Formal object oriented requirements: simulation, validation and verification

J. Paul Gibson,
Computer Science Department,
NUI Maynooth, Ireland.
pgibson@cs.may.ie

Abstract

Requirements engineering is the first step in the software engineering process. 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 as such we can say that they are simulations which need to be validated to show that they actually correspond to what exists or what is required. They also have to be verified to show their consistency.

Requirements models have 3 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. We advocate a formal object oriented approach which can be presented in a client-friendly manner, using graphical representations.

The overall theme of this paper is the triangle of integration in our simulations — we integrate user-friendly (graphical) animation of operational requirements during validation, together with proof of logical properties during verification, together with the structural object oriented concepts which support formal incremental development techniques.

1 Introduction

Simulation is concerned with constructing an abstract model of a real system. 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. These two models can be integrated into a single abstraction representing the interface between the system and its 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 simulations 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 the simulation models which correspond to the user requirements. Graphical animation of the models is central to our method.

2 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. The customer may be one person, in which case the process is greatly simplified; however, it is more likely that the customer is a group of clients, each with their own particular needs. One of the main problems in dealing with a set of customers is that the interelated set of requirements must be incorporated into one coherent and consistent framework. Each client must be able to validate his (or her) own needs irrespective of the other clients (unless of course these needs are contradictory).

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 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.

3 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. There are three different points of view:

- 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 (Gibson & Méry 1998*a*) — 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.

In this paper, we review how our formal object oriented development method integrates these different view points.

3.1 Why formalise?

Formal methods are necessary in achieving *correct* software: that is, software that can be proven to fulfil its requirements. Formal specifications are unambiguous and analysable (Turner 1993). Building a formal model improves understanding (Gibson 1993). The modelling of nondeterminism, and its subsequent removal in formal steps, allows design and implementation decisions to be made when most suitable. *Correctness preserving transformations* facilitate the automatic generation of more efficent code whilst guaranteeing the preservation of original behaviour. Formal models are amenable to mathematical manipulation and reasoning, and facilitate rigorous testing procedures.

We advocate the use of formal methods in the building of requirements models. Only through formal models can re-use be controlled at all levels of abstraction. Furthermore, only through formal techniques can the client be sure that their requirements are truly met by the implementations. There are three 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 (P. Gibson & Méry 1997).
- 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 (Gibson 1993).
- 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*. We adopt a mixed semantic model (P. Gibson & Méry 1997), based on an integration of LOTOS (Bolognesi & Brinksma 1987), TLA+ (Lamport 1995) and B (Abrial 1996), which presents user friendly graphical views on the different abstraction mechanisms (Gibson & Méry 1998*b*).

3.2 Why graphical?

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.

3.3 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 (Coad & Yourdon 1990*a*, Coad & Yourdon 1990*b*, Booch 1991, Meyer 1987). 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(Constantine 1989, Cutts 1991, DeMarco 1979). 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. Finally, a re-usable structure is often known as an architecture. The key to building good requirements models is to model understanding as structure and to provide facility for structural re-use.

3.4 Method integration

The four most important parts of our development method depend on formal specification techniques:

- Firstly, we have an executable model which is useful for validating the dynamic behaviour.
- Secondly, we have a logical model for specifying less operational properties that can be validated statically.
- Thirdly, we have a formal verification that the executable model fulfils the requirements of the logical model.
- Finally, we promote incremental development whereby each step can be compositionally validated and/or verified through re-use of static and dynamic analysis.

4 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 (Gibson 1997) is a good example which we will return to in section 7.)

4.1 Refinement and Validation

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. In an ideal environment the analyst would identify the set of concepts with which the customer understands their needs, map these concepts onto the formal semantics of one (or all) of our modelling languages and let the customer construct their own requirements model. In practice it is more feasible to expect the customer and analyst to work together during the construction and refinement of requirements.

4.2 Incremental Animation

We advocate 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.

4.3 Verification and theorem proving

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 (B-core 1996) to show that all the operations of the class are closed with respect to its invariants.
- **Fairness:** Using TLP (Engberg 1994), the TLA theorem prover, we are able to prove eventuality properties.
- **Algebraic Composition:** In (Gibson 1998), we introduced the notion of re-usable analysis techniques based on performing analysis on abstract superclasses and abstract composition mechanisms.

5 Our tools

Our development framework integrates three different tools:

- An animator for validating the behaviour with the client
- A prover for verifying the logical consistency of the requirements
- A development manager for incremental refinement of requirements.

This integration is important because it is counter-productive to try and separate these three aspects: 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.

5.1 Client-orientation

Given a flexibility in the way in which different structures can be used to specify the same requirements, it is often difficult to judge which structure is *best*. 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. If the analyst sees a better way of structuring the requirements then it is up to them to explain it to the client and get their acceptance that it is indeed an improvement.

5.2 Visual Abstraction - the importance of mappings

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 reuse 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 Interface design

The creation of user defined mappings is very much related to the concept of user interface design. Extra burden is placed on the analyst to provide mappings which are acceptable to the user. However, this extra work does lead to the development of rapid prototypes in which the interface of the system has been given as much consideration as the functionality which it offers.

6 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.

The starting point for the construction or analysis of a requirements model is the library of predefined classes. In general, it is useful to be able to distinguish between:

- Generally Applicable Library Classes — behaviour which is well tried and tested and useful in a wide range of problem domains. Such classes are the fundamental building blocks for all modellers.
- Problem Domain Library Classes — behaviour which is well tried and tested and useful within particular problem domains. Modellers will be knowledgeable about only a subset of these libraries.
- System Development Libraries — behaviour which is currently under development.

In any particular problem domain, modelling is concerned with the class specifications and the class mappings. The client can re-use default mappings of the general classes or choose to define their own mappings. Most problem domain library classes will contain routines for parameterised mappings which allow the client to choose between different graphical representations. Finally, part of the process of developing new classes is in developing default mappings for graphical representation of these classes.

7 An industrial case study: feature interactions

7.1 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 (Bouma & Velthuijsen 1994, Cheng & Ohta 1995) 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 (Blom 1997). 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 (Gibson 1997, Zave 1993).

7.2 Informal requirements models

Intuitively, we can see that a client often wishes 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.

Our method supports the specification, integration, validation and verification of all these different requirement types (Gibson 1998).

7.3 Integrating our formal requirements models

Three types of formal models are used. Firstly, we have an executable model (written in LOTOS (Guillemot, Haj-Hussein & Logroppo 1991) using an object-based style (Gibson 1994)) 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 (Lamport 1990) 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 formal development of features. We are in the process of integrating the different semantics into one coherent model (Gibson & Méry 1996, Gibson & Méry 1997, Gibson & Méry 1998*b*, Gibson & Méry 1998*a*).

7.4 Simulation: validation and verification

We have text-based animators for validating the operational requirements in all three formal models. 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 set of phones[1] 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 animator 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.

8 Conclusions

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.

[1] In all cases, it was necessary to consider no more than 3 phones to validate any given service.

References

Abrial, J.-R. (1996), *The B Book*, Cambridge University Press.

B-core (1996), B-Toolkit User's Manual, Release 3.2, Technical report, B-core.

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

Bolognesi, T. & Brinksma, E. (1987), 'Introduction to the ISO specification language LOTOS', *Computer Networks and ISDN Systems* **14**, 25–59.

Booch, G. (1991), *Object oriented design with applications*, Benjamin Cummings.

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

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

Coad, P. & Yourdon, E. (1990*a*), *Object oriented analysis*, Prentice-Hall (Yourdon Press).

Coad, P. & Yourdon, E. (1990*b*), *Object oriented design*, Prentice-Hall (Yourdon Press).

Constantine, L. (1989), Beyond the madness of methods: System structure methods and converging design, *in* 'Software Development 1989', Miller-Freeman.

Cutts, G. (1991), *Structured system analysis and design method*, Blackwell Scientific Publishers.

DeMarco, T. (1979), *Structured analysis and system specification*, Prentice-Hall.

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

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

Gibson, J. P. (1994), Formal object based design in LOTOS, Tr-113, University of Stirling, Computing Science Department, Stirling, Scotland.

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

Gibson, J. P. (1998), Towards a feature interaction algebra, *in* 'Feature Interactions In Telecommunications V', IOS Press, Lund,Sweden.

Gibson, J.-P. & M´ery, D. (1996), A Unifying Model for Specification and Design, Rapport Interne CRIN-96-R-110, CRIN, Linz (Austria).

Gibson, M. & M´ery (1997), Feature interactions: A mixed semantic model approach, *in* 'Irish Workshop on Formal Methods', Dublin, Ireland.

Gibson, P. & M´ery, D. (1998*a*), Always and eventually in object models, *in* 'ROOM2', Bradford.

Gibson, P. & M´ery, D. (1998*b*), Fair objects, *in* 'OT98 (COTSR)', Oxford.

Guillemot, R., Haj-Hussein, M. & Logroppo, L. (1991), Executing large LOTOS specifications, *in* 'Proceedings of Prototyping, Specification, Testing and Verification VIII', North-Holland.

Lamport, L. (1990), A temporal logic of actions, Technical Report 57, DEC Palo Alto.

Lamport, L. (1995), TLA$^+$, Technical report, December.

Meyer, B. (1987), 'Re-usability: the case for object oriented design', *IEE Software Engineering* .

P. Gibson, B. M. & M´ery, D. (1997), Specification of services in a compositional temporal logic, Rapport de fin du lot1 du marche no 961B CNET-CNRS CRIN, CRIN.

Turner, K. (1993), *Using FDTS: An Introduction To ESTELLE, LOTOS and SDL*, John Wiley and Sons.

Zave, P. (1993), 'Feature interactions and formal specifications in telecommunications', *IEEE Computer Magazine* pp. 18–23.