Formal requirements models: simulation, validation and verification

J. Paul Gibson
Department of Computer Science,
National University of Ireland, Maynooth

Date: February 2001

Technical Report: NUIM-CS-2001-TR-02

Key words: method, synthesis, analysis, object oriented, correctness

Abstract

Requirements models have three distinct roles — they are the principle media of communication between clients and requirements engineers, they are the only model upon which rigorous and automated analysis can be carried out before development begins, and they are the structural foundation upon which design and implementation depend. A major part of building requirements is the modelling of the system to be developed (or updated) together with the system environment. These models are, of course, abstractions of the real world and their operational semantics can be executed to provide a simulation model for validation: to show that the behaviour specified actually corresponds to what exists or what is required. The models also have to be verified to show their logical consistency. A formal object oriented method facilitates incremental development, where the integration of simulation (through animation), theorem proving and model checking increases confidence in model correctness.

Formal requirements models: simulation, validation and verification

Abstract

Requirements models have three distinct roles — they are the principle media of communication between clients and requirements engineers, they are the only model upon which rigorous and automated analysis can be carried out before development begins, and they are the structural foundation upon which design and implementation depend. A major part of building requirements is the modelling of the system to be developed (or updated) together with the system environment. These models are, of course, abstractions of the real world and their operational semantics can be executed to provide a simulation model for validation: to show that the behaviour specified actually corresponds to what exists or what is required. The models also have to be verified to show their logical consistency. A formal object oriented method facilitates incremental development, where the integration of simulation (through animation), theorem proving and model checking increases confidence in model correctness.

1 Introduction

Simulation is fundamental for the analysis of complex systems, including models of customer requirements. Requirements modelling is concerned with synthesising and analysing the abstract requirements of a client: the *what*, not the *how*. Requirements models are naturally decomposed into two parts: the model of the system to be built and the model of the system environment. Often, an implementation architecture exists such that new requirements must be built onto an already developed system. In this case it is very important that a *correct* simulation (abstraction) of the already existing system is incorporated into the requirements model. Of course, if this system was originally developed using a formal method then a specification of the system, which has already been validated, could be re-used for this purpose; and its integration could be formally verified. New requirements need to be validated — the client has to be willing to accept that the model actually represents their needs. The requirements also have to be verified — to show the logical consistency of the different needs (both old and new) and different points of view. The process of requirements engineering continually improves our models until the *best* abstraction of the client's needs is reached and design can begin to transform the *what* into the *how*.

This paper reports on a formal object oriented method for incrementally constructing, validating and verifying requirements models. Simulation plays an important role in our method as it allows us to perform analysis — animation, validation and verification — in a constructive manner. The remaining sections of the paper are as follows. Section 2 comments on our choice of a formal object oriented method. Section 3 introduces our formal semantics, based on object oriented concepts, which forms the basis of our modelling language. Section 4 details the importance of non-determinism in the models, and the complementary triangle between simulation, validation and verification. Section 5 comments on the tools which we use during synthesis and analysis, and the need for an integrated environment. Section 6 shows how our method has been successfully applied in the domain of telephone service development. Section 7 concludes.

2 Formal object oriented requirements models

2.1 The importance of requirements engineering

Analysis is the process of maximising *problem domain understanding*. Only through complete understanding can an analyst comprehend the responsiblities of a system. The modelling of these responsiblities is a natural way of expressing system requirements. The simplest way for an analyst to increase understanding is through interaction with the customer, where one of the most common problems is that an interelated set of requirements must be incorporated into one coherent and consistent framework. Interaction with the customer is an example of informal communication. It is an important part of analysis and, although it cannot be formalised, it is possible to add rigour to the process. A well-defined analysis method can help the communication process by reducing the amount of information an analyst needs to assimilate. By stating the type of information that is useful, it is possible to structure the communication process. Effective analysis for building requirements models is dependent on knowing the sort of information that is required, extracting it from the customer, and recording it in some coherent fashion. In other words, requirements capture and analysis is concerned with simulation of client's needs through abstraction.

2.2 Requirements Models — integrating different needs

The requirements model is important as it acts as the communication medium through which the client, analyst and developers can improve their mutual understanding of the client's requirements. The client understands their needs from an abstract view point which hides the *how* of the system to be developed. They have operational requirements which are usually expressed as sequences of actions (or events) which they would (or would not) like to be possible when they use the system. They also have logical requirements based on *always* and *eventually* concepts [22] — they require some things to be true always and these must be expressed as safety properties; and they require that some things must eventually happen and these must be expressed as liveness properties. The designer must be able to understand the abstract needs of the client and transform these needs into an implementation. The requirements model should act as a contract between the client and the developer. It should also be possible to verify that an implementation is correct with respect to the customer's requirements. This is the role of the designer. The analyst must help the customer to construct and validate their requirements. Furthermore, it is the responsibility of the analyst to verify that the operational requirements are consistent with the logical requirements. After validation, it is the analyst who acts as the principle interface between the designers and the requirements models.

A major problem exists in convincing each of these three different groups of the utility of the requirements models. In our method, we emphasise the need for client-oriented models — if the client cannot understand the requirements then validation cannot be done correctly and the rest of the development process is compromised. When in doubt, the best rule is to let the client's understanding of their needs provide the underlying structure of the requirements model.

2.3 Why formalise?

Formal methods are a tool for achieving *correct* software: that is, software that can be proven to fulfil its requirements. Formal specifications are unambiguous and analysable [29]. Building a formal model improves understanding [21]. The modelling of nondeterminism, and its subsequent removal in formal steps, allows design and implementation decisions to be made when most suitable. Nondeterminism is also the key to validating ones simulation models.

We advocate the use of formal methods in the building of requirements models. Through formal models, re-use can be controlled at all levels of abstraction and the client can be more confident that their requirements are truly met by the implementations. There are four important aspects to the use of formal methods for requirements capture:

- The method must be compositional so that incremental development is supported. Furthermore, the method must support high-level structuring mechanisms which correspond to the way in which the client structures their understanding of their needs. In fact, we propose following a formal object oriented approach [19].
- The method must offer a means of specifying operational requirements for animation during validation. The requirements models which we use correspond to compositional state transition systems and object oriented structuring mechanisms such as extension, specialisation, delegation, subclassing and inheritance are provided by a formal semantics which define a correspondance between state machines and objects [21].
- The method must support multiple views on a system so that the client can control execution of a subset of system behaviour whilst a simulator controls the other parts in a manner which corresponds to how the system behaves, or should behave, in the real world.
- The method must offer a means of specifying logical requirements. A purely operational view allows only the specification of safety properties — *bad things can never happen.* We

also require a means of specifying liveness properties which state that *something good will eventually happen.*

Three types of formal models are used. Firstly, we have an executable model (written in LOTOS [23] using an object-based style [15]) which is useful for compositional animation. Secondly, we have a logical model (based on the B method) which is used to verify the state invariant properties of our system (statically). Finally, we use TLA [26] to provide semantics for a static analysis of liveness and fairness properties. No one model can treat each of these aspects, yet each of these aspects of the conceptualisation are necessary in the synthesis and analysis of formal requirements models. We are in the process of integrating the different semantics into one coherent model [14, 18, 20, 22].

2.4 Why object oriented?

We advocate an object oriented approach to structuring our requirements models. Object oriented methods encompass a set of techniques which have been, and will continue to be, applied in the successful production of complex software systems [7, 8, 4, 28]. The methods are based on the simple mathematical models of abstraction, classification, refinement and polymorphism. Central to the success of object oriented techniques is the support they offer to re-use at all levels of abstraction. Re-use and structure are just as important during requirements capture as during implementation.

Structure is fundamental to all stages of system development: it provides the framework upon which already developed parts of a system can be re-used. Structured analysis and requirements capture methods have been successfully applied in many different problem domains during the last twenty years[9, 11, 12]. It is clear that there is a symbiotic relationship between structure and re-use: *classification* facilitates re-use of abstractions and relations between abstract behaviours, *composition* facilitates re-use of concrete behaviour, *refinement* facilitates re-use of verification and validation, *configuration* facilitates re-use of composition mechanisms. The key to building good requirements models is to model understanding as structure and to provide facility for structural re-use.

Object orientation supports an incremental approach where requirements are continually changed, and animated step-by-step. We support four main types of increment —

- **Subclassing**: An already specified class can be used as the abstract superclass of a new subclass. The subclass must, when working in our formal object oriented framework, exhibit all the properties of the superclass, and so this can be re-used during validation and verification of the new behaviour.

- **Delegation**: An already specified class can be used as a component of a new class. The behaviour of the old class is encapsulated behind a well-defined interface and, again, we can re-use our understanding of the old class in the validation and verification of the new class.
- **Co-operation**: Two, or more, already specified classes can be configured in order to define the required behaviour of a new class. Our method formalises such configuration through the use of invariants which act as a means of glueing together the components in a way which guarantees correctness.
- **Structure re-use**: As understanding of the problem domain increases due to the continually improving requirements model, it is often the case that the client gains some insight into their problem which allows them to re-structure their understanding. In this case, our object oriented method provides a means of transforming the structure of the original model in a localised manner.

3 Semantic Framework

Labelled state transition systems are often used to provide executable models during analysis, design and implementation stages of software development [9, 11, 12]. In particular, such models are found in the classic analysis and design methods of [4, 8, 10]. However, a major problem with state models is that it can be difficult to provide a good system (de)composition when the underlying state and state transitions are not easily conceptualised. The object oriented paradigm provides a natural solution to this problem. By equating the notion of class with the state transition system model and allowing the state of one class to be defined as a composition of states of other classes, the O-LSTS approach provides a means of specifying such models in a constructive fashion.

The O-LSTS semantics also permit us to view objects at different levels of abstraction. Firstly, using an abstract data type (ADT) we can specify the functionality of an object at a level of abstraction suitable for requirements capture[27]. Secondly, we can transform our ADT requirements into a parameterised process algebra (LOTOS[3, 30]) specification where we consider how the system can interact with its environment. Finally, we consider modelling the underlying implementation environment, where we can view the objects in our designs as clients and servers in a distributed, concurrent network.

3.1 ADT for global system validaion

The simplest way to introduce the ADT view is through a *standard* example: A `Queue` of `Integers` is specified using the OO ACT ONE specification language from [21].

```
CLASS Queue USING Integer
      LITERALS empty
      STRUCTURES Aqueue(Queue, Integer)
      ACCESSORS is-empty:Bool
      TRANSFORMERS push(Integer)
      DUALS pop:Integer
      EQUATIONS
            empty.is-empty = True; Aqueue(Q,I).is-empty=False;
            Q.push(I)=Aqueue(Q,I);
            empty.pop = empty RETURNS EXCEPTION;
            Aqueue(Q,I).pop=Q RETURNS I
 ENDCLASS Queue
```

The Queue uses a predefined class Integer (which itself uses Bool). The literal and structure members define all the possible (states of the) objects in the class. There is one non-structured literal value empty. All other elements are structured from two components, namely a Queue and an Integer, using the AQueue operator. The class interface is defined by the three different sets of services offered: a transformer changes the internal state of an object, an accessor returns some value without changing internal state, a dual returns a value and may also change the internal state. The services may be parameterised (e.g. push) to represent the input data passed to the server object when a service is requested. The equations are used to define a semantics for the interface services. It should be noted that every class has, by default, an implicit EXCEPTION value which can be used in cases like the popping of an element from an empty queue. Furthermore, variables which are not typed explicitly may have their types inferred where there is no ambiguity. We also have tools for verifying the completeness and consistency of such specifications.

Consider a system made up of two Queues (as specified above) whose behaviour is given by the OO ACT ONE below. The TwoQs provides two services: push (which pushes elements onto the first queue component) and pop (which pops elements from the second queue component); and the internal state transition move transfers elements from the first queue onto the second queue.

```
CLASS TwoQs USING Queue
      STRUCTURES QQ(Queue, Queue)
      TRANSFORMERS push(Integer)
      DUALS pop:Integer
      INTERNAL move
      EQUATIONS
            QQ(Q1,Q2).push(I) = QQ(Q1.push(I),Q2);
            QQ(Q1,Q2).move = QQ(Q1.pop(),Q2.push(Q1.pop()));
            QQ(Q1,Q2).pop = QQ(Q1,Q2.pop()) RETURNS Q2.pop();
```

```
ENDCLASS TwoQs
```

The internal transitions, like `move`, represent nondeterminism in the requirements model which needs to be resolved through implementation or, when we come to validate our models, made more concrete through simulation.

Using only the ADT specification, we can validate the whole system through animation, where the user is required to take control over all the actions in the system and this assumes that they have global understanding. This is fine for small systems (like `TwoQs`) but is not realistic when we have large, complex systems of many components. In such cases, we have to permit compositional validation. A first step towards this is to formally specify the way in which components communicate and interact. For this we use a process algebra.

3.2 Process Algebra for communication validation

The first step is to specify how the complete system interacts with its environment. This is done by wrapping the ADT specification inside a process specification. The ADT part of the LOTOS specification, not shown in the code below, is generated automatically from the OO ACT ONE specification, and is used to parameterise the process definition for the corresponding behaviour. In this case, type `TwoQs` parameterises the behaviour of process `TwoQs`.

```
PROCESS TwoQs[push, pop](QQ:TwoQs):  NOEXIT:=
      HIDE move IN
      ( push?  Integer1:Integer; TwoQs[...](push(QQ,Integer1))
      )[]
      ( pop; pop!  popRESULT(QQ); TwoQs[...](pop(QQ))
      )[]
      ( move; TwoQs[...](move(QQ)) )
ENDPROC (* TwoQs *)
```

This particular LOTOS design, chosen for its simplicity, specifies that a *remote procedure call* protocol is used for communication with the `TwoQs` process. (Other types of protocol can also be generated automatically.) The `push` operation is carried out synchronously between the object and its environment. The `pop` operator requires some result to be returned and we model the communication of the result as an event different from the service request. The `move` operation is hidden from the environment of the `TwoQs` process: as such, the movement of elements between the two queues cannot be determined by the `TwoQs` client(s).

3.3 Process algebra structure for compositional validation

It is now possible, using a pre-defined correctness preserving transformation, to re-use the compositional structure found in the OO ACT ONE specification in a structurally equivalent compositional LOTOS specification. This results in an equivalent specification with two `Queue` processes synchronising on an internal `move` event. We illustrate this below, in a partial specification.

```
PROCESS TwoQs[push, pop](QQ(Q1, Q2)):  NOEXIT:=
 HIDE move, pop1, push2 IN
Queue[push, pop1](Q1)
|[pop1]|
Control[move, pop1, push2]
|[push2]|
Queue[push2, pop](Q2)
where ...ENDPROC (* TwoQs *)
```

In this specification, we now have a control process which models the internal movement of elements between the queue processes. Using the structure in this specification, we can now carry out our validation in a compositional manner. We can validate any single process component (or sets of components) through simulation of the other components in the system. For example, perhaps the client wants only to validate the behaviour of the input queue. In our animation we would provide only control of the `push` and `pop` operations of the input queue; all other operations would be controlled by a simulator. In a second case, the designer may wish only to validate that the movement of messages is being correctly modelled by the control process. Here, the animator would have to simulate the pushing of messages onto the system and the popping of messages off the system (in a realistic manner).

3.4 Client-Server View and eventuality requirements

The type of LOTOS design seen above is quite close to the type of *client-server* model that is found in many reference models for software development, see [25, 24], for example. We can say that the environment of a `TwoQs` process is its client. Now let us consider a *liveness* property which we would reasonably require such a system to fulfil. The nondeterministic `move` operation cannot be guaranteed to be carried out when we specify only *safety* properties. We may require that if an element is pushed onto the first queue, then it will eventually be moved to the second queue. We do not wish to specify how this happens, only that it does. This is the essence of abstraction with regards to the nondeterminism in our system: we need to be able to specify *fairness* at the 'class level of abstraction'.

Furthermore, we need to consider what happens when a server has multiple (concurrent) clients. If an object in the server's environment requests a `push` how can we be sure that it will be carried out even though, in this case, it is always enabled? The problem is as follows: if the server is shared between other clients then how do we guarantee that one client's requests will eventually be carried out? Certainly, we could specify some sort of queueing protocol for simulation purposes. In essence, we use the TLA to verify properties of our simulations.

4 Simulation, validation and verification

4.1 Validation and verification

It is important to understand the difference between validation and verification: the first is is about checking that a formal model correctly captures the client's needs, and the second is about checking that a formal model meets the requirements of another formal model. We can verify the consistency of a requirements model by showing that it's operational requirements meet its logical requirements, or that two sets of logical requirements are not contradictory. This is not validation; it is, however, complementary to validation — it should not be possible for a client to validate a model which is logically inconsistent (i.e. impossible to implement). Such a situation arises out of contradictory requirements and is often seen when requirements are extended independently. (The feature interaction problem [16] is a good example.)

4.2 Nondeterminism and simulation

Communication between a (sub)system and its environment can be modelled nondeterministically. For example, with the `move` transition between queues we are forced to simulate the `moves` in order to validate the system. Temporal logic can be used to prove that the simulation is correct with respect to certain temporal constraints.

4.3 Animation, validation and simulation

The key to our validation is the operational object oriented semantics which, through graphical animation, provide support for customers to validate their understanding of their requirements, rather than validating their understanding of the models. However, when the validation involves simulation of part of the internal parts of the system then there is a problem if we cannot be sure that the simulation is correct: if the simulation is wrong then the customer has correctly validated the external behaviour of the system but wrongly validated the internal parts. In other words, they are happy with the behaviour specified but make wrong assumptions about how it will be implemented. For this reason, we also need to consider validation of our simulation models.

4.4 Verification, theorem proving, model checking and simulation

As explained earlier, we have to be able to verify the logical consistency of our requirements —

- **Invariants:** These are defined in all structured classes. We use a theorem prover [1] to show that all the operations of the class are closed with respect to its invariants. For example, to prove that integers are never lost in the move between queues, we specify that the number of elements in queue1 and the number of elements in queue2 is *always* the same as the number popped off subtracted from the number pushed on.

- **Fairness:** Using TLP [13], the TLA theorem prover, we are able to prove eventuality properties. For example, by specifying fairness on the movement of elements (moves will always eventually take place if there is something to move) then we can prove that any element pushed on will *eventually* arrive at the second queue.

- **Algebraic Composition:** In [17], we introduced the notion of re-usable analysis techniques based on performing analysis on abstract superclasses and abstract composition mechanisms. For example, we can prove that if we replace the move operation by two moves around a thrid queue then we do not change the abstract behaviour of our system.

4.5 Simulation validation

We have noted that we had to simulate parts of the system in order to validate other parts. In this way we can test products and services before we have to make changes to the underlying implementation architecture, and before providing the customer with a working model. In the parts of the system being simulated, there is a fundamental difficulty in trying to compare the behaviour of the simualtion model and the behaviour of the reality being modelled. In our formal approach, we can specify logical properties that are obtained through observation (or analysis) of the real world, and verify that these properties are consistent with our simulations.

5 Tool Integration

5.1 Why integrate?

It is important that a development method is supported by a suite of tools for synthesis and analysis which share a common formalism and a common interface. Our development framework includes six different tools:

- An animator for validating the behaviour with the client
- A simulator for control non-determinism during animation
- A prover for verifying the logical consistency of the requirements
- A model checker for testing the properties that the prover cannot verify automatically
- A library of classes, graphical mappings, validations, proofs and test suites
- A development manager for incremental refinement of requirements.

This integration is important because it is counter-productive to try and separate these aspects. A model checker is an automated animation process where all possible sequences of actions are tested. A model checker with user control over choice of actions provides the same functionality as an animator. A model checker with statistical control over nondeterminism resolution provides the same functionality as a simulator. Furthermore, animation can verify logical properties dynamically, provers often animate in order to identify critical cases, and incremental development involves refinement of both logical and operational properties.

5.2 Why Graphical Animation?

Graphical views have long been used to represent large quantities of information in a simple and concise form. Humans have evolved a very complex mechanism for collecting and colating information that is presented graphically. Understanding the information depends on clarity of expression which, in turn, relies on meaningful structure. Graphical models can provide both these properties. Graphical views are prominent at all stages of software development because of their ability to convey structural aspects of a system.

All standard software visual models are particular types of graph — each model attaches meaning to the labelling of nodes and links and the relationship defined between connected nodes. Categorisation of graphical models is simply a grouping together of models in which the meaning attached to the views shares some commonality. It is precisely the meaning attached to graphical views which distinguishes different models.

The underlying modelling language (semantic basis) is a major influence on the structure of a visualisation environment. Because the environment manipulates components in the language, this directly influences the environment's structure and form, though not necessarily its presentation to the user. A visual representation must be able to naturally model a conceptual system with the minimum amount of mental transfer and mapping on the part of the modeller (or viewer). We advocate an approach in which the fundamental modelling blocks are objects and classes.

There are three distinct modes of operation in our method:

- **Visualisation** is the process by which mappings are defined between formally specified models and graphical constructs.
- **Synthesis** is the creation of new classes of behaviour and re-use of already existing classes. Synthesis mechanisms utilise the visual mappings and may even be defined in terms of visual manipulations.
- **Analysis** is the feedback step. The development of requirements models is an evolutionary process. Initially, there will be many problems which will gradually be removed by customer and analyst. Analysis can be improved through the use of visual mappings and graphical animations.

Animation is the visual analysis of the dynamic properties of the requirements models. In our method we encourage *experimentation*, where the client animates many different test scenarios for any given model.

5.3 Experimentation

Experimentation is the phase that follows the construction of a new requirements model. The purpose of experimentation is to learn more about the system under study by subjecting its model to various interaction sequences selected from ligitimate inputs. The process of constructing experiments is itself a modelling activity: one builds a model (or models) of the environment of the system being analysed. This can be done in an ad-hoc fashion by the viewer subjectively selecting interactions during each cycle of the animation. We must also provide facility for a more planned creation of experiments which permit the controlled exercise of the system through different simulation scenarios. There are a number of important aspects to experimentation:

- *Full animation vs Statistics Gathering*
  The experiment, together with the system model, may be executed without interaction from the viewer. This auto-animation can either be presented to the viewer *as-if* they were involved in the visual interactions. Contrastingly, the viewer may not wish a full animation to be presented. In many cases the animation process is being used to check a set of predefined properties or for the purpose of gathering statistics. We offer each of these facilities.
- *What vs How*
  Experimentation, as a closed model, can present the behaviour of a system as a black box

— the internal state of the system can be abstracted away from and only the sequence of interactions need to be presented for analysis. In other words, the analysis is concerned only with *what* the system is doing at its external interface. This type of black box testing is fine in requirements models which are complete. However, whilst the modelling process is continually refining both *what* is being specified and *how* it is being specified, it is important that the viewer can choose to see different internal properties of the model in question.

- *Nondeterminism*
  During requirements capture, the modeller often wishes to specify nondeterministic behaviour. There is flexibility (during animation) in choosing random number generators or using a pre-defined algorithm or data file for simulating this nondeterminism..

5.4 Library as language: the future ideal

We believe that the future of our method depends on the notion of *library as language*. One of the keys to the success of object oriented programming languages is the way in which new programmers can learn the language in a problem specific way, through use of libraries of classes. Each programmer must understand the fundamental concepts and language constructs, but the class libraries then act as the language extensions. Often, object oriented programmers are expert in certain problem domains and this corresponds to the libraries with which they are familiar. Requirements capture techniques should, we believe, offer the same advantages. The client should be able to build models using their own language and this can be achieved by the analyst creating libraries of re-usable classes which are client-oriented. These libraries then define the vocabulary of the problem domain being modelled.

6 An industrial case study: telephone service development

6.1 Problem Overview

The complexity of standard telephone behaviour is growing exponentially due to the number of services (or features) available. The feature interaction problem occurs when two or more features, whose individual behaviours are easy to specify and validate with the client, introduce unforseen problems when they are asked to work together (see [5, 6] for a wide range of papers on the subject). Formal methods have been proposed as a means of controlling the complex analysis required for the detection and resolution of these problems [2]. It is well accepted that these formal techniques should be applied as early as possible in the development process. Thus we have a need for formal requirements models [16, 31].

6.2 Informal requirements models

Intuitively, we can see that a telephone user may wish to express different types of requirements:

- **Safety requirements** — where the user specifies things that must never happen. These can be state based, sequence based or property based. A state based safety requirement corresponds to the user never wanting to be in a certain concrete state (e.g. My answering machine should never take a message from a FAX). A sequence based safety requirement corresponds to the user never witnessing a sequence of external actions (e.g. putting the phone `on-hook` , lifting the phone `off-hook` and then hearing a `busy signal`). A property based safety requirement corresponds to a state based requirement where the state is specified abstractly over a number of possible states which are not explicitly listed (e.g. never wanting to pay for an overseas call).

- **Liveness requirements** — where the user specifies that something good will eventually happen. These usually correspond to different types of fairness or eventuality needs. For example, my answering machine should eventually take a message if I don't reply. Eventuality requirements are common in user-oriented specifications because they help to abstract away from the network.

- **Nondeterministic and consistency requirements** — where the user specifies a number of different behaviours which would be acceptable and may require that the nondeterminism be resolved in a consistent fashion. For example, I don't mind whether my answering machine or my FAX get priority so long as the priority cannot change unexpectedly.

- **Compositional requirements** — where the user specifies new needs by 'combining' already existing services. For example, a user with an `answering machine` and `call hold` may wish to let a held caller leave a message while they are waiting, and thus give them the option of abandoning the call if they have to wait too long.

- **Specialisation and extension requirements** — where the user specifies new needs by making refinements to already existing services. For example, a user may require an answering machine which restricts the length of messages that can be left.

In the following sections, we comment on the modellingof the *plain old telephone service* (POTS) and the verification and validation of POTS requirements.

6.3 The formal POTS models

Consider the (partial) OO ACT ONE specification of the POTS phone, where we list only a subset of the equations:

```
    CLASS POTS USING Number, Signal, Hook
          STRUCTURES phone(Signal, Hook)
          ACCESSORS listen:Signal, on-off:  Hook
          TRANSFORMERS lift, drop, dial(Number)
          INTERNAL otherbusy, otherfree, otherhook, otherdialIn, noline
          EQUATIONS
                phone(ringing, on).lift() = phone(talking, off);
                phone(talking, off).drop() = phone(silent, on);
                phone(talking, off).otherhook() = phone(noline, off);
...ENDCLASS POTS
```

The telephone system is then composed from a number of these phones, together with a model of the underlying network. We note that there is much more nondeterminism in the model and therefore much more work for the simulator to do during animation.

6.4 Validation and verification

We have text-based animators for validating the operational requirements in the composed system. We also have JAVA mappings from textual specifications to graphical representations. Current work is concerned with integrating these mappings into the animation tools in order to provide graphical animations. Validation of the telephone models has been done using a mixture of text and graphics. The animators have been used in three distinct ways:

- We use the formal specifications of a network and a number of phones (in practice, three phones are sufficient for validation of the simple POTS with no additional features) in a composition which simulated a telephone system. All nondeterminism was resolved internally and the analysts, designers and clients could watch *random* animations as a means of validating the integration of the network and telephone models.

- We had client-led animations where the system was partly controlled by the network simulator and partly controlled by the client. The client could control any number of phones and the animator controlled the remaining part of the system simulation.

- We had designer-led animations where all the phones were controlled by the simulator and the designer chose how to resolve the nondeterminism in the network.

In all cases animation was done in parallel with verification of *always* and *eventually* properties. In fact, animation was also a good tool for improving the understanding of the proof process.

7 Conclusions

Simulation is a tool which can help to locate errors earlier in the development process and reduce costs and compress schedules. Our formal models — at all levels of abstraction — are fully machine-intelligible and machine interrogable. The compositional validation techniques make it much easier to locate modelling errors which is crucial when dealing with complex systems.

We advocate a mixed-semantic approach to requirements engineering. Only through formal methods can integration of different client's needs be verified. Only through graphical animation can clients be expected to validate complex models. The quality of the validation depends on the quality of the model of the system to be developed *and* the quality of the model of the environment of this system. These models can be used as simulations in order to facilitate different abstractions on the same complete system. Simulation, validation and verification are complementary aspects of requirements capture.

References

[1] B-core. B-Toolkit User's Manual, Release 3.2. Technical report, B-core, 1996.

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

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

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

[5] L. G. Bouma and H. Velthuijsen, editors. *Feature Interactions In Telecommunications.* IOS Press, 1994.

[6] K. E. Cheng and T. Ohta, editors. *Feature Interactions In Telecommunications III.* IOS Press, 1995.

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

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

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

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

[11] Geoff Cutts. *Structured system analysis and design method.* Blackwell Scientific Publishers, 1991.

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

[13] U. Engberg. *TLP Manual-(release 2. 5a)*-PRELIMINARY. Department of Computer Science, Aarhus University, May 1994.

[14] J.-P. Gibson and D. Méry. A Unifying Model for Specification and Design. Rapport Interne CRIN-96-R-110, CRIN, Linz (Austria), July 1996.

[15] J. Paul Gibson. Formal object based design in LOTOS. Tr-113, University of Stirling, Computing Science Department, Stirling, Scotland, 1994.

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

[17] J. Paul Gibson. Towards a feature interaction algebra. In *Feature Interactions In Telecommunications V*, pages 217–231, Lund,Sweden, September 1998. IOS Press.

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

[19] J. Paul Gibson, Bruno Mermet, and Dominique Méry. Specification of services in a compositional temporal logic. Rapport de fin du lot1 du marche no 961B CNET-CNRS CRIN, CRIN, 1997.

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

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

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

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

[24] ISO/IEC. Specification styles for structuring of OSI formal descriptions. ISO/IEC JTC1/SC21/N669, International Organisation for Standardisation, 1989.

[25] ISO/IEC. Working document on topic 6.2 - formalisms and specifications. information retrieval, transfer and management for osi. ISO/IEC-JTC1/SC21/WG7, International Organisation for Standardisation, 1989.

[26] L. Lamport. A temporal logic of actions. Technical Report 57, DEC Palo Alto, april 1990.

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

[28] B. Meyer. Re-usability: the case for object oriented design. *IEE Software Engineering*, March 1987.

[29] K.J.T. Turner. *Using FDTS: An Introduction To ESTELLE, LOTOS and SDL*. John Wiley and Sons, 1993.

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

[31] Pamela Zave. Feature interactions and formal specifications in telecommunications. *IEEE Computer Magazine*, pages 18–23, August 1993.