POTS :An OO LOTOS Specification

J. Paul Gibson, Yassine Mokhtari*
LORIA-UMR n° 7503-CNRS &
Université Henri Poincaré
BP 239, 54506 Vandoeuvre-les-Nancy
France

January 1998

Key Words: Validation, Telephone Service, Formal Methods

Abstract

We believe that a more rigorous method of specification and validation can be achieved by first developing a *specification architecture* whose high-level semantics are based on object oriented concepts. This architecture promotes the construction of new functionality in a formal manner using rigorously notions of composition and inheritance. An object oriented approach will also facilitate incremental approaches to validation and verification.

We present our first steps towards producing such an architecture for the Plain Old Telephone Service (POTS), which is specified and validated using a formal object oriented language based on LOTOS. The method by which the formal model is derived from the informal understanding of the requirements is examined. Validation based on meta-analysis of the problem structure is elucidated.

*email: gibson@loria.fr, mokhtari@loria.fr

Contents

1 Introduction

The problem of feature interactions in telephonic systems is well documented. Formal languages have been used in the development of such systems to improve the means of analysing requirements models for undesirable behaviour. Unfortunately, there is no high-level means of synthesising and analysing systems in which many features exist (as is the case in real telephone networks). Thus, it has been necessary in the past to utilise ad-hoc means of specifying features and testing their functionality in complete systems.

We believe that a more rigorous method of feature composition and validation can be achieved by first developing a *service specification architecture* whose high-level semantics are based on object oriented concepts. This architecture promotes the construction of new features in a formal manner using rigorously notions of composition and inheritance. The object oriented approach will also facilitate incremental approaches to validation and verification.

We present our first steps towards producing such an architecture. Specifications are given using LOTOS (we discuss the different styles and techniques which this language supports and motivate the adoption of an object oriented approach) although the same principles should be applicable in a wide range of languages.

2 Introducing LOTOS

LOTOS (*Language Of Temporal Ordering Specifications*), see [11, 48, 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 [21]) and process algebra part (similar to CSP [27] and CCS [37]) is advantageous sinceit provides the flexibility of two different semantic models for expressing behaviour, whilst managing to integrate them in a relatively coherent fashion.

2.1 Syntax and Semantics

LOTOS is a language using a finite alphabet of observable actions to describe systems as a set of interacting processes. One can distinguish between basic-LOTOS and full-LOTOS. In basic-LOTOS, the interaction between processes is described by pure synchronisation without exchanging values, whilst in full-LOTOS processes can exchange values of particular **sorts**[1]. Process definitions can also be parameterised by **sorts**. It is the abstract data type part of the language which defines the sets of values (**sorts**) which can be used for such parameterization in the process algebra specifications.

[1]In fact, **sorts** can be used to parameterise the labelling of events at particular gates. Then, events can be conceptualised as some sort of exchange of data between behaviour expressions during event synchronisation.

Syntax: Basic LOTOS

In this section, we present the syntax of basic-LOTOS. For further details on these topics we refer to [5]. In the syntax of basic-LOTOS, a process is defined by its behaviour expression. In the usual manner, construction operators are defined for creating new behaviour expressions from simpler expressions (see the following table) :

Name	*Syntax*
inaction	$stop$
unobservable action prefix	$i; B$
observable action prefix	$g; B$
choice	$B-1[]B-2$
general parallel composition	$B_1\|[g_1, \cdots, g_n]\|B_2$
pure interleaving	$B_1\|\|\|B_2$
full synchronisation	$B_1\|\|B_2$
hiding	$hide\ g_1, \cdots, g_n\ in\ B$
process instanciation	$P[g_1, ..., g_n]$
successful termination	$exit$
enabling	$B_1 >> B_2$
disabling	$B_1[> B_2$

Semantics: basic-LOTOS

The operational semantics of LOTOS is defined as a simple behavioural tree of actions. The set of actions that a behavioural expression can perform at any given time in its execution are defined by following the following axioms and inference rules. The following notations are used:

S denotes the set of user-definable gates
S' denotes any subset of S
$g_1, \cdots, g_n$ ranges over S
g^+ ranges over $S \cup \{\delta\}$
μ ranges over $S \cup \{i\}$
μ^+ ranges over $S \cup \{i, \delta\}$
$B, B_1, B_1', \cdots$ represent behaviour expressions

Name	*Axioms and Inferences rules*
Inaction	*no rules*
Prefix	$\mu; B \xrightarrow{\mu} B$
Termination	$\textbf{exit} \xrightarrow{\delta} \textbf{stop}$
Choice1	$\dfrac{B_1 \xrightarrow{\mu^+} B_1'}{B_1[]B_2 \xrightarrow{\mu^+} B_1'}$
Composition1	$\dfrac{B_1 \xrightarrow{\mu} B_1' \textit{ and } \mu \notin S'}{B_1\|[S']\|B_2 \xrightarrow{\mu} B_1'\|[S']\|B_2}$
Composition3	$\dfrac{B_1 \xrightarrow{g^+} B_1' \textit{ and } B_2 \xrightarrow{g^+} B_2'}{B_1\|[S]\|B_2 \xrightarrow{g^+} B_1'\|[S]\|B_2'}$
Hiding1	$\dfrac{B \xrightarrow{\mu^+} B' \textit{ and } \mu^+ \notin S}{\textbf{hide } S \textbf{ in } B \xrightarrow{\mu^+} \textbf{hide } S \textbf{ in } B'}$
Hiding2	$\dfrac{B \xrightarrow{g} B' \textit{ and } g \in S}{\textbf{hide } S \textbf{ in } B \xrightarrow{i} \textbf{hide } S \textbf{ in } B'}$
Enabling1	$\dfrac{B_1 \xrightarrow{\mu} B_1'}{B_1 >> B_2 \xrightarrow{\mu} B_1' >> B_2}$
Enabling2	$\dfrac{B_1 \xrightarrow{\delta} B_1'}{B_1 >> B_2 \xrightarrow{i} B_2}$
Disabling1	$\dfrac{B_1 \xrightarrow{\mu} B_1'}{B_1[> B_2 \xrightarrow{\mu} B_1'[> B_2}$
Disabling2	$\dfrac{B_2 \xrightarrow{\mu^+} B_2'}{B_1[> B_2 \xrightarrow{\mu^+} B_2'}$
Disabling3	$\dfrac{B_1 \xrightarrow{\delta} B_1'}{B_1[> B_2 \xrightarrow{\delta} B2}$
Instanciation	$\dfrac{\textbf{Process } P[g_1', \cdots, g_n'] := B_p \textit{ and } B_p[g_1/g_1', \cdots, g_n/g_n'] \xrightarrow{\mu^+} B'}{P[g_1, \cdots, g_n] \xrightarrow{\mu^+} B'}$
Relabeling1	$\dfrac{B \xrightarrow{g'} B', \phi = [g_1/g_1', \cdots, g_n/g_n'] \textit{ and } g/g' \in \phi}{B\phi \xrightarrow{g} B'\phi}$
Relabeling2	$\dfrac{B \xrightarrow{\mu^+} B' \textit{ and } \mu^+ \notin \{g_1', \cdots, g_n'\}}{B\phi \xrightarrow{\mu^+} B'\phi}$

2.2 Specification Styles

Different styles have been identified for writing LOTOS specifications [48]. Each style is closely related to the description of the behaviour of the system. The style can be said to reflect the way in which the behaviour being specified is understood. A style can enforce a particular problem conceptualisation by restricting the way in which the LOTOS can be used.

In the desciption of observable behaviour, a system behaves like a *black box*, i.e. the internal events are not visible. Two fundamental styles dominate:

- The **monolitic style** uses the choice and sequence operators to explicitly define the behaviour as an action tree. This style is useful for debugging and testing the specification of small systems.

This style does not adhere to any of the commonly accepted principles for the construction of non-trivial behaviour. It provides no means of decomposing problems and the specifications constructed in this way are said to be semantically flat.

- The **Constraint-Oriented style** composes a set of independent constraints in parallel. A constraint is simply a behavioural expression at some level of abstraction. We identify three types of constraint which are useful for the development of system behaviour:

  - The **local constraints** are applied internally to the components of the system.
  - The **end-to-end constraints** are applied between the different components.
  - The **global constraints** are applied to the overall of the system (at its external interface).

  The constraint oriented style is abstract, implementation-independent, and provides a stepwise means for developing system requirements. However, it is not well suited to the specification of problems where the constraint decomposition does not reflect the structure of how the system is to be understood. Certainly, the notion that two interacting entities can constrain each other's behaviour is useful (but defining the entities themselves as constraints does not seem a fruitful approach to structuring a system in an understandable way. Furthermore, such an approach does not give the specifier a means of using internal structuring mechanisms to imprve the understanding, development and re-use of components.

System behaviour can also be described as a *white box*, i.e. the internal and external events are observable. One can identify two fundamental styles :

- The **State-Oriented style** can be viewed as the extension of the monolitic style with state variables. The specification is defined as sets of alternative (parameterised) sequences. Each alternative is guarded. This style provides a directly implementable system described as an automata. However, it is tedious for defining systems with many different states. The paremeterisation provides only one level of compositionl structure whereas it would be beneficial to view a system as state machines inside state machines inside state machines etc ....

- The **Resource-Oriented style** describes each resource of the system as a process and uses the parallel operator to compose them. The internal actions can be hidden by making them invisible from the external behaviour of the system. In this style, the specification imposes a correspondance between specification and implementation. Furthermore, although there are often many different types of resource in a particular problem domain, the style does not offer any means of helping to distinguish between them. Classification of different types of resources is a useful way to improve understanding and component re-use. The resource oriented style does not address these issues.

In practice, a mixture of theses styles is often used to describe system behaviour (for instance we can use the resource-oriented style at the top of the specification and each resource can use any style among the monolithic, the constraint-oriented and the state-oriented). More generally, the different styles are clearly suited to modelling behaviour at different levels of abstraction. The problem with this is that during system development there is never a consistent level of abstraction for the whole system model, and so we end up having LOTOS specifications with a mixture of styles. This inconsistency is made worse by the fact that choices of style, changes of style, different mixtures of style, etc ... are not well documented (or even understood). What we need is a consistent specification style which can be used at different levels of abstraction. One such *style* is *object oriented*, and we base our work on the formal object oriented development method (FOOD) in [22]. This is reviewed in the next section.

2.3 LOTOS in POTS specifications: a critique

The Plain Old Telephone Service (POTS) problem has been addressed by LOTOS specifiers in many different papers.

[2] views the telephone system as a set of nodes connected by lines. Each line can be used to transmit several conversations simultaneously. Each conversation is called a *channel*. A conversation is not transmitted on one single channel: it follows a path formed by several physical lines. The nodes consist of *switching modules* performing the switching modules of channels. They proposed an algebraic abstract model, modeled as an object, that records the history of functions which are used to manipulate it. (For example, the function that may connect or disconnect a channel is modelled as a service at the interface of the object.)

Tvrdy[29] published an early paper about the specification of telephone systems in LOTOS; this was followed by several more specific papers[9, 36, 51]. In general, the specification of the telephone system is done using the constraint-oriented style. The description uses a mixture of different styles but with the constraint-oriented style dominating the others. These works are similar in their specification of the Plain Old Telephone Service (POTS). The underlying structure is common (although the specifications themselves are very different). This structure is illustrated in figure1.

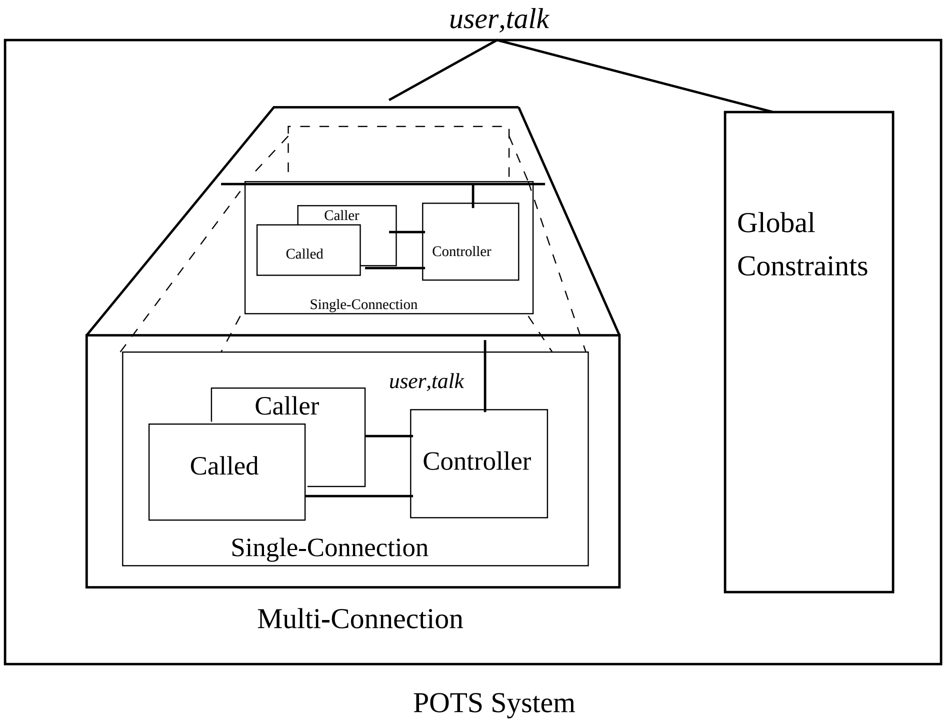

Figure 1: POTS Arcitecture

Intuitively, composition is represented by containment: for example, many single-connection processes are composed to make a multi-connection process. Interleaved processes (when there is no synchronisation) are drawn on top of each other: for example, caller and called in a single connection both interact with controller but never synchronise with each other. Processes synchronisations are represented by thick lines connecting process boxes: for example, POTS synchronises with multi-connection and global constraints (on the same gates).

Within this architecture, the integration of a new feature is done by making the appropriate modification to the user on which the feature will be activated *i.e the caller or the called or both* as well as its *controller*. The main drawback is that for each feature the specifier must decide how to extend the POTS system to support it: the efficiency of this method depends on the knowledge and experience of the specifier. This structuration suffers from its inability to permit incremental system extensions which do not require changes to the general architecture model. Furthermore, the notion of feature is absent from the conceptualisation: the addition and combination of features is just somehow tagged onto the system structure in some unobvious manner.

The two different types of extension are illustrated in figure 2

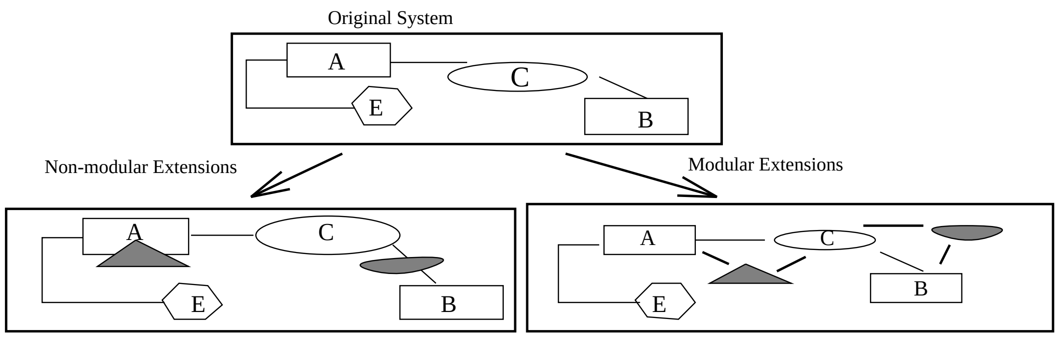

Figure 2: Extended POTS

We aim for an architecture which supports the incremental development of systems in a modular fashion. The principle is one of *open* for extension but *closed* to alteration. An object oriented strategy adheres to this principle.

3 OO LOTOS

LOTOS is chosen as a suitable object oriented requirements capture and design language for the telephone service (and features) specification because:

- it suits our need for semantic continuity from analysis to design: the ACT ONE requirements model can be incorporated within the full LOTOS design model.
- LOTOS has already been the subject of research with regard to its suitability for modelling object oriented systems and incorporating object oriented principles: for example, see [3, 50, 44, 34, 32, 18, 13].
- There is wide support, often in the form of tools, for the static analysis and dynamic execution of LOTOS specifications: for example, see [49, 26, 6, 40].
- LOTOS has already been applied to the problem of POTS and feature specification, although not in an object oriented manner.

3.1 Object Oriented Principles and Advantages

The object oriented paradigm arose out of the realisation that functional decomposition is not the only means of structuring code: an alternative is to construct a system based on the structure of the data[2]. Emphasis on data structure led to the encapsulation of functional behaviour within data entities: *objects*[3].

Object oriented concepts were conceived in Simula [39], went through infancy in Smalltalk [25, 24] and could be said to be leaving adolescence, and approaching maturity, in the form of many different languages (for example: Objective C [17], C++ [47], LOOPS [4], Flavours [12, 38], CLOS [19, 31], Eiffel [35] and Common Objects [46]) and methods (for example: those of Rumbaugh [45], Coad and Yourdon [14, 15], Cox [17], Meyer [35] and Booch [8, 7]).

There has been much interest in combining formal and object oriented methods. The research falls into two main categories:

- **i) Using Object Oriented Techniques To Construct Formal Models**
  The success of object oriented techniques in software development has led to much interest in using the same techniques for building formal models. Much of this work centres on the definition of object oriented constructs, or the interpretation of object oriented concepts, in an existing formal language. Good examples of the type of work which has been done can be found in [13, 3, 18, 32, 44, 50, 23, 43, 34, 33]. This work has led to recognition of the inconsistent use of object oriented terminology, highlighting the need for a concensus of opinion. Further, much of the work shows the difficulties inherent in modelling object oriented behaviour in a semantic framework which was not designed for such a purpose.

- **ii) The Development of Object Oriented Semantics**
  The lack of agreement on the meaning of object oriented constructs, reinforced by the informal semantics of most object oriented programming languages, has led many people to produce formal object oriented semantics, for example see [10, 55, 42, 52, 20]. The thesis by Wolczko [54] provides a more complete view of the technical issues, whilst Wegner [53] and America [1] examine the philosophical aspects.

The FOOD approach of Gibson [22] is based on these theoretical foundations and has also proven itself in the specification of service specifications in an industrial telecommunciations environment, For that reason, it is chosen as our method for the development of our LOTOS POTS specifications.

The principles which make object oriented approaches sucessful are, in our opinion, as follows:

- **Conceptual consistency**
  This is the ability to reason about systems at different levels of abstraction using the same concepts (albiet, also expressed at different levels of abstraction). Shifting levels of abstraction does not mean changing the things that you are thinking about only the way in which you are thinking about them. Thus, an object oriented model can progress from the abstract to the concrete in a continuous fashion.

- **Simplicity**
  The main concepts are those of encapsulation, composition and classification. These notions are

[2] Of course, there are programming languages which do not place emphasis on functional or data structure, but we do not consider them in any detail as part of this work.

[3] Two well known data-based sofware development methods which are generally accepted as not being object oriented are the quite similar approaches put forward by Jackson [30] and Orr [41]. These approaches are closely related to the object oriented paradigm in the initial analysis stages, but digress from the standard object oriented view as they approach implementation.

familiar to all engineers and provide a good basis upon which problem understanding and modelling can begin.

- **Open for extension**
  The notion of subclassing provides a powerful means of extending the behaviour of a system (or subsystem) class without having to make changes to already specified behaviour.
- **Closed to alteration**
  Once parts of a system are coded and validated then they can be incorporated in a new system (i.e. reused) in a very safe way which ensures that the behaviour they offer is not changed (even though their implementation may be changed).
- **Emphasis on re-use**
  The methods emphasize re-use by incorporating re-use operators as part of the language semantics (they are not just syntactic sugarings).
- **Controlled Polymorphism**
  The ability of an object to be viewed as a member of different classes (depending on context of use) is very important. This is a powerful mechanism which is also open to abuse in universally polymorphic languages. However, the classification hierarchy in object oriented systems provides a means of controlling this facility (without reducing its utility). By allowing any object to be treated as a member of any of its ancestor classes, we can use the property of *substitutability* to maintain the correctness of our ever changing systems.

3.2 OO ACT ONE: Requirements Models

The abstract data typing language ACT ONE provides an executable (abstract) model for customer validation. The structure of the resulting ACT ONE requirements model corresponds to the structure of the problem domain, as communicated by the customer.

The concrete syntax which we employ in the object oriented requirements model incorporates the following:

- A means of categorising entities into classes of behaviour.
- A mechanism for representing a set of operations associated with each class, where each operation associates one or more classes of entity with a resulting class of entity. In other words, a means of recording the external interface of a class so that all operations (on class members) can be statically 'type checked' for correctness.
- A means of defining the behaviour associated with each operation. In other words, a set of equations or axioms which give meaning to the operations.
- A facility for defining one class of behaviour in terms of other component classes of behaviour.
- An explicit means of representing the structure of the problem domain.
- Parameterised classes of behaviour (genericity)
- Inclusion polymorphism (subclassing).

An ADT provides us with a means of specifying 'implementation free' behaviour. This is ideal for requirements capture: analysts must try to identify and record *what* is required rather than *how* these requirements are to be met. However, a set of requirements must always contain some structure otherwise it would be impossible to record or understand them. The object oriented method of analysis and requirements capture encourages the recording of certain structural aspects of the problem domain. This aids understanding and gives the designers an initial structure upon which the design can be developed. In this way a formal statement of object oriented requirements is useful in later stages of development on two accounts: it unambiguously defines what is needed and it provides a structure for understanding the needs.

It is clear that type and class should not be confused [16], but we do believe that types can be used to implement the semantics of the class notion. Types are more general than classes. In our development we generate type specifications from a formal model of object oriented requirements. The set of behaviours that can be specified in this way is much smaller than the set of all behaviours which can be specified using ADTs. The differences between types and classes (subtypes and subclasses) arise from the way in which the terminology is applied rather than from differences in the underlying principles. The three roles of types, namely abstraction, re-use and validation, are equally applicable to classes:

- **Abstraction:** classes define an abstract interface behind which all the properties of objects in the class are encapsulated.
- **Re-use:** classes provide a fundamental package of re-usable behaviour.
- **Validation:** object oriented systems can be statically analysed to guarantee that all service requests to each object in the system, which may occur in the system lifetime, are available as part of the interface of the class to which the object belongs.

Problems arise in conceptually relating class with type when type is taken to represent a purely static syntactic interface. It is necessary to consider the behaviour offered by type 'members' through their interfaces. Abstract data types provide both syntactic and semantic views of interface. Consequently, we support the view that classes and ADTs can be usefully related in a formal framework.

The exact syntax and semantics of our OO ACT ONE is shown in the POTS requirements models in the next section.

3.3 OO LOTOS: Design Models

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

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

Given a set of system requirements specified in ACT ONE, there are a number of different ways in which these requirements can be translated to an initial abstract LOTOS design. Object oriented designers must initially identify the communication aspects of the way in which the underlying object oriented behaviour is to be fulfilled. The designers of a system must decide how the behaviour is to be offered at its external interface (and what this external interface should look like). This simple decision can affect the rest of the design process. Identifying an object oriented communication model and specifying the translation from OO ACT ONE, is not simple. There are a number of alternative models and a number of ways in which these can be specified. Four of these alternatives are examined in the thesis by Gibson[22]. The list is not exhaustive and the ways of specifying the models are limitless.

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

3.4 OO Development

We show the effectiveness of LOTOS in representing object oriented POTS designs. The object oriented LOTOS specification is open to formal manipulation, as part of the design process, as a means of aiming towards a particular implementation environment. However, for the sake of simplicity, we report only on the most abstract POTS design which includes a high degree of implementation freedom whilst capturing our requirements precisely and completely.

The POTS development illustrates how an object oriented approach narrows the gap between analysis and design. Consistency of representation, and conceptual congruence between the way in which the problem is defined and the way in which it is solved in an implementation, led us to believe that it was correct to use the structure of the initial analysis model as a high level design. Much of the work done in analysing a problem is therefore incorporated in the design of a solution. Our initial analysis identified components of the system, i.e a decompositional approach was applied to the whole problem. Each of the components was further decomposed until decomposition was no longer necessary to improve understanding. This implied a purely top-down method, whereas the actual development combined this with a bottom-up strategy. Components, i.e. the object oriented processes, were specified such that different parts of the system were at different levels of abstraction during the development of the specification.

Architectural Concerns

There is a definite structure contained in the LOTOS specifications of the POTS model. We need to address the question of why it is there. In large, complex specifications it is impracticable to reason about the behaviour of the system as a whole. This suggests that a conceptual decomposition of the problem must be inherent in the specification. We will refer to a specification as being structured only if the decomposition of the problem is explicit.

The main benefit of having a structured specification is that it can be used to aid understanding[4]. Clearly a specification that is easier to understand is also easier to implement. The introduction of structure should also make the system more amenable to changes whose effects are kept localised as much as possible.

Arguments against structured specifications concentrate on their constraining nature. Implementers may argue that the decomposition of the system, as captured by the specification, is not the way in which they would choose to structure the code. There may be a conflict of interest between writing the specification to aid understanding and writing it in a way that eases the step towards implementation.

This is certainly a concern when implementers are working to specific constraints imposed by the programming language or performance demands. In some cases the conflict of interest between the

[4] It is likely that a poor structure could also hamper understanding.

specification and the implementation may mean that the structure within the specification cannot be followed by the implementers. It is effective use of systems analysis and bottom-up knowledge that mitigates against this.

Within the object oriented paradigm, this problem is not as prominent. The decomposition of a system into a set of communicating objects is consistent at all stages of the development cycle. In particular, there is a closer binding between the specification and implementation architectures. The price for this is that object oriented specifications are less abstract. The structure of the problem has not been abstracted; instead it has been carefully represented in the specification with the intention that it be used in the implementation.

A perfect scenario would occur when the initial system analysis produced an informal problem decomposition that is acceptable to both specifiers and implementers. Here we mean acceptable in the sense that the decomposition can be used directly by both. In the object oriented paradigm this would require the objects within the specification to have counterparts in the implementation. The structure of the system would then be completed by realising the communication between these components.

There are other advantages in reusing the specification structure in the implementation:

- **Generality**
  Writing a structured specification requires making a number of design decisions. In an object oriented specification these decisions are often concerned with producing components that are general. This generality can then be exploited for component extension and reuse.

- **Testing**
  Making the specification and implementation structures as isomorphic as possible makes traceability easier, in the sense of a design audit. Testing the system can be done in a bottom-up fashion. This gives more confidence in the code being a valid implementation of the specification.

- **Controlling Change**
  Extending or changing the system can be achieved in a more controlled manner. In an object oriented specification, modifications can be kept localised. However, if the implementation has a different structure from the specification then updating the code to match the specification may require changes across the whole system. Structural compatibility means that changes to the specification can be more easily incorporated in the implementation. Verification of the new system can concentrate on the components that have been altered.

The POTS problem domain is also unique because of the inherent executability of the requirements models. Telecom engineers are well versed in presenting informal feature specifications as types of simple state transition systems. The OO ACT ONE semantics are based on a formailisation of objects as instances of state transition machines (together with a sound theory for the relationship between classes of objects). Thus, we hope to show that an object oriented approach is also natural for feature specification.

4 POTS Specification

4.1 Introducing POTS and Features

The telephone system may be regarded as a system with a set of phones and a switch(`POTS`) which establish the connection between them see the figure below 3.

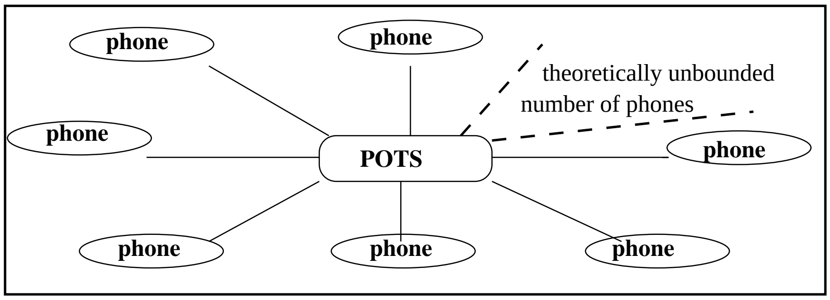

Figure 3: Telephone System

Each phone is identified by a unique number. Thus it also identifies the user. The users communicate via `POTS` by using a well defined set of primitives. These primitives reflect what the user can do. A simple scenario to illustrate how the user can used the system may be the user **lifts** his handset and **dials** the number of the callee. Next, the **connection** will be established between the users if the callee is not **busy** and they begin **talking**. Finally, the communication terminates by one of the users **dropping** their handset. `POTS` allows at most two users to be connected and several sumletanous connections. The possibility for several users to be engaged ar the same time in one connection is considered as an additional feature. The behaviour of POTS is that which any (competent) telephone user is familiar with.

The `plain old telephone service` (POTS) forms the basis for many complex forms of telephone feature. A feature is some service offered to meet some communication need. Features are the fundamental building blocks for telephonic communication. The difference between feature and service (which is some sort of base behaviour for a group of features) is not important in our presentation.

We must ask why the development of telephone services is in any way different from the development of other systems. Why cannot we apply traditional development methods in the development of such systems? The answer is: we can employ standard methods but there is an added complexity in the specification of feature which requires something more. Namely, telecom systems evolve at a rapid rate and the increments of development are features. Thus we do not have a stable base upon which to build a firm foundation. The only means of providing a stable architecture of development is to develop a high-level understanding of features and to build a generic framework in which this understanding is captured.

The feature interaction problem is stated simply as follows. A feature interactionis a situation in which system behaviour (specified as some set of features) does not as a whole satisfy each of its component features individually. The problem is further compounded by features being deliberately defined to interact. These so called "wanted feature interactions" are problematic because they are fundamentally no different from "unwanted interactions". We therefore require a means of identifying all types of interaction, letting the customer decide whether the interaction is wanted, an if not wanted then providing a facility for removing it. This is where a formal language for the representation of features is very important.

As we see things, the problem with the current work on feature interaction is not that the work is incorrect, but is that it is not easily extensible to be able to manage future feature requirements that arise from the rapidly growing capabilities of our telecom systems. The introduction of a new feature is not incremental in the sense that we require. For a new feature specification we must be able to validate its behaviour before it is implemented. We must be able to validate its behaviour without having to validate the behaviour of all other features in the network (again). We must be able to identify areas where feature interactions do not exist, may exist and definitely exist. Thus making our integration of a new feature into a system much easier. Finally, we must be able to re-use already existing features in the production of new features. This is not a simple task.

4.2 POTS OO ACT ONE requirements model

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

- **UID** and **UIDGen**
  These are used to identify telephones (users) and to generate new user identifications which have not been previously allocated.
- **USet**
  This is simply a set of users with operators for adding and deleting users, and checking if the set is empty or contains a particular user.
- **UPair**
  This class is a simple 2-tuple (pair) of user identifiers.
- **UPSet**
  This class is a set os identifier pairs.
- **Hook**
  This class provides a simple enumeration of the state of the hook of a telephone (either `on` or `off`).
- **Signal**
  This class is a simple enumeration of the state of the signal from a telephone (whilst on or off hook). The signals available are `silent, ringing, talking, ready` and `busy`.

The means by which the ACT ONE syntax and semantics are used to define classes of behaviour is illustrated by the `Telephone` and `POTS` class specifications which follow. It should be noted that these ACT ONE specifications can be derived directly from a more syntactically rich object oriented language called OO ACT ONE. This enforces the object oriented principles which would not be evident in any arbitrary ACT ONE specification.

4.2.1 The Telephone Class

The `ACT ONE` interface specification of the *Phone* class is given by the operations defined below:

type Telephone is UserID, POTS **sorts** Telephone
opns
NewTelephone:UID -> Telephone (* INITIALISER *)
strTel: UID, Hook, Signal -> Telephone (* STRUCTURE *)
listen: Telephone -> Signal (* ACCESSOR *)
offHook: Telephone -> Bool (* ACCESSOR *)

```
ID: Telephone -> UID (* ACCESSOR *)
enabled: Telephone -> Bool (* ACCESSOR *)
drop, lift: Telephone -> Telephone (* TRANSFORMER *)
dial: Telephone, UID -> Telephone (* TRANSFORMER *)
dialIn: Telephone, UID -> Telephone (* INTERNAL *)
otherBusy,otherFree:Telephone -> Telephone (* INTERNAL *)
otherChangeHook:Telephone -> Telephone (* INTERNAL *)
TelephoneEXC:-> Telephone (* EXCEPTION *)
```

The following notes should help to explain the syntax and semantics of the ACT ONE code.

- The `type Telephone` is used to package together a number of `sort` definitions together for re-use. In this case it packages two predefined `type` packages (`UserID` and `POTS`) together with a new sort (`Telephone`) into a new `type` package.
- The `NewTelephone` is an INITIAL value which corresponds to a Phone in its initial state.
- The `STRUCTURE StrTel` is used to define the components of every phone to be fixed as a triple of an identifier, a hook state and a signal state.
- The `ACCESSORS` returns value to the requester of such a service (without changing the internal state of the object). The `enabled` accessor is common to all OO ACT ONE objects. It returns true provided the object is not in an exception state.
- The `TRANSFORMERS` define services which change the state of a `Telephone` object, but do not require any result to be returned to the service requester.
- The `INTERNALS` define state transformations that occur nondeterministically inside the `Telephone` and cannot be requested through its external interface.
- The `EXCEPTION` is common to all OO ACT ONE specifications and is used to represent undesirable (or as yet undefined) behaviour.

The semantics of the `Telephone` class are defined by the ACT ONE equations of the `Telphone` sort. These are as follows:

```
eqns forall UID1,UID2: UID, tel1,tel2: Telephone,
hook1,hook2: Hook, signal1,signal2: Signal
ofsort Bool
offHook(strTel(UID1, hook1, signal1)) = hook1 eq off;
enabled(TelephoneEXC) =false; enabled(strTel(UID1,hook1,signal1)) =true;
ofsort Signal
listen(strTel(UID1, hook1, signal1)) = signal1;
ofsort UID
ID(strTel(UID1,hook1,signal1)) = UID1;
ofsort Telephone
NewTelephone(UID1) = strTel(UID1, on, sil);
(listen(tel1) eq tal) and not(offhook(tel1)) => talk(tel1) = tel1;
not((listen(tel1) eq tal) and not(offhook(tel1))) =>
talk(tel1) = TelephoneEXC;
hook1 eq on => drop(strTel(UID1,hook1,signal1)) = TelephoneEXC;
hook1 eq off => drop(strTel(UID1,hook1,signal1)) = strTel(UID1,on,sil);
```

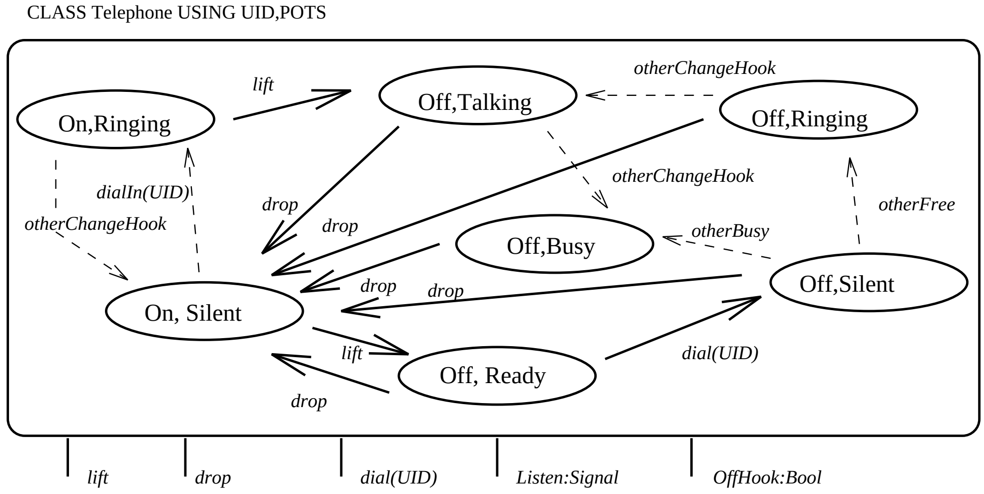

Figure 4: Phone O-LSTS Interface Diagram

(* LIFT *)
hook1 eq off =>
lift(strTel(UID1,hook1,signal1)) = TelephoneEXC;
(hook1 eq on) and (signal1 eq sil) =>
lift(strTel(UID1,hook1,signal1)) = strTel(UID1,off,rea);
(hook1 eq on) and (signal1 eq rin) =>
lift(strTel(UID1,hook1,signal1)) = strTel(UID1,off,tal);
(* DIAL *)
hook1 eq on => dial(strTel(UID1,hook1,signal1),UID2) = TelephoneEXC;
(hook1 eq off) and (not(signal1 eq rea)) =>
dial(strTel(UID1,hook1,signal1),UID2) = TelephoneEXC;
(hook1 eq off) and (signal1 eq rea) =>
dial(strTel(UID1,hook1,signal1),UID2) = strTel(UID1,off,sil);
hook1 eq off =>
dialIn(strTel(UID1,hook1,signal1), UID2) = TelephoneEXC;
(hook1 eq on) and not(signal1 eq sil) =>
dialIn(strTel(UID1,hook1,signal1), UID2) = TelephoneEXC;
(hook1 eq on) and (signal1 eq sil) =>
dialIn(strTel(UID1,hook1,signal1), UID2) = strTel(UID1, on, rin);
(* OTHERBUSY, OTHERFREE, OTHERCHANGEHOOK ... Similarly*)
end type (* Telephone *)

The `O-LSTS` diagram for the `Telephone` class is given in figure4.

This is a state transition system representation of the telephone behaviour. The accessors are not marked (they are simply null state transitions which return the appropriate values representing the state of the phone). Internal transitions are identified by the dotted state transformations.

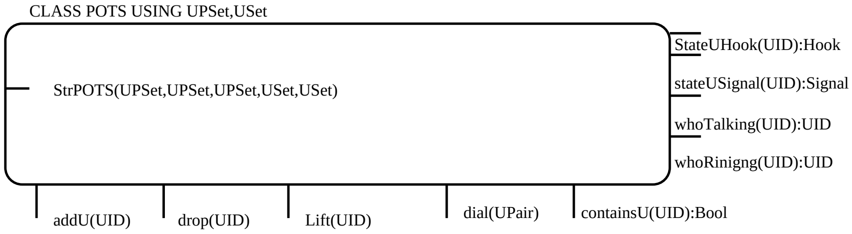

Figure 5: POTS O-LSTS Interface Diagram

4.2.2 POTS

The interface specification of the `POTS` class is defined by the ACT ONE operations below:

> **type** POTS is UPSet, USet **sorts** POTS, Hook, Signal, HS, STist
> **opns**
> NewPOTS:-> POTS (* INITIALISER *)
> strPOTS: UPSet,UPSet,UPSet,USet,USet -> POTS (* STRUCTURE *)
> stateUhook: POTS, UID -> Hook (* ACCESSOR *)
> stateUsignal: POTS, UID -> Signal (* ACCESSOR *)
> containsU: POTS, UID -> Bool (* ACCESSOR *)
> enabled: POTS -> Bool (* ACCESSOR *)
> whoRinging: POTS, UID -> UID (* ACCESSOR *)
> whoTalking: POTS, UID -> UID (* ACCESSOR *)
> dial: POTS, UPair -> POTS (* TRANSFORMER *)
> drop: POTS, UID -> POTS (* TRANSFORMER *)
> addU: POTS, UID -> POTS (* TRANSFORMER *)
> lift: POTS, UID -> POTS (* TRANSFORMER *)
> POTSEXC: -> POTS (* EXCEPTION *)

The `POTS` specification also includes the specification of the `Hook` and `Signal` classes. They provide the behaviour of a system of telephone (users) which can communicate in pairs. There is no (theoretical) bound on the number of users. The validation of `POTS` was much more complicated than the validation of the `Telephone` class. The O-LSTS interface diagram of `POTS` is shown in figure 5. Note that the internal state decomposition does not match with the external accessor services: the external view is by user whilst the internal view is predominantly by user pairs. This is typical in object oriented specifications where the user interface abstracts away from much more complex internal details (much like the way in which the telephones on a real network abstract away from the underlying network complexity).

The structure of the `POTS` class is of interest. The first parameter represents the set of user pairs who are talking to each other. The second parameter represents the set of user pairs where the first is off hook and hearing a ringing sound and the second is on hook and hearing a ringing sound. The first has called the second and the second has yet to answer (even though they are free to do so). The third parameter represents the set of user pairs (caller and callee) where the caller is receiving a busy signal because the callee is not available. The fourth parameter is the set of telephones which are currently on hook. The final parameter is the set of all user (identifiers).

Clearly the internal state representation of the POTS system is in some ways arbitrary. In object oriented terms, it is only the state which can be seen through the external (accessor) operators which is important. The state composition above was chosen for its extensibility and simplicity.

4.3 Validation

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

Testing also increased understanding of the problem domain. For example, we discovered many case scenarios which were incorrectly specified (in our original models and pre-exisiting models of POTS). Testing was simply a means of validating that desirable scenarios were allowed by our ACT ONE model, and that undesirable scenarios were forbidden. The tests were carried out by hand using the lite toolset. We believe that we will need to create new tools which hide the underlying LOTOS and present behaviour at a level of semantics more appropriate to the communication of requirements in the problem domain.

The validation was carried out by completely checking all possible states in our state transition system requirements models. The new state of the system after a given transformation is specified by an opertion. Term rewriting of ACT ONE expressions is used as the operational semantics for our requirements model. The new state of a system (after a transformation) is validated through the appliation of accessor operations. This is illustrated below for the `Telphone` class.

4.3.1 Telephone Validation

The following code was generated by the lite tool set. It completely validates the telephone requirements model:

```
state !U0 !On !silent
— I(dialIn) ?UID1-0:UID —— state !U0 !On !ringing
— lift —— state !U0 !Off !ready
state !U0 !Off !ready
— drop —— state !U0 !On !silent
— dial ?UID1-1:UID —— state !U0 !Off !silent
state !U0 !Off !silent
— drop —— state !U0 !On !silent
— I(otherFree) —— state !U0 !Off !ringing
— I(otherBusy) —— state !U0 !Off !busy
state !U0 !Off !ringing
— drop —— state !U0 !On !silent
— I(changeHook) —— state !U0 !Off !talking
state !U0 !Off !busy
— drop —— state !U0 !On !silent
state !U0 !Off !talking
— drop —— state !U0 !On !silent
— I(changeHook) —— state !U0 !Off !busy
state !U0 !On !ringing
— lift —— state !U0 !Off !talking
— I(changeHook) —— state !U0 !On !silent
```

It should be noted that these tests were carried out on the process algebra specification of the `Telephone` behaviour which was generated as the initial object oriented design. The process algebra simply wraps the functional behaviour specified by the ACT ONE in a communication shell (where gates correspond to services). This process algebra code is automatically generated from the ACT ONE and it aids the testing of the ACT ONE behaviour model.

4.3.2 POTS Validation

The validation of the POTS object acts as a good example of how to completely test a system with a potentially infinite number of states. Clearly, the number of states in the system grows exponentially with respect to the number of telephones. Every telphone in the system is in one of six states (the state where the telephone is off and silent cannot occur in POTS because the system knows immediately if the other phone being called is busy or free and therefore the callee never enters the intermediate off-silent state). The other six states (as seen in the telephone specification) can be read through the external accessors of the POTS class.

Given that the POTS system can have any number of users then how do we validate its behaviour with the customer. The simple answer is to start by validating a system with one user, then a system of two users and then a system of three users, etc .... Then, we can reach a point (n-users) where the addition of another user does not add any further complexity to the observable behaviour. Thus the system is completely validated. (A proof by induction could be carried out if we have some sort of meta-language for validation. However, this is beyond the scope of our work.) The question is: what is the n-value for the number of users in the case of POTS. The answer is three.

Given a system of POTS with one user then (from that user's point of view) the addition of another user can increase the number of states that this user can be in. For example, the user cannot talk unless there is someone else to talk to. Clearly, then, we must test with more than one user.

Given a POTS with only two users, there are some states which we can test which we cannot test with one user. However, two users is not enough because if we add another user then (from the original user's point of view) there are some states which they can (collectively) be in when there is a third user which they cannot reach by themselves. For example, user1 can be on and ringing whilst user2 can be off and busy. This is not possible unless a third user is dialling user1. Clearly, then, we must validate POTS with at least three users. A simple analysis, however, shows that there is no three user view of the POTS system which cannot be reached with three users alone. Thus a complete validation of POTS can be done by validating POTS with three users. We should also note that in POTS we can add users dynamically as behaviour progresses. However, this does not change the state of the existing telephones in the system and so we do not need to test this dynamic addition. Instead, we test POTS with a static number of users (namely three).

Given a 3-user POTS, we have a finite state machine of 216 states to validate (with approximately 4 transitions from each state). This is more than can simply be done by hand (even with use of the lite tool set). To simplify the task, we identify symmetry in the state model. Clearly, the state of a 3-user POTS is a permutation of the state of any three telephones. Thus, we reduce the number of states we have to validate to 56. Finally, we note that the state invariant of POTS can be used to reduce this number even further. For example, the invariant states (amongst other things) that the number of users talking is always even. We are left with 16 states and 37 transitions to validate. This is much more manageable and was done quite quickly using the available tools.

Validating the Invariant

The use of an invariant to reduce the state space being tested is dependent on two things. Firstly, we must prove that all initial states (in this case there is just one) satisfy the invariant. Secondly, we must prove that all state transformations are closed with respect to the invariant. This is easily done in the POTS ACT ONE specification by ensuring that no exception values occur in a complete trace of behaviour.

4.4 POTS System Design

Given the (validated) ACT ONE specifications of the `POTS` and `Telephone` classes, we move forward to the design of a parallel system of telephones. This is done using the process algebra part of LOTOS, together with the ACT ONE requirements model. The first step is to generate full LOTOS specifications of `POTS` and `Telephone` processes. The functionality (state changes) of the system is maintained directly through the ADT specification. The way in which a system offers this functionality through its interface is defined by the process algebra part. A simple remote-procedure-call semantics for service requests is chosen in the initial designs, for simplicity.

```
PROCESS Telephone[drop, lift, dial, listen, offHook, dialIn, otherChangeHook, otherBusy, other-
Free] (Telephone1: Telephone): noexit:=
hide otherChangeHook, otherBusy, otherFree in

([enabled(drop(Telephone1))] -> drop!ID(Telephone1);
Telephone[drop, lift, ..., otherFree] (drop(Telephone1))
)[]
([enabled(lift(Telephone1))] -> lift!ID(Telephone1);
Telephone[drop, lift, ..., otherFree] (lift(Telephone1))
)[]
(dial?UID1:UID!ID(Telephone1)[enabled(dial(Telephone1,UID1))];
Telephone[drop, lift, ..., otherFree] (dial(Telephone1,UID1))
)[]
(listen!ID(Telephone1); listen!listen(Telephone1)!ID(Telephone1);
Telephone[drop, lift, ..., otherFree] (Telephone1)
)[]
(offHook!ID(Telephone1); offHook! offHook(Telephone1)!ID(Telephone1);
Telephone[drop, lift, ..., otherFree] (Telephone1)
)[]
(dialIn!ID(Telephone1)?UID1:UID[enabled(dialIn(Telephone1,UID1))];
Telephone[drop, lift, ..., otherFree] (dialIn(Telephone1, UID1))
)[]
([enabled(otherChangeHook(Telephone1))] -> otherChangeHook!ID(Telephone1);
Telephone[drop, lift, ..., otherFree] (otherChangeHook(Telephone1))
)[]
([enabled(otherFree(Telephone1))] -> otherFree!ID(Telephone1);
Telephone[drop, lift, ..., otherFree] (otherFree(Telephone1))
)[]
([enabled(otherBusy(Telephone1))] -> otherBusy!ID(Telephone1);
Telephone[drop, lift, ..., otherFree] (otherBusy(Telephone1))
) endproc (* Telephone *)
```

The enabled operation (common to all OO ACT ONE specifications) can now be used to guarantee that events occur only when they are possible (for example, a user cannot lift a phone which is already off hook). It should also be noted that internal (nondeterministic) transitions are now hidden from the external user of the telephone object.

The initial LOTOS `POTS` design follows the same pattern as that for the `telephone`:

PROCESS POTS[dial, drop, lift, addU, stateUhook, stateUsignal, whoRinging, whoTalking]
(POTS1: POTS): **noexit** :=
(dial? UID1:UID? UID2:UID [enabled(dial(POTS1, strUPair(UID1,UID2)))];
POTS[dial, ..., whoTalking] (dial(POTS1, strUPair(UID1,UID2)))
)[]
(drop? UID1:UID[enabled(drop(POTS1,UID1))];
POTS[dial, ..., whoTalking] (drop(POTS1, UID1))
)[]
(lift? UID1:UID[enabled(lift(POTS1,UID1))];
POTS[dial, ..., whoTalking] (lift(POTS1, UID1))
)[]
(addU? UID1:UID[enabled(addU(POTS1,UID1))];
POTS[dial, ..., whoTalking] (addU(POTS1, UID1))
)[]
(stateUsignal?UID1:UID; stateUsignal!stateUsignal(POTS1,UID1);
POTS[dial, ..., whoTalking] (POTS1)
)[]
(stateUhook?UID1:UID; stateUhook!stateUhook(POTS1,UID1);
POTS[dial, ..., whoTalking] (POTS1)
)[]
(whoRinging?UID1:UID; whoRinging!whoRinging(POTS1,UID1);
POTS[dial, ..., whoTalking] (POTS1)
)[]
(whoTalking?UID1:UID; whoTalking!whoTalking(POTS1,UID1);
POTS[dial, ..., whoTalking] (POTS1)
) **endproc** (* POTS *)

These objects are now to be incorporated in a LOTOS design of a distributed parallel POTS system. The process specifications of these components are maintained throughout the whole design process. The final telephone network structure is shown in figure 6.

The telphone and POTS processes are used as specified in the initial design. A new control process is used to organise the communication between the central POTS database and the network of telephones. Each telephone is hidden behind a telephone interface which is used to control the multi-way synchronisation of the LOTOS process algebra. Unfortunately, in LOTOS the number of gates is static in a given specification and so we cannot create new gates as we create new telephone processes. Hence, we need to have all telephones synchronise on the same internal `t`gates when they communicate with the control. This is achieved by using the interface processes to participate in all events, but to ignore those which are not specifically targetted at its particular phone.

4.5 Verification

Given the LOTOS specification, we must now verify it against the initial requirements model. This is done by running the same validation test cases through our smile simulator. In this case, we must

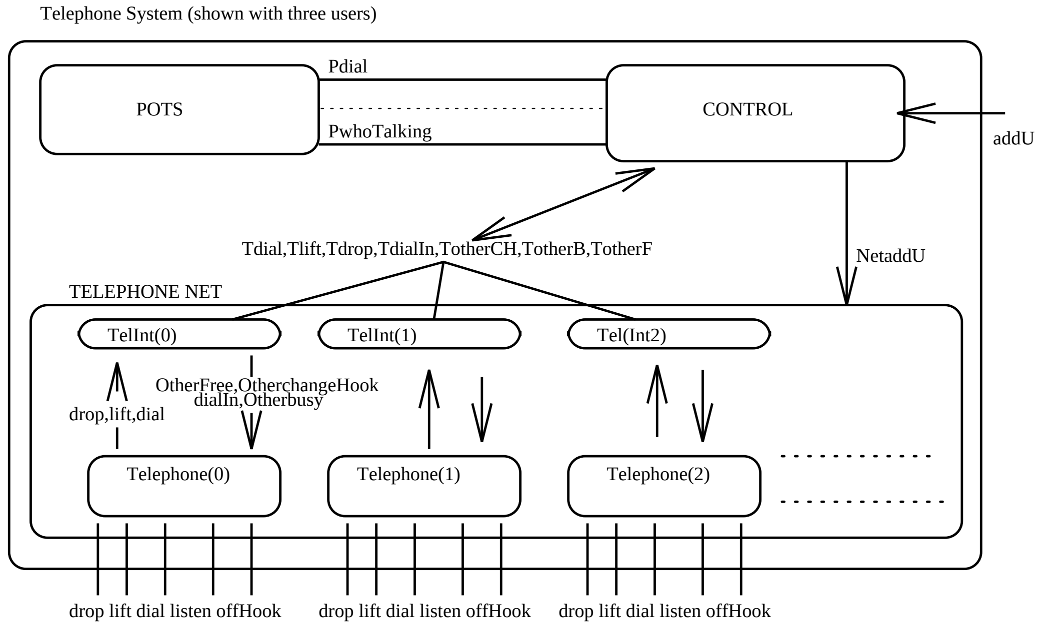

Figure 6: Telephone System High Level Design

also attempt to test different resolutions of internal events. During system execution, we often have a choice between a set of internal events. The way in which such a choice is made should not alter the external correctness of the system with respect to observable behaviour. The formal verification of this was not carried out as part of our research. However, it should be noted that because we have a formal requirements model (and a formal design model) we should (in theory) be able to prove correctness. This is for future work.

4.6 Adding Features

The suitablity of the architectures for the addition of features is now examined. Two features were examined: call waiting and three way calling. We do not go into any details of these features because we are not yet happy with their specifications. We have mangaged to add the features (and validate) their behaviour without too much trouble. However,we have not yet mnaged to do this without changing the internal structure of our system. The main reason for this is that we have not managed to find a suitable object conceptualisation for a feature. Once this is developed then it should be possible to plug together features in a way similar to which we put together POTS and the network of telephones. This is our current area of research.

Call Waiting

Call waiting (Cw) is a feature whereby a customer in the talking state is alerted by a call waiting tone when another caller is attempted to call him, The feature allows the customer (with the service activated) to alternaly talk to the original and newly calling parts. If the customer with the service activated hangs up while one party is on hold, the customer with the service activated is either automatically rung back and upon answer is connected to the held party or the held party gets the busy signal.

Three way calling

The Three way calling (Twc) is a feature that allows the customer in the talking state to add a third way party to the call. To add a third party to the call, the Twc customer place the other party on hold and dials the number of the third party and then the Twc connection will be established. If either of the other two parties hangs up while the service activating customer remains offhook, the Twc is returned to a two party connection between the remaining party.

5 Conclusion: A Feature Oriented LOTOS Architecture

We wish to be able to construct systems of telephone services where features can be requested (and disposed of) dynamically by telephone users, and where new features can be added by telephone service providers. We believe that an object oriented LOTOS framework may be a step towards this goal. We hope to classify different categories of feature and provide mathematical theoerems for the way in which categories interact (a kind of feature interaction meta-analysis). Based on this theory, we hope to provide high level construction mechanisms (defined using OO LOTOS) which can be used to build systems from feature objects. Then, we have an architecture which is feature oriented.

LOTOS is a suitable specification language for capturing the behaviour of a complex system as a collection of interacting concurrent objects. There is a correspondence between objects and processes, and between message passing and event synchronisation. This helped to incorporate the structure of the specification in the design. A consequence of this was an obvious relationship between the different stages of the development process. The LOTOS specification acts as a formal design. It not only specifies the requirements of the system, but it also provides a framework within which the implementation can be built.

The advantage of a consistent specification style is the ability to structure specifications in such a way that design approach can be explicitly stated. Complex architectures, in particular, require a consistent structured approach to aid comprehension. The object oriented LOTOS style seems ideal because of the way it allows for the modelling of systems as interacting parts, each of which can have a straightforward mapping onto real world implementation entities.

Inheritance

The OO LOTOS work provides inheritance mechanisms based on the notion of subclasses as valid replacements. Subclassing in the POTS problem domain depends on our ability to identify classification relationships between groups of services that share specific properties. We have reason to hope that features will fall into a finite set of *feature classes* and that we will be able to formally identify properties which will be useful in the construction of new features from those features that are classified appropriately. Thus we will have sets of abstract feature classes and a high level of analysis based on the abstract behaviours. Thus feature interaction can be dealt in a generalised manner. Features and (combinations of features) will inherit properties that will be useful for validation and verification.

Before inheritance can be used at all stages of the development process, it is important not only that inheritance relationships exist but also that they can be exploited. Libraries of components need to be created and inheritance needs to be used as naturally as any other language construct. High level tools create a *meta language* which is customer oriented in the sense that the fundamental building blocks are those seen in the problem domain (rather than those seen in the specification language semantics). This is a great advantage.

OO LOTOS Deficiencies

LOTOS does not provide us with semantics for the fair resolution of nondeterminism. Continuing work shows the need for such *fairness* in the specification of other features (and feature combinations). LOTOS has not got great tool support for the validation and verification of telephone services. This tools are all language dependent in the sense that they are based on the low level semantic constructs. We require a tool which is more problem domain oriented (though which may be based on underlying LOTOS semantics).

References

[1] P. America. Object oriented programming: a theoretician's introduction. Technical report, Philip's Research Laboratories, Eindhoven, 1988.

[2] B. Biebow and J. Hagelstein. Algebraic specification of synchronisation and erros: A telephoni c example. *Lecture Notes in Computer Science*, 154:1–14, 1983.

[3] S. Black. Objects and LOTOS. Technical report, Hewlett-Packard Laboratories, Stoke Gifford, Bristol, 1989.

[4] D.G. Bobrow, K. Kahn, G. Kiczales, L. Masinter, Stefik M., and Zdybel F. Commonloops: Merging Lisp and object-oriented programming. In *ACM SIGPLAN Notices*, pages 17–29, November 1986.

[5] T. Bolognesi and E. Brinksma. Introduction to the ISO specification language LOTOS. *Computer Networks and ISDN Systems*, 14:25–59, 1987.

[6] T. Bolognesi and M. Caneve. SQUIGGLES: A tool for the analysis of LOTOS specifications. In K.J.T. Turner, editor, *The 2nd International Conference on Formal Description Techniques (FORTE 89)*, 1989.

[7] G. Booch. *Object Oriented Development.* IEE Software Engineering, February 1986.

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

[9] R. Boumezbeur. *Design, Specification and Validation of Telephony Systems in LOTO.* PhD thesis, University of Ottawa, 1991.

[10] R. Breu. *Algebraic Specification Techniques in Object Oriented Programming Environments.* Springer-Verlag, 1991. Lecture Notes in Computing Science, number 562.

[11] E. Brinksma and Scollo G. Formal notions of implementation and conformance in lotos. Mem: INT-86-13, University of Twente, NL, December 1986.

[12] H. I. Cannon. Flavours. Technical report, MIT International Laboratories, Cambridge (Mass), 1980.

[13] Robert Clark. Using LOTOS in the object based development of embedded systems. In *The Unified Computation Laboratory.* The Institute of Mathematics and its Applications (OUP), 1991.

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

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

[16] W. Cook, W. Hill, and P. Canning. Inheritance is not subtyping. In *Proceeedings of 19th ACM Symposium on Principles of Programming Systems and Languages*, pages 125–135, 1989.

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

[18] E. Cusack. Formal object oriented specification of distributed systems. In *Specification and Verification of Concurrent Systems*, University of Stirling, 1988.

[19] L. G. DeMichiel and R. P. Gabriel. The Common Lisp Object System: an overview. In J. Bezivin, J. Hullot, P. Cointe, and H. Lieberman, editors, *Proceeedings of the 1987 European Conference on Object Oriented Programming (ECOOP 87)*, pages 151–170. Springer-Verlag, June 1987.

[20] T. B. Dinesh. *Object-Oriented Programming: Inheritance to Adoption.* PhD thesis, University of Iowa, 1992.

[21] H. Ehrig and Mahr B. *Fundamentals of Algebraic Specification I.* Springer-Verlag, Berlin, 1985. EATCS Monographs on Theoretical Computer Science (6).

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

[23] J.Paul Gibson and Lynch J.A. Applying formal object oriented design principles to Smalltalk-80. *British Telecom Technology Journal*, 3, July 1989.

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

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

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

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

[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] I.Tvrdy. Formal modelling of telematic services in lotos. *Microprocessing and Microprogramming*, 25:313–318, 1989.

[30] M.A. Jackson. *System Development.* Prentice-Hall, 1983.

[31] S. E. Keene. *Object-oriented programming in Common Lisp: a programmer's guide to CLOS.* Addison-Wesley, 1989.

[32] M. Lai and E. Cusack. Object oriented specification in LOTOS and Z or, my cat really is object oriented. In de Bakker, J. W. et al., editors, *Proc. Foundations of Object Oriented Languages*, pages 179–202. Springer Verlag, 1991. Lecture Notes in Computer Science, Number 489.

[33] K. Lee, S. Rudkin, and K. Chon. Specification of a sieve object in objective LOTOS. Technical report, British Telecom Research Laboratories (Formal Methods Group), St. Vincent House, Ipswich, 1990.

[34] T. Mayr. Specification of object oriented systems in LOTOS. In *The 1st International Conference on Formal Description Techniques (FORTE 88)*, 1988.

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

[36] L.Logrippo M.Faci and B. Stepien. Formal specification of telephone systems in lotos : constraint-oriented style approach. *Computer Networks and ISDN Systems*, 21:53–67, 1991.

[37] R. Milner. *A Calculus of Communicating Systems.* Springer-Verlag, 1980.

[38] D. A. Moon. Object-oriented programming with Flavours. In *ACM SIGPLAN Notices*, pages 1–8, November 1986.

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

[40] K. Ohmaki, K. Futatsugi, and K. Takahashi. A basic LOTOS simulator in OBJ. Computer Language Section, Computer Science Division, Electrotechnical Laboratory, 1-1-4 Umezono, Japan, Draft Report, 1990.

[41] K. Orr. *Structured Systems Development.* Yourdon Press, 1977.

[42] M. Papathomas. *Language Design Rationale and Semantic Framework for Concurrent Object-Oriented Programming.* PhD thesis, University of Geneva, 1992.

[43] K. Raymond, P. Stocks, and D. Carrington. Specifying ODP systems in Z. Technical report, University of Queensland, March 1990.

[44] S. Rudkin. Inheritance in LOTOS. In K. Parker and G. Rose, editors, *Formal Description Techniques IV.* North-Holland, 1991.

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

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

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

[48] Kenneth J. Turner. The formal specification language LOTOS: A course for users. Technical report, Department of Computing Science, University of Stirling, 1990.

[49] van Eijk, Vissers, and Diaz. *The Formal Description Technique LOTOS.* North-Holland, Amsterdam, 1989.

[50] W.H.P. van Hulzen. Object oriented specification style in LOTOS. Lo/wp1/t1.1/rnl/n00002, LOTOSPHERE, 1989.

[51] Jose Luis Vivas. *Design of Telephony Services in LOTOS.* PhD thesis, Uppsala Univeristy, 1994.

[52] D. Walker. π calculus semantics of object-oriented programming languages. Technical Report ECS-LFCS-90-122, Computer Science Department, Edinburgh University, Laboratory for Foundations of Computer Science, October 1990.

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

[54] Mario Wolczko. *Semantics of Object-Oriented Languages.* PhD thesis, University of Manchester, 1988.

[55] P. Yelland. First steps towards fully abstract semantics for object oriented languages. In S. Cook, editor, *Proceeedings of the 1989 European Conference on Object Oriented Programming (ECOOP 89)*, pages 347–367. Cambridge University Press (on behalf of British Computer Society), 1989.