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 M´ery, 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: *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.

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.

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.

OO Development

Design must be targetted towards a particular implementation. Within the POTS development and for 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. Consequently, this suggests that there is a closer binding between the specification and implementation architectures. We refer to a specification as being structured only if the decomposition of the problem is explicit. The advantages and the drawbacks of the structured specifications are approached in more detail in (Gibson and Mokhtari, 1998).

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 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 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 figure2. 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

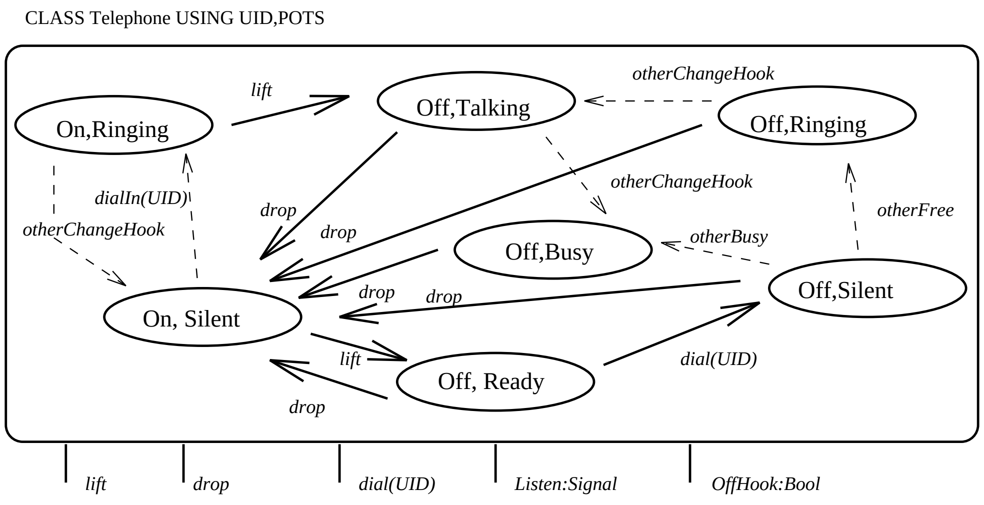

Figure 1: Phone O-LSTS Interface Diagram

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
— 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 automat-

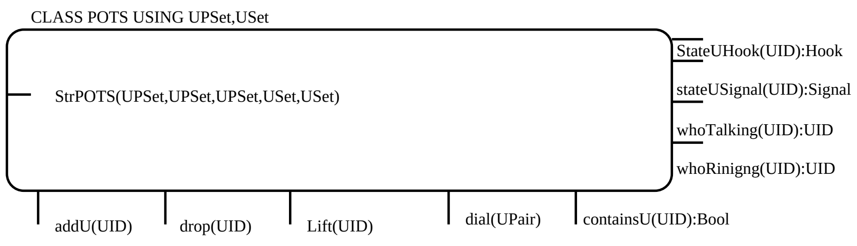

Figure 2: POTS O-LSTS Interface Diagram

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

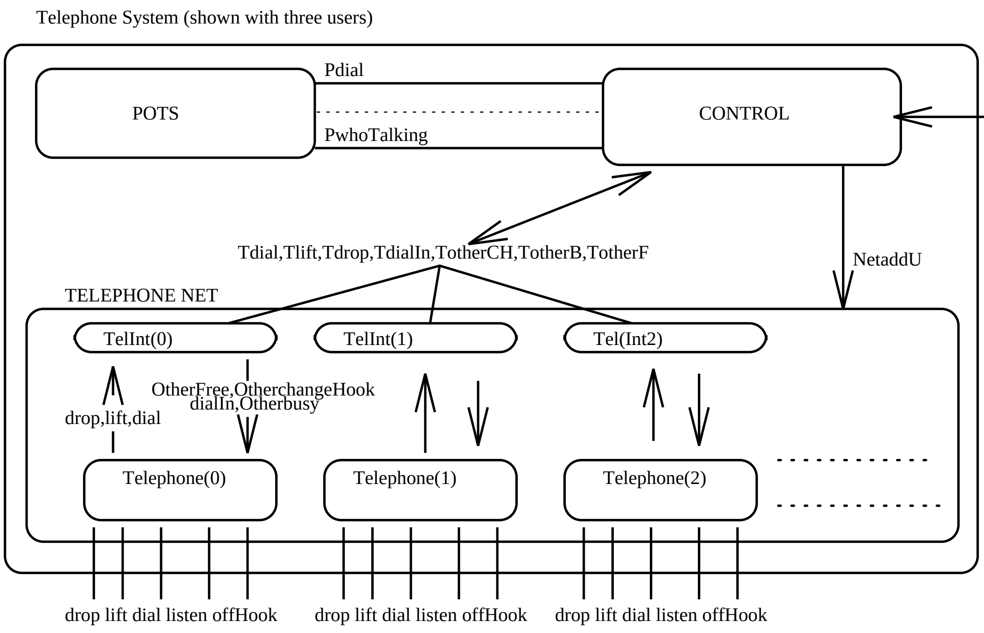
Telephone System (shown with three users)
POTS
Pdial
PwhoTalking
CONTROL
addU
Tdial,Tlift,Tdrop,TdialIn,TotherCH,TotherB,TotherF
NetaddU
TELEPHONE NET
TelInt(0)
TelInt(1)
Tel(Int2)
drop,lift,dial
OtherFree,OtherchangeHook
dialIn,Otherbusy
Telephone(0)
Telephone(1)
Telephone(2)
drop lift dial listen offHook
drop lift dial listen offHook
drop lift dial listen offHook

Figure 3: Telephone System High Level Design

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

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

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

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

Bouma, L. G. and Velthuijsen, H., editors (1994). *Feature Interactions In Telecommunications*. IOS Press.

Breu, R. 1991. *Algebraic Specifi cation Techniques in Object Oriented Programming Environments*. Springer-Verlag. Lecture Notes in Computing Science, number 562.

Cameron, E. J., Griffeth, N. D., Lin, Y., Nilson, M. E., and Schnure, W. K. 1994. A feature interaction benchmark for in and beyond. In *Feature Interactions In Telecommunications*.

Cheng, K. E. and Ohta, T., editors (1995). *Feature Interactions In Telecommunications III*. IOS Press.

Clark, R. 1991. Using LOTOS in the object based development of embedded systems. In *The Unifi ed Computation Laboratory*. The Institute of Mathematics and its Applications (OUP).

Cusack, E. 1988. Formal object oriented specifi cation of distributed systems. In *Specifi cation and Verifi cation of Concurrent Systems*, University of Stirling.

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

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

Gibson, J. and J.A., L. (1989). Applying formal object oriented design principles to Smalltalk-80. *British Telecom Technology Journal*, 3.

Gibson, J. P. 1997. Feature requirements models: Understanding interactions. In *Feature Interactions In Telecommunications IV*, Montreal, Canada. IOS Press.

Gibson, M. and M´ery 1997. Feature interactions: A mixed semantic model approach. In *Irish Workshop on Formal Methods*, Dublin, Ireland.

Gibson, P. and Mokhtari, Y. 1998. Pots: An OO LOTOS specification. Technical Report CRIN-98-R-013, CRIN.

Jackson, M. 1983. *System Development*. Prentice-Hall.

Lai, M. and Cusack, E. 1991. 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. Lecture Notes in Computer Science, Number 489.

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

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

Orr, K. 1977. *Structured Systems Development*. Yourdon Press.

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

Prehofer, H. 1997. An object oriented approach to feature interaction. In *Feature Interactions In Telecommunications IV*, Montreal, Canada. IOS Press.

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

Rochefort and Hoover 1997. An exercise in using constructive proof systems to address feature interactions. In *Feature Interactions In Telecommunications IV*, Montreal, Canada. IOS Press.

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

Turner, K. 1997. An architectural foundation fo relating features. In *Feature Interactions In Telecommunications IV*, Montreal, Canada. IOS Press.

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

Veldhuijsen, H. 1995. Issues of non-montonicity in feature interaction detection. In *Feature Interactions In Telecommunications III*.

Walker, D. 1990. 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.

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

Wolczko, M. 1988. *Semantics of Object-Oriented Languages*. PhD thesis, University of Manchester.

Yelland, P. 1989. First steps towards fully abstract semantics for object oriented languages. In Cook, S., 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).

Zave, P. 1993. Feature interactions and formal specifications in telecommunications. *IEEE Computer Magazine*, pages 18–23.

APPENDIX

Telephone class

```
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);
(* 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 *)
```

POTS class

```
type POTS is UPSet, USet sorts POTS, Hook, Signal, HS, STist
opns
NewPOTS:-> POTS (* INITIALISER *)
strPOTS: UPSet,UPSet,UPSet,USet,USet -> POTS (* STRUC-
TURE *)
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 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 identifi cations 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 identifi ers.
- **UPSet**
  This class is a set os identifi er 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

```
PROCESS Telephone[drop, lift, dial, listen, offHook, dialIn,
otherChangeHook, otherBusy, otherFree] (Telephone1: Tele-
phone): 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); lis-
ten!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))] -> otherChange-
Hook!ID(Telephone1);
Telephone[drop, lift, . . . , otherFree] (otherChange-
Hook(Telephone1))
)[]
([enabled(otherFree(Telephone1))] -> other-
Free!ID(Telephone1);
Telephone[drop, lift, . . . , otherFree] (otherFree(Telephone1))
)[]
([enabled(otherBusy(Telephone1))] -> other-
Busy!ID(Telephone1);
Telephone[drop, lift, . . . , otherFree] (otherBusy(Telephone1))
) endproc (* Telephone *)
```

The enabled operation (common to all OO ACT ONE specifi cations) 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

```
PROCESS POTS[dial, drop, lift, addU, stateUhook, stateUsig-
nal, whoRinging, whoTalking] (POTS1: POTS): noexit :=
(dial? UID1:UID? UID2:UID [enabled(dial(POTS1, strU-
Pair(UID1,UID2)))];
POTS[dial, . . . , whoTalking] (dial(POTS1, strU-
Pair(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; stateUsig-
nal!stateUsignal(POTS1,UID1);
POTS[dial, . . . , whoTalking] (POTS1)
)[]
(stateUhook?UID1:UID; sta-
teUhook!stateUhook(POTS1,UID1);
POTS[dial, . . . , whoTalking] (POTS1)
)[]
(whoRinging?UID1:UID; whoRing-
ing!whoRinging(POTS1,UID1);
POTS[dial, . . . , whoTalking] (POTS1)
)[]
(whoTalking?UID1:UID; whoTalk-
ing!whoTalking(POTS1,UID1);
POTS[dial, . . . , whoTalking] (POTS1)
) endproc (* POTS *)
```