Animating formal specifications: a telephone simulation case study

Jean-Paul Gibson, Dominique Méry, Yassine Mokhtari

To cite this version:


HAL Id: inria-00108114
https://inria.hal.science/inria-00108114
Submitted on 19 Oct 2006

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
ANIMATING FORMAL SPECIFICATIONS: A TELEPHONE SIMULATION CASE STUDY

J. Paul Gibson
Dep. of Computer Science
NUI Maynooth, Ireland
E-mail: pgibson@cs.may.ie

Dominique Méry and Yassine Mokhtari
LORIA, Université Henri Poincaré
BP 239, 54506 Vandœuvre-lès-Nancy
E-mail: {mery,mokhtari}@loria.fr

KEYWORDS
LOTOS, OO Analysis, software architecture

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 rigorous 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.

INTRODUCTION
The problem of feature interactions in telephonic systems is well documented (Zave, 1993; Cameron et al., 1994; Bouma and Velthuijsen, 1994; Cheng and Ohta, 1995). Formal languages have been used in the development of such systems to improve the means of analysing requirements models for undesirable behaviour (Rochefort and Hoover, 1997; Prehofer, 1997; Blom, 1997; Gibson, 1997; Turner, 1997; Veldhuijsen, 1995; Gibson and Méry, 1997). 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.

OBJECT ORIENTED FRAMEWORKS FOR REQUIREMENTS CAPTURE AND DESIGN
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\(^1\). Emphasis on data structure led to the encapsulation of functional behaviour within data entities: \textit{objects}\(^2\). The principles which make object oriented approaches successful 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 (albeit, 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 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

---

\(^1\)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.

\(^2\)Two well known data-based software development methods which are generally accepted as not being object oriented are the quite similar approaches put forward by Jackson (Jackson, 1983) and Orr (Orr, 1977). 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.
- **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.

There has been much interest in combining formal and object oriented methods. Object oriented methods can be used to aid the construction of formal methods (Clark, 1991; Black, 1989; Cusack, 1988; Lai and Cusack, 1991; Rudkin, 1991; van Hulzen, 1989; Gibson and J.A., 1989; Raymond et al., 1990; Mayr, 1988; Lee et al., 1990). Formality can help to improve object oriented software development techniques (Wegner, 1987; AMERICA, 1988; Wolczko, 1988; Breu, 1991; Yelland, 1989; Papathomas, 1992; Walker, 1990; Dinesh, 1992).

The FOOD (Formal Object Oriented Development) approach of Gibson (Gibson, 1993) is based on these theoretical foundations and has also proven itself in the specification of service specifications in an industrial telecommunications environment. For that reason, it is chosen as our method for the development of our LOTOS POTS specifications. In particular, the author presents an Object-Labelled Transition System (O-LSTS) model which can be used at different levels of software development (from analysis and capture requirements to design and implementation).

### OO ACT ONE: Requirements Models

O-LSTS specifications (class specifications) are written in OO ACT ONE (a language whose syntax is similar to the ADT ACT ONE). The translation of O-LSTS requirements model to the abstract data typing language ACT ONE provides an executable (abstract) model for customer validation. The object oriented method encourages the recording of certain structural aspects of the problem domain. 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.

### OO LOTOS: Design Models

The step from analysis 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 OO ACT ONE requirements model into a full LOTOS Design and this can be done in many ways (Gibson, 1993). This initial step is used to reason in terms of higher level constructs such as communicating processes. This is examined in more detail when we show the LOTOS POTS design.

- **POTS SPECIFICATION**

**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. 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 use 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 simultaneous connections. The possibility for several users to be engaged at the same time in one connection is considered as an additional feature. A feature is some service offered to meet some communication need. Features are the fundamental building blocks for telephonic communication.

**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.

**Telephone class**

The state transition diagram for the Telephone class is illustrated in figure 1. It is written in OO ACT ONE and translated to ACT ONE code (see appendix).

**POTS**

In a similar fashion we can define the interface specification of the POTS class, which is depicted in figure 2. 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 2. 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.

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-existing 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 operation. Term rewriting of ACT ONE expressions is used as the operational semantics for our requirements model. The new state of a system (after a transformation) is validated through the application of accessor operations. This is illustrated below for the Telephone class.

Telephone Validation

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

```
state !U0 'On 'silent
— drop — state !U0 'On 'talking
— I(dialIn) ?UID1-0:UID —— state !U0 'On 'ringing
— lift — state !U0 'Off 'ready
dia!UID — state !U0 'On 'silent
— dial ?UID1-1:UID —— 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
— I(changeHook) — state !U0 'Off 'talking
— drop — state !U0 'Off 'talking
— state !U0 'On 'silent
— state !U0 'On 'silent
— I(changeHook) — state !U0 'Off 'talking
— state !U0 'Off 'talking
— state !U0 'On 'silent
— state !U0 'On 'silent
— I(changeHook) — state !U0 'Off 'talking
— 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 automat-
ically generated from the ACT ONE and it aids the testing of the ACT ONE behaviour model.

**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 telephone 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.

**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 (see appendix). The initial Lotos **POTS** design follows the same pattern as that for the telephone.

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 3.

The telephone 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 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.

**CONCLUSION**

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 theorems 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.

**REFERENCES**


Blom, J. 1997. Formalisation of requirements with emphasis on feature interaction detection. In Feature Interactions In Telecommunications IV, Montreal, Canada. IOS Press.


### Telephone class

**type** Telephone is UserID, POTS sorts Telephone

**ops**

- **NewTelephone**: UID -> Telephone (+ INITIALISER *)
- **strTel**: UID, Hook, Signal -> Telephone (+ STRUCTURE *)
- **listen**: Telephone -> Signal (+ ACCESSOR *)
- **otherBusy,otherFree**: Telephone -> Telephone (+ INTERNAL *)
- **otherChangeHook**: Telephone -> Telephone (+ INTERNAL *)
- **drop, lift**: Telephone -> Telephone (+ TRANSFORMER *)
- **dial**: Telephone, UID -> Telephone (+ TRANSFORMER *)
- **on, off**: Telephone, UID -> Telephone (+ TRANSFORMER *)
- **tel1**: Telephone, Hook -> Telephone (+ TRANSFORMER *)
- **otherFree**, **listen**, **drop**, **talk**, **hook1**, **hook2**, **signal1**, **signal2**, **on, off**: Signal
- **eqns for all**: VID1, VID2: Telephone, hook1,hook2 : Hook, signal1,signal2 : Signal
- **ofsor**: Bool
- **not** ((listen(tel1) eq tal) and not(offhook(tel1))) = true;
- **telephone**: Telephone (* TRANSFORMER *)
- **otherBusy, otherFree**: Telephone (* INTERNAL *)

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 non-deterministically 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 Telephone sort. These are as follows:

```
ecns for all VID1, VID2: Telephone, hook1, hook2: Hook, signal1, signal2: Signal
ofsor: Bool
    offHook(strTel(VID1, hook1, signal1)) = hook1 eq off;
    enabled(TelephoneEXC) = false;
    enabled(strTel(VID1, hook1, signal1)) = true;
ofsor: Signal
    listen(strTel(VID1, hook1, signal1)) = signal1;
ofsor: VID1
    ID(strTel(VID1, hook1, signal1)) = VID1;
ofsor: Telephone
    NewTelephone(VID1) = strTel(VID1, 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(VID1, hook1, signal1)) = TelephoneEXC;
    hook1 eq off => drop(strTel(VID1, hook1, signal1)) = strTel(VID1, on, sil);
    (* LIFT *)
    hook1 eq off =>
    lift(strTel(VID1, hook1, signal1)) = TelephoneEXC;
    (hook1 eq on) and (signal1 eq sil) =>
    lift(strTel(VID1, hook1, signal1)) = strTel(VID1, off, rea);
```
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.

- **Set**
  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.

- **USet**
  This class is a set of identifiers 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.

### Telephone Design

```haskell
PROCESS Telephone(drop, lift, dial, listen, offHook, dialIn, otherChangeHook, otherBusy, otherFree) (Telephone1: Telephone): noexit =
  hide otherChangeHook, otherBusy, otherFree in

  (notEmpty(strTel(UID1, hook1, signal1)) = strTel(UID1, strTel, tab);
  (* 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, sli);
  hook1 eq off =>
  dialIn(strTel(UID1, hook1, signal1), UID2) = TelephoneEXC;
  (hook1 eq on) and (not(signal1 eq sli)) =>
  dialIn(strTel(UID1, hook1, signal1), UID2) = strTel(UID1, on, rin);

  (* OTHERBUSY, OTHERFREE, OTHERCHANGEHOOK ... Similarly *)
end type (* 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.

### POTS System Design

```haskell
PROCESS POTS[dial, drop, lift, dial, listen, offHook, dialIn, otherChangeHook, otherBusy, otherFree] (POTS1: POTS): noexit =
  (drop? UID1: UID [enabled(drop(POTS1, UID1))];
  Telephone(dial, lift, ..., otherFree) (drop(Telephone1, UID1)));

  (listen? UID1: UID (listen(POTS1) | UID1);
  Telephone(dial, lift, ..., otherFree) (drop(Telephone1, UID1)));

  (offHook? UID1: UID (offHook(POTS1) | UID1);
  Telephone(dial, lift, ..., otherFree) (drop(Telephone1, UID1)));

  (dialIn? UID1: UID (dialIn(POTS1) | UID1);
  Telephone(dial, lift, ..., otherFree) (dialIn(Telephone1, UID1)));

  (whoTalking? UID1: UID (whoTalking(POTS1) | UID1);
  Telephone(dial, lift, ..., otherFree) (otherChangeHook(Telephone1)));

  (whoRinging? UID1: UID (whoRinging(POTS1) | UID1);
  Telephone(dial, lift, ..., otherFree) (otherBusy(Telephone1)));

  (whoTalking? UID1: UID (whoTalking(POTS1) | UID1);
  Telephone(dial, lift, ..., otherFree) (otherBusy(Telephone1)));

POTS System Design
```