Formal Requirements Engineering: Learning from the students

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

Abstract

Formal methods are becoming increasingly important in many areas of software development and should be incorporated in the teaching of software engineering. Requirements capture is, in our opinion, the hardest stage of development for students to learn and for lecturers to teach. This paper reports on our experience in teaching requirements engineering using formal methods, where we advocate a multiple methods approach in which students get to evaluate a large range of specification languages: students are more likely to learn the principles of good requirements engineering rather than become experts in one particular (formal) method. The need for formality is introduced step-by-step, where new concepts are identified by the students through the use of case studies. These concepts are then formalised in the most appropriate language or notation. Students are encouraged to question the need for formality — each requirements engineering method is a compromise and the use of formal models needs to be placed within the context of the choices that a requirements engineer has to make.

1 Introduction

Formal methods should be taught as part of any degree in computing science or software engineering. We believe that discrete mathematics is the foundation upon which software development can be lifted up to the heights of a true engineering discipline. The transfer of formal methods to industry *cannot* be expected to occur without first transferring, from academia to industry, graduates who are well grounded in such mathematical techniques.

This paper reports on our first attempt to teach a formal methods course as part of a degree in software engineering. In [15] we gave a global picture of teaching formal software engineering. This paper examines the requirement engineering issues, which were the most difficult to explain to the students for a variety of reasons:

- **Moving from informal to formal** — the requirements document is the first point of reference in the software development process. The step of going from informal understanding of a problem to a (formal) recording of this understanding is very difficult to learn (and to teach).
- **Coping with changing needs** — it is the nature of requirements to change. Thus, students must learn how to develop techniques which are both flexible *and* incremental. This involves a deep understanding of the compromises that exist within modelling.
- **Working in different problem domains** — to build good requirements models one must have a good understanding of the problem domain which is being modelled. When teaching a requirements engineering course there is always a risk that one will end up teaching about problem domains rather than about requirements modelling. However, students must also learn that requirements modelling and analysis go hand-in-hand: if they work in well-understood domains then they will never learn the importance of the analysis.
- **The need for customer orientation** — one must not lose sight of the customer in the whole process.

Rather than concentrating on one particular requirements engineering formalism or method, we worked on a set of small case studies, using mathematics and specification notations in a flexible and intuitive manner, where the students could appreciate the need for formality. Each case study was intended to illustrate *why* formalism was needed, *what* sort of formalism could meet our needs and *how* to define and (re)use this formalism. The case studies were not intended to rival those which the students had already seen in standard modelling languages — they had previously seen

ER diagrams, UML, SSADM and many other graphical notations — our goal was to show that there was a need for formality and that the formal models could complement the less rigorous approaches.

An unexpected result was that we also identified weaknesses in our understanding of formal methods: students' *naive* questioning helped us to identify how the methods, and the teaching of these methods, could be improved. In brief, it was not just the students who were learning!

Before we proceed to the main body of this report, we give some background information concerning the course. The course was taught at the Université Henri Poincaré (Nancy I), France. It was part of the degree *Ingnierie Mathmatiques et Outils Informatiques*. The title of the course is (after translation): `Software Engineering (using formal methods)`. The degree would be the equivalent of an MSc at a British University. There were 19 students who had already studied software engineering using many different models and methods. The course was taught in 36 hours: we estimate that between 12 and 16 hours were spent on formal requirements engineering (not including their practical course work).

In section 4.6, we comment on the feedback we had from the students which led us to change some aspects of the material and teaching techniques. We also comment on how some of the course material is being re-used in a Masters of Software Engineering degree at NUI Maynooth, Ireland.

2 Related Work

The teaching of formal methods has been somewhat neglected as a subject in conferences and journals: the motivation for this paper was to increase awareness of the need for collaboration in this area. However, we have found some material which, if not directly related to our problem of formal requirements engineering, did influence our approach.

2.1 Workshops

A CTI workshop on *Teaching Formal Methods*, was held at the University of Huddersfield in September 1995. In general, this workshop addressed the problems of motivation, doing real proofs, and integrating formality with more graphical development methods. Specific problems of teaching with Z [28], one of the favoured teaching languages, and choosing teaching material were also examined. The workshop did not directly address the problem of requirements capture.

An earlier workshop — *Teaching Formal Methods Curriculum Development Workshop* — was held in Hamilton College Clinton, New York, in August 1994. The purpose of the workshop was to develop modules and materials for teaching formal methods in an undergraduate setting. The modules covered material such as: propositional/predicate calculus, with applications to assertions/pre- and post-conditions, loops and invariants, category theory, algorithm design, parallel constructs, operational semantics, formal methods with OOP, and applications of mathematica. Again, the workshop did not address the subject of this paper.

2.2 Published papers

We were also encouraged by a range of papers which explain the teaching of functional programming [20, 21] logic [17, 19] and discrete mathematics [18] to computing science students. (Hart et. al. [16] suggest that this can be done with school children, and many of their techniques are equally applicable for our university students!) There is also an interesting calculator case study [26] which illustrates the formal reasoning about programs, and comments on how this can be used to introduce formal methods.

2.3 Web material

A large number of universities also provide information, on the internet, with regard to their *formal methods* courses. It is beyond the scope of this paper to review all the material that is available: our impressions are that most courses are method-oriented and concentrate on verification rather than validation techniques. This would not be suitable for teaching requirements modelling.

3 The Introductory Lectures

3.1 Software Engineering and Formal Methods

By way of motivation, and introduction, a brief overview of the first lecture is given in figure 1, where the following questions were answered: What is software engineering? What is a formal method? Why apply formal methods in software engineering?

Figure 1 illustrates the different steps in a traditional engineering process: analysis, requirements capture, design, implementation, and evolution. The formal methods are principally concerned with maintaining *correctness*, the property that an abstract model fulfils a set of well defined requirements [2, 4, 8, 7], between the initial *customer oriented* requirements model and the final *implementation oriented* design. The formal boundaries break down at either end of the software development process because, in general, target implementation languages are not formally defined and customer understanding of their requirements is not complete.

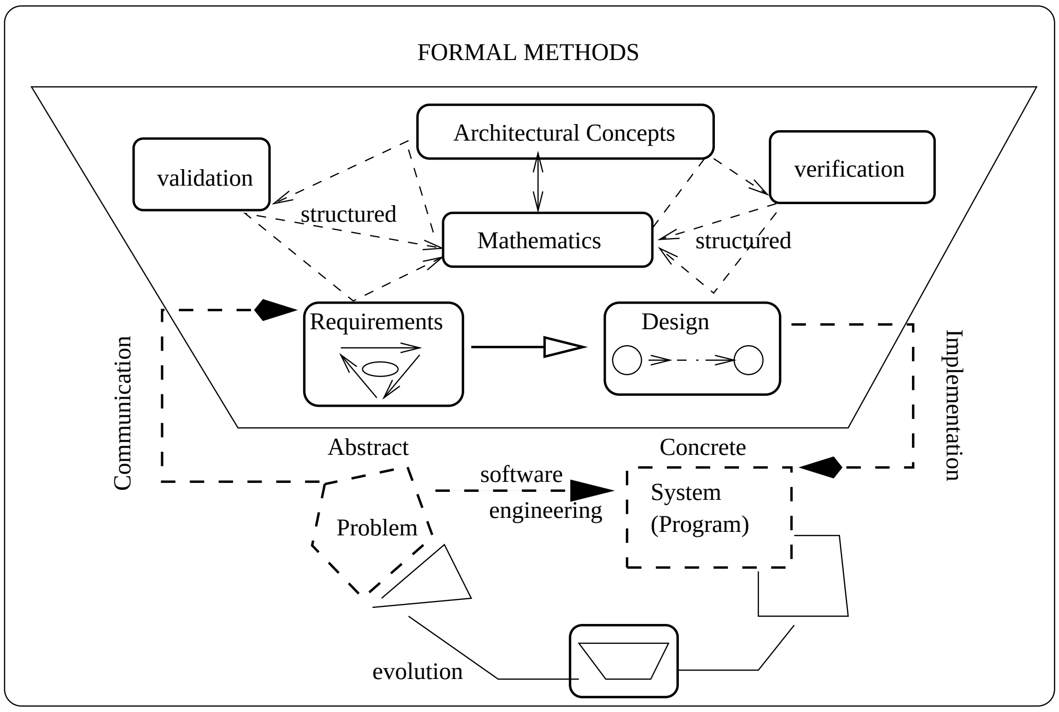


Figure 1. Software Engineering and Formality

Software development has reached the point where the complexity of the systems being modelled cannot be handled without a thorough understanding of underlying fundamental principles. Such understanding forms the basis of scientific theory as a rationale for software development techniques which are successful in practice. This scientific theory, as expressed in rigorous mathematical formalisms, must be transferred to the software development environment. Only then can the development of software systems be truly called *software engineering*.

3.2 Requirements Engineering: an overview

Requirements engineering appears as the first step in the formal development process. As such, a general overview of this development step formed the backbone of the 2nd lecture. The goal was to explain the following:

- **Why requirements engineering is important**
  Analysis is the process of maximising *problem domain understanding*. Only through complete understanding can an analyst comprehend the responsibilities of a system. The modelling of these responsiblities is a natural way of expressing system requirements. The modelling process increases understanding. Once the model is sufficiently rich to express all that is needed, then the analysis is complete and design can begin.

- **Why the customer is important**
  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 *Requirements Capture and Analysis* (RCA) process is much simplified; however, it is more likely that the customer is a group of clients, each with their own particular needs. These clients may be people, machines, or both. One of the main problems in dealing with a set of customers is that the inter-related set of requirements must be incorporated into one coherent 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).

- **Why the process can never be perfect**
  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.

- **Why requirements engineering is difficult**
  The analysis model must be capable of fulfilling two very different needs. Firstly, it must be *customer oriented*, i.e. there must be a direct correspondence between the model and how the customer views the problem. Secondly, the model must be useful to designers. The system requirements must be easily extracted, and the structure of the problem domain must be visible for (potential) re-use in the solution domain. The easiest way in which a model can play this dual role is if the same underlying notions and principles are present in the problem and solution spaces.

- **How formality can help**
  Mathematical rigour is necessary for formal validation, testing and completeness and consistency checking. The advantages of formal methods in the specification of requirements are well documented (see [6, 3, 9], for example).

- **How formality can hinder**
  Formal methods do not come for free. They require much more rigorous development techniques which are more time consuming and more difficult to master. Furthermore, formal methods risk being too difficult for the client (or engineer) to understand. Extra work is required to make them presentable to anyone other than the (mathematically oriented) requirements engineer.

- **The difference between validation and verification**
  It is important that the requirements engineer understands that validation is about checking that a formal model correctly captures the client's needs, and that verification 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, but this is not validation.

3.3 Requirements Engineering: goals, principles and methods

The main goals of the formal requirements engineering part of the course were: teaching principles, teaching (requirements) engineering as a process of compromise, giving an overview of standard formal techniques and methods, and teaching students how to evaluate tools and techniques with respect to different problem domains. We also wanted to emphasise that formality is not the solution to all engineering problems and that the mathematical models should be complementary to the less rigorous techniques which are more commonly used in industry.

Five of our case studies, reviewed in section 3, illustrate how we attempted to reach these goals. The case studies involved using a number of different formalisms; in this paper we report on using ACT ONE (an ADT[24] which forms part of the formal specification language LOTOS[5]) for explaining abstraction, using Caml (a functional programming language, based on SML[27]) for explaining design transformations and equivalence, using OO ACT ONE [11] for explaining incremental development, using temporal logic (TLA[22]) for explaining integration problems, and using purely operational state transition models for explaining the need for nondeterminism. This list is not exhaustive: in other studies we used, for example, B[1], Z[28] and PVS[25]. We do not claim to have taught any of these formalisms in any detail. We did, however, try to convey the idea that some formalisms are better suited to some problems than others.

4 The case studies

4.1 Teaching — abstraction

It is often said that requirements state *what* not *how*. This can be quite misleading since the crux of the matter is the notion of abstraction. We should say that requirements should be abstract enough to allow for many different correct implementations, whilst being concrete enough so that clients and designers can use them as a medium for understanding what is really required.

Abstract data types are often used to show how abstract requirements models can be built. We expanded upon this by showing how an ADT can be thought of as an abstract class specification. Then, we showed how such a class can be used as a super-class of a more concrete implementation class. We also wanted to show that requirements may be both logical and operational. The concept of invariant properties, within the ADT specification of a class, showed the two different points of view. A simple set class was used to illustrate these important aspects of requirements capture.

4.1.1 The set requirements

The original idea for this study came from a French text on graph algorithms [23], where the author explained that the way in which sets where implemented has a great influence on how they can be used for graph problems: where graphs are specified as sets of nodes and arcs. We examined how different set structures could benefit the implementation of certain algorithms whilst other structures made the implementation more difficult. In this way we argued that over-specification of an abstract set could have unwanted knock-on effects with regards to the later design stages.

To begin, the students were asked to specify a set using the abstract data type ACT ONE. All the students took a linear approach much like one would see in an implementation using linked lists. The listing, below, is typical of a correct specification developed by the students:

```
TYPE Set is element SORTS Set
OPNS empty:-> set
     add: set -> bool
     remove: set,element -> set
     contains: set, element -> bool
EQNS remove(empty, el1) = empty;
     [el1 eq el2] =>
      remove(add(S,el1),el2)=remove(S,el2);
     [el1 neq el2] =>
      remove(add(S,el1),el2)=add(remove(S,el2),el1);
     contains(empty,el) = false;
     [el1 eq el2] =>
      contains(add(S,el1),el2)=true;
     [el1 neq el2] =>
      contains(add(S,el1),el2)=contains(S,el2);
```

We went on to show the students that the internal (list-like) structure of their specification was quite arbitrary and there were a number of ways in which the set specification could have been written: as an ordered list or binary tree, for example.

4.1.2 Some requirements engineering problems

In the ADT specification the groups produced fundamentally two different (yet equivalent) specifications. Two groups produced specifications in which `adding` an element first checked if the element was already in the set and did not change the set if this was true. Three groups produced specifications (similar to the code above) in which the `remove` was defined to remove multiple elements whilst the `add` allowed multiple entries. One group fell between these stools and did not realise that there was a problem with multiple elements. The students wanted to know which specification was *best*: here we had to explain the notion of *equivalence*, *invariants* and the need for *extensibilty*. A more difficult question was how to specify the set more abstractly so that both of these specifications were *correct*. This led the students to pose the following question:

> *Why do we say that one model is more concrete (or abstract) than another if they express the same requirements?*

4.1.3 Lessons learned

It was important to teach the students that operational semantics were not the only option for building requirements models. The set could be specified informally as:

> *One can add and remove elements, and test if an element is in the set. After removing an element, the test will return false (for the element just removed). After adding an element, the test will return true (for the element just added), until the element is removed.*

These are properties that we would like a requirements model to exhibit and as such could be said to be more abstract requirements. We showed how the students' abstract data type specifications could be proven to exhibit such properties (expressed in temporal logic). The problem seemed to be that certain requirements are quite naturally specified operationally, others are better specified logically, and the majority need a mixed-semantic point of view.

After this case study we realised the need to look at the notion of *abstraction level*. In fact, we identified 2 aspects of a specification which could be said to influence its level of abstraction, namely: the amount of *structure* and the amount of *nondeterminism*. It can be argued that adding structure to a specification can reduce implementation freedom: it may be very difficult to restructure a specification towards a certain implementation architecture (during design) if the structures found at both ends of the development spectrum are conceptually very different. Furthermore, it can be argued that nondeterminism in a specification provides an explicit engineering choice which must be taken when mapping onto a particular implementation architecture: the more nondeterminism then the greater level of freedom in making design decisions. However, we stress that this issue is not clear cut and there is no generally applicable measurement for *level of abstraction* within a specification model.

4.2 Teaching — importance of structure

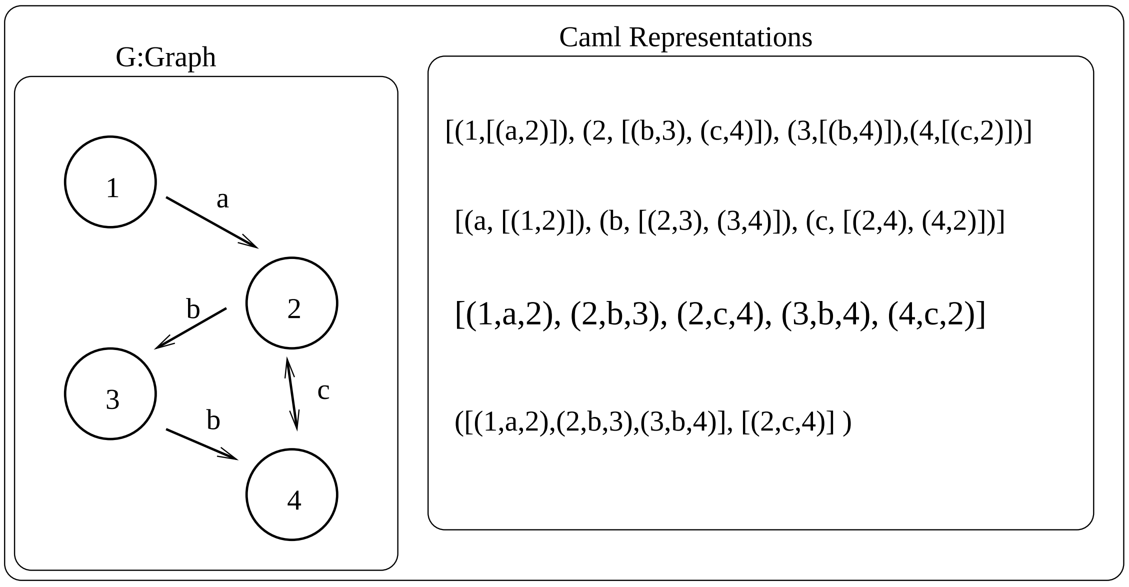
The original idea for this study came from working on graph algorithms, using Caml (a functional programming language based on SML[27]). The goals were to examine the importance of structure in requirements models and show how equivalent specifications could have different structures.

4.2.1 The graph requirements

The question posed was as follows:

> Using lists and cartesian products, represent the graph `G`, as shown in figure 2.

The four most interesting representations, proposed by the students, are shown to the right of the diagram. Using their chosen representations, they were then asked to write conversion functions for going from one form to any of the others, thus illustrating that their equivalence was based on *iso-*



G:Graph
Caml Representations
1
2
3
4
a
b
b
c
[(1,[(a,2)]), (2, [(b,3), (c,4)]), (3,[(b,4)]),(4,[(c,2)])]
[(a, [(1,2)]), (b, [(2,3), (3,4)]), (c, [(2,4), (4,2)])]
[(1,a,2), (2,b,3), (2,c,4), (3,b,4), (4,c,2)]
([(1,a,2),(2,b,3),(3,b,4)], [(2,c,4)])

Figure 2. Structuring Understanding

morphic mappings. The result was that the following questions were posed:

- Using the first notation, what graph is represented by `[(1,[(a,2)]),(3,[(b,1)])]`?
- Using notations 2, 3 and 4, how do you represent a graph which contains a node unconnected to other nodes? For example, the graph with one node and no arcs can be represented using the first notation as `[(1,[])]` but cannot be represented in the other three notations.
- In all the notations, the order of the elements in the lists is unimportant: is this the same sort of equivalence as seen between different representations?
- In the fourth and final notation, the second list (with a single element `(2,c,4)`, in this example) represents those arcs which are bi-directional. In this case a list element `(x,char,y)` is equivalent to `(y,char,x)`: what sort of equivalence is this?

4.2.2 Requirements engineering issues

Unintentionally, the graph example had shown the need for *invariants* in compositionally structured specifications, where the invariants are used to identify constraints between the states of the specification components. The first graph representation requires that all target nodes at the ends of arcs, are themselves found in the graph. The students quite easily specified this invariant with a Caml boolean function. In the 3 other representations, there is no need for an invariant because the list of arcs implicitly defines the nodes found in the graph. However, none of these can represent the case of an unconnected node. Thus, these representations are *incomplete*.

4.2.3 Lessons learned

Given a flexibility in the way in which different structures can be used to specify the same requirements, the students wanted to know how they could judge which structure was *best*. At this point we emphasised 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.

4.3 Teaching — incremental development

The goals of this case study were to show the importance of re-use (even in the early stages of development) and to formalise the different types of re-use — re-use of abstractions, re-use of behaviour, re-use of validation, re-use of verification, re-use of structure, and re-use of methods.

4.3.1 The drawing program requirements

This example was inspired from a long running (and recurrent) thread in the `comp.object` newsgroup, where the seemingly trivial specification of a square as a subclass of a rectangle was shown to be problematic. The problem posed was the following:

> In a drawing program, shapes are to be represented on the screen and manipulated. There is already a mathematical classification of shapes which exists in the problem domain. For example, *a square is a rectangle with all four sides equal*. Can, and should, we use this *is-a* relationship to define a square as a subclass of rectangle in our specification of the requirements of a drawing program?

All the students said that the *is-a* relationship should be used. We then posed the question of what happens if one of the program's functions is to move elements around the screen. Having said that there was no problem, they were then asked *why* there was no problem, and could they specify a function which would cause a problem. After some encouragement, they managed to say that stretching a shape may cause a problem: after you stretch a square it may no longer be a square.

The students had already seen the importance of invariants when they worked on the graph case study: here they immediately saw the need of a `square` class invariant to specify that all its sides are of equal length. Provided that none of the rectangle operations can break this invariant then the `square` can be defined as a subclass of the `rectangle`. However, if an operation such as `stretch` is part of the `rectangle` interface, then `square` cannot be defined as a subclass. Furthermore, if the `square` is defined as a subclass of a `rectangle` then the `square` itself cannot have a subclass *extended* by an operation like `stretch`.

4.3.2 Requirements issues: Incremental verification and validation

The drawing tool requirements, written in OO ACT ONE[11] provided the students with a first chance to use some of the tools which often accompany formal methods. An animator let them validate behaviour compositionally. An ACT ONE consistency checker was used to verify the invariant properties for each of the classes.

4.3.3 Lessons learned

The students appeared to understand the role of formal methods in subclassing better than their role in composition. As part of the drawing tool requirements, it was necessary to integrate interacting functionality. The composition of the overall system required a clear understanding of the way in which components where to be configured. The hardest step for the students was validation of the configuration process: they had requirements components which could have been put together in a number of different ways and this configuration was not explicitly treated in the informal requirements which I had given them. We returned to this problem in the telephone feature case study (see below).

In addition, we saw that re-use during requirements modelling often depends on identification of re-usable composition mechanisms, rather than just re-usable components. Unfortunately, it was too difficult for the students to understand this issue at this early stage. In fact, structural and architectural re-use can be said to be the hardest part of all software engineering.

4.4 Teaching — integrating different formalisms

The students were asked to analyse the problem of building requirements models for telephone services (like call forwarding, call identification, call screening, etc ...). Within a few minutes they had identified the problem of feature interactions [29]. This case study was intended to show that this was a problem that existed at the requirements level, and which was best analysed using a combination of different models[14]. It was also intended to illustrate the need for formal methods.

4.4.1 The telephone service requirements

We decided to concentrate on two well-understood services: an answering machine and call forwarding. These services had to be added to the standard plain old telephone service (POTS) requirements. It became clear to the students that there were 3 types of telephone service requirements:

- **operational** —we should be able to perform certain sequences of actions. For example: *I want to be able to phone someone without an answering machine, have my call forwarded to an answering machine, and leave a message.*

- **safety** —we should never reach a state where some unwanted property is true. For example: *I never want to leave a message and talk to someone at the same time.*

- **liveness** —we should eventually be able to reach a certain state or be able to perform a certain action. For example: *I should eventually get to talk or to leave a message.*

The need for liveness led us to introduce the temporal logic of Lamport (TLA[22]) and its theorem prover TLP[10].

4.4.2 Requirements engineering issues

The difficulty was not in formalising each of these requirements — they could now, after being introduced to TLA, identify which formalisms would be best suited to modelling each of the properties. The difficulty was in finding a way of integrating them. In fact, only through integration can the feature interaction be found:

> *I phone someone with an answering machine and my call is forwarded to another phone which has no answering machine. If my call is unanswered then I may no longer be able to leave a message at the original phone.*

4.4.3 Lessons learned

The students learned that having an informal understanding of a problem domain may not always help the process of building a good requirements model within that domain. In a familiar problem domain, one is tempted to incorporate informal understanding in the models, without making this explicit. More worryingly was the way in which this manifests itself as implicit assumptions about the environment of the system being modelled.

The students also now saw the importance of mixed semantic models[13]. However, as this is an on-going area of research, we considered it too advanced a topic for the students to address in any more detail.

4.5 Teaching — putting it all together

The original motivation for this lift system problem came from a study which was carried out when testing LOTOS for specifying problems with an object oriented approach [12]. This problem was given as a course project (3 or 4 students in each group). The problem was for them to specify (in whatever way they wished) the requirements of a lift. The goal was that they would begin to appreciate the need for formality (particularly in the logic of lift movement between floors).

4.5.1 The lift requirements

The informal requirements given to the students were as follows:

> *Specify the requirements that a user would place on a lift system. Try to specify what is required rather than how it is to be achieved. Explain how you would validate that your requirements match the user's needs. After such validation, explain how you would use your specification to verify that a particular lift system behaved correctly.*

The lift case study was a great success (for all the wrong reasons). We were hoping that their informal specifications would be ambiguous, incomplete and inconsistent, thus showing the need for formal models. However, the students were one step ahead, again. Three groups took an operational approach to specification — handing in what amounted to well-documented pieces of C++ and JAVA code. The other two groups shocked us even more by specifying the problem at a logical level of abstraction. They stated, using temporal logic:

> *When I arrive at a lift on floor* `x` *and I want to go to floor* `y`*, the lift will eventually arrive at* `x`*, let me enter, eventually arrive at* `y`*, and let me exit.*

4.5.2 Requirements Issues: Over- and under-specification

The operational groups clearly had no problems with the validation of their specification, but did not understand the verification part of the problem. The logical groups did not know what they had to validate, but knew precisely how to verify that a given lift *worked*.

To test their *understanding*, we proposed two lift implementations:

- A 'supermarket model', where the user who wishes to use the lift has to take a ticket and wait their turn. The lift serves only 1 user at a time: going to collect them at their current floor and then taking them to their requested floor.
- A 'no-logic model' in which the lift moves continually from top to bottom, and back from bottom to top, stopping for a few moments at every floor.

Using the case study, we now had examined the problems of *overspecification*, the integration of *logical* and *operational* views, and the difference between validation and verification.

4.5.3 Lessons learned

To complete the study, we have set an exam question on the problems of compositional development and re-use at different levels of abstraction. The students were asked to suggest ways in which lift systems could be composed from 2, or more, lift components. They realised that it was easier to re-use abstract components than it was to re-use more concrete components. In this way they learned the value of abstraction in requirements models.

4.6 Course Feedback

At the end of the course, the students were asked to complete a questionnaire (a copy of which can be provided on request). Some of the most interesting feedback is summarised below:

- They would have preferred to spend more time learning how to use the tools themselves rather than having to rely on their teachers to demonstrate their (in)effectiveness.
- They were frustrated at not having a real customer to communicate with, because their teachers were always used to play this role, and they thought this was rather *unnatural*.
- A minority stated that they would rather have learned 1 formal method in great detail rather than just touching the surface of a number of methods.
- A majority acknowledged the benefit of using formal methods but could not see how they could be easily incoporated into the development tools with which they were already familiar.

In the Autumn of 1999, we had the opportunity to reuse some of the course material in the software engineering masters degree program at NUI Maynooth. We decided to address some of the issues raised by the students. In particular, we thought it important to choose 1 formal method which they could learn in more detail whilst showing that it had weaknesses in certain areas. We could then cover other methods, in less detail, by showing how they addressed the perceived weaknesses of our chosen language. It was also decided to try and better explain how the formal methods could be better integrated with other *industry strength* development methods.

To meet these aims, we focussed on formalising object oriented modelling. Our chosen specification technique was LOTOS; this provided ADT and process algebra parts, and had associated object oriented specification styles. Furthermore, the tool set incorporated an animator. Our teaching technique continued to be case-study oriented, yet our consistent use of a single specification language (and style) was, we deemed, an improvement.

The masters program is not yet completed, so we have not yet received any detailed feedback from our questionnaires. However, informal feedback from the students suggested that they would have prefered to have spent more time on the other *interesting methods*! They also thought that it would have been better for them to see how requirements engineering fitted into the whole software process.

5 Conclusions

This paper examines the particular problems of teaching formal requirements engineering. Through a number of case studies we showed that we still have much to learn about the best way to teach this subject. In fact, we acknowledge the need to integrate different formal models with the more successful, but less rigorous, approaches which are currently being used in real industrial projects. The integration of these different views is part of our current research[13].

References

[1] J.-R. Abrial. *The B Book*. Cambridge University Press, 1996.

[2] R. Baber. *The Spine of Software — Designing Provably Correct Software: Theory and Practice, or: A Mathematical Introduction To The Semantics Of Computer Programs*. John Wiley and Sons, 1987.

[3] D. Bjoener and C. B. Jones. *Formal Specifi cation and Software Development*. Prentice-Hall International, 1982.

[4] T. Bolognesi. Fundamental results in the verifi cation of observational equivalence: a survey. In H. Rudin and W. C.H., editors, *Protocol Specifi cation, Testing and Verifi cation VII*. North-Holland, 1988.

[5] T. Bolognesi and E. Brinksma. Introduction to the ISO specifi cation language LOTOS. In *The 1st International Conference on Formal Description Techniques (FORTE 88)*, 1988.

[6] R. Bulzer and N. Goldman. Principles of good software specifi cation and their implications for specifi cation languages. In *Proc. of Reliable Software*, pages 58–67. Cambridge, Mass., 1979.

[7] E. Cusack. Refi nement, conformance and inheritance. In *Open University workshop on the theory and practice of refi nement*, 1989.

[8] R. DeNicola. Extensional equivalence for transition systems. *Acta Informatica*, 24:211–237, 1987.

[9] A. Diller. *An Introduction To Formal Methods*. John Wiley and Sons, 1990.

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

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

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

[13] J. P. Gibson, B. Mermet, and D. M´ery. Feature interactions: A mixed semantic model approach. In *Irish Workshop on Formal Methods*, Dublin, Ireland, July 1997.

[14] J. P. Gibson, B. Mermet, and D. M´ery. Specifi cation of services in a compositional temporal logic. Rapport de fi n du lot1 du marche no 961B CNET-CNRS CRIN, CRIN, 1997.

[15] J. P. Gibson and D. M´ery. Teaching formal methods: Lessons to be learned. In *2nd Irish Workshop on Formal Methods*, Cork, Ireland, July 1998.

[16] R. Hart, Maltas. Teaching discrete mathematics in grades 7 –12. In *Mathematics Teacher*, volume 83, pages 362–367, 1990.

[17] R. J. Logic in fi rst courses for computing science majors. In *World Conference on Computers in Education*, pages 467–477, 1995.

[18] R. J. A three paradigm fi rst course for cs majors. *Proceedings of 26th ACM Tech. Symposium SIGCSE Bulletin*, 27(1):223–227, 1995.

[19] R. J. A logical foundation course for cs majors. In *Australian Computer Science Education Conference*, pages 135–140, July 1996.

[20] v. d. H. G. Joosten S., van der Berg K. Teaching functional programming to fi rst-year students. In *Journal of Functional Programming*, volume 3, pages 49–65, 1993.

[21] Lambert. Using miranda as a fi rst programming language. *Journal of Functional Programming*, 3(1):5–34, 1993.

[22] L. Lamport. A temporal logic of actions. *ACM Trans. Prog. Lang. Syst.*, 16(3):872–923, May 1994.

[23] G. L´evy. *Algorithmique Combinatoire: Mithodes Constructives*. DUNOD, 1994.

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

[25] S. Owre, N. Shankar, and J. B. Rushby. *The PVS Specifi cation Language*. Computer Science Laboratory, SRI International, CA, Feb. 1993.

[26] Reeves and Goldson. The calculator project - formal reasoning about programs. In P. M., editor, *Proceedings Software Engineering Conference SRIG-ET*, pages 166–173. IEEE Computer Society Press, 1995.

[27] A. Wikstrom. *Functional Programming Using Standard ML*. Prentice-Hall, 1987.

[28] J. Woodcock and J. Davies. *Using Z Specifi cation, Refi nement, and Proof*. Prentice Hall International Series in Computer Science. Prentice Hall, 1996.

[29] P. Zave. Feature interactions and formal specifi cations in telecommunications. *IEEE Computer Magazine*, pages 18–23, Aug. 1993.