

(2) Connection Type

Connection type partitions features into those which are concerned with 2-users at one time (single connections) and those which allow more than 2-users *on the same line* (multiple connections).

(3) Refinement Type

Most features in CS-1 are intended to alter/extend POTS functionality. Such features are said to be of refinement type POTS. However, other features are not designed to change the functionality of POTS but are intended to alter/extend other features in the system. These are said to be of refinement type Feature.

(4) Impact Type

Impact type classifies *where* in our requirements models there is a need to make changes in order to incorporate the new functionality. A local impact is one in which only the telephone model of the service requester has to be changed. A global impact is one in which all the telephone models have to be change. A network impact is one in which the network model (between users/telephones) has to be changed.

(5) Composition Type

Composition type corresponds to the way in which a feature has been specified to configure with POTS. Each configuration is specified as an ensemble of compositions. Many such configurations are common to different features. Thus, composition type identifies the structure that exists between POTS and the feature being classified.

(6) Point of view Type

There are two different models for most features: the caller and the callee. Some features change the behaviour of the feature subscriber as a caller, as a callee, or as both. The role of the point of view classification is to make this property an explicit part of the requirements model.

6 Constructing an Algebra: a meta-anlysis

6.1 Pair-wise Analysis

Our case study was extended to examine compositions between pairs of features. We identified 5 different types of feature pairs:

- **Independent**: When the features are combined with POTS, there is no interaction between the features. There is no sharing of actions or state and, thus, no communication between the two features. In other words, the individual behaviour of each feature (with POTS) is exactly as before.
- **Perfect Friends**: The two features do communicate (either through shared state or actions) but this communication in no way alters the behaviour of either feature and so they do not *interact* in the sense used in this report.
- **Friends**: These features do not interact provided some of the nondeterminism in either/both of the models is resolved in some specific way. In some sense, they can be said to make implementation decisions which help their friend feature to work correctly.

- **Politicians**: There is an interaction where both features cannot work exactly as before. However, a resolution mechanism exists whereby some sort of feature priority can be used to resolve the problem automatically. Such a resolution mechanism permits a subset of one or both of the features to be maintained when the two features are combined.

- **Enemies**: There is an interaction and no resolution technique exists other than to say that when one feature is operating the other must be dormant. Two such features can exist in the same system but dynamically both features are never executing at the same time.

Using these pair-wise composition categories, we hope to be able to arrive at the point where we can formally state the following sort of properties, for example:

- Billing features are independent from interface features.

- A local-caller and a local-callee feature are always perfect friends.

- Two POTS state-refinements are friends iff they refine different states.

- Two features which refine the same feature are politicians if they refine the same state, and share an action which exits this state.

- Two features are enemies if they refine the same state but do not share an action which exits this state.

We emphasise that these properties illustrate the type of analysis which we believe can be achieved when a classification hierarchy is formalised. The process of formalising such analysis is the main goal of our current research.

6.1 An n-wise to pair-wise transformation

Of course, as we have pointed out earlier, pair-wise feature interaction is only a small part of the problem. The real difficulty lies in analysing systems with many features operating together. However, there is much hope that most interactions arise out of pair-wise configurations. For example, consider 3 features A,B and C; we believe that A,B and C interact implies that A and B, or A and C, or B and C interact in most[3] cases. Only in a small percentage of cases do A,B and C interact when the pairs AB, AC and BC do not. Thus, n-wise analysis of n features can be done using (n*(n-1)/2) pair-wise analyses (most of the time). The key to our meta-analysis is to formally define *most of the time* and have a simple algorithm for deciding when such a pair-wise approach works. This is another part of our current research.

Given a complete classification of features, we still need to prove the consistency and completeness of a feature interaction algebra. At the moment, such an algebra is an ideal more than anything else. As well as feature classes we have to consider feature composition types. Furthermore, another dimension may be added when we define different means of composing features (polymorphic "+" semantics) in response to *politician* and *friend* interaction types. This will further complicate our goal of developing a usable algebra.

7 Conclusions

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 themselves 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

[3]Our case study would suggest 90 percent, but more analysis needs to be carried out.

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. Such an algebra should provide a meta-analysis framework in which the complexity explosion nature of feature interactions is greatly reduced.

References

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

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

[3] 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.

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

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

[6] Brad Cox. *Object oriented programming: an evolutionary approach*. Addison-Wesley, 1986.

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

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

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

[10] Mermet Gibson and Méry. Feature interactions: A mixed semantic model approach. In *Irish Workshop on Formal Methods*, Dublin, Ireland, July 1997.

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

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

[13] Adele Goldberg. *Smalltalk-80: The Interactive Programming Environment*. Addison-Wesley, 1984.

[14] Adele Goldberg and David Robson. *Smalltalk-80: The language and its implementation*. Addison-Wesley, 1983.

[15] C.A.R Hoare. *Communicating Sequential Processes*. Prentice-Hall International, 1985.

[16] B. Liskov and Zilles S. Programming with abstract data types. In *ACM SIGPLAN Notices*, number 4 in 9, pages 50–59, 1974.

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

[18] B. Meyer. *Object Oriented Software Construction*. Prentice Hall, 1988.

[19] Kristen Nygaard and Ole-Johan Dahl. Simula 67. In Richard W. Wenenlblat, editor, *History of Programming Languages*. Wenelblat, 1981.

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

[21] A. Snyder. Common objects: an overview. *ACM SigPlan Notices*, October 1986.

[22] Bjarne Stroustrup. *The C++ Programming Language*. Addison Wesley, 1986.

[23] P. Wegner. Dimensions of object-based language design. In *Special Issue of SIGPLAN notices*, pages 168–183, October 1987.