Teaching Formal Methods: Lessons to learn

Paul Gibson, Dominique Méry
(gibson/mery)@loria.fr

Université Henri Poincaré (Nancy I),
UMR N° 7503 LORIA, Campus Scientfique,
BP 239, 54506 Vandoeuvre-Les-Nancy, France

Abstract. 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. These graduates must bring a positive, yet realistic, view on the application of formal methods. Our goal is to produce software engineers who will go out into industry understanding the principles of specification, design and implementation. As these graduates develop their engineering skills, in an industrial setting, they should have the means, and the motivation, to integrate formality and rigour into any environment in which they are found. In this way, the formal methods should start to 'sell themselves'.
This paper reports on our first attempt to teach a formal methods course as part of a degree in software engineering. Rather than concentrating on one particular method, we worked on a set of small case studies, using the mathematics in a flexible and intuitive manner, where the students could appreciate the need for formality. Each case study was intended to illustrate, in turn, the need for some fundamental formalism. 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!

1 Introduction

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. Building a formal model improves understanding. 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. However, formal methods are not

widely used in software development. In most cases, this is because they are not suitably supported with development tools. Further, many software developers do not recognise the need for rigour. The most obvious solution to this problem is to teach formal methods to the graduates who will be the backbone of the industry in the years to come.

This paper reports on the teaching of: *The application of formal methods in software engineering*. 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?

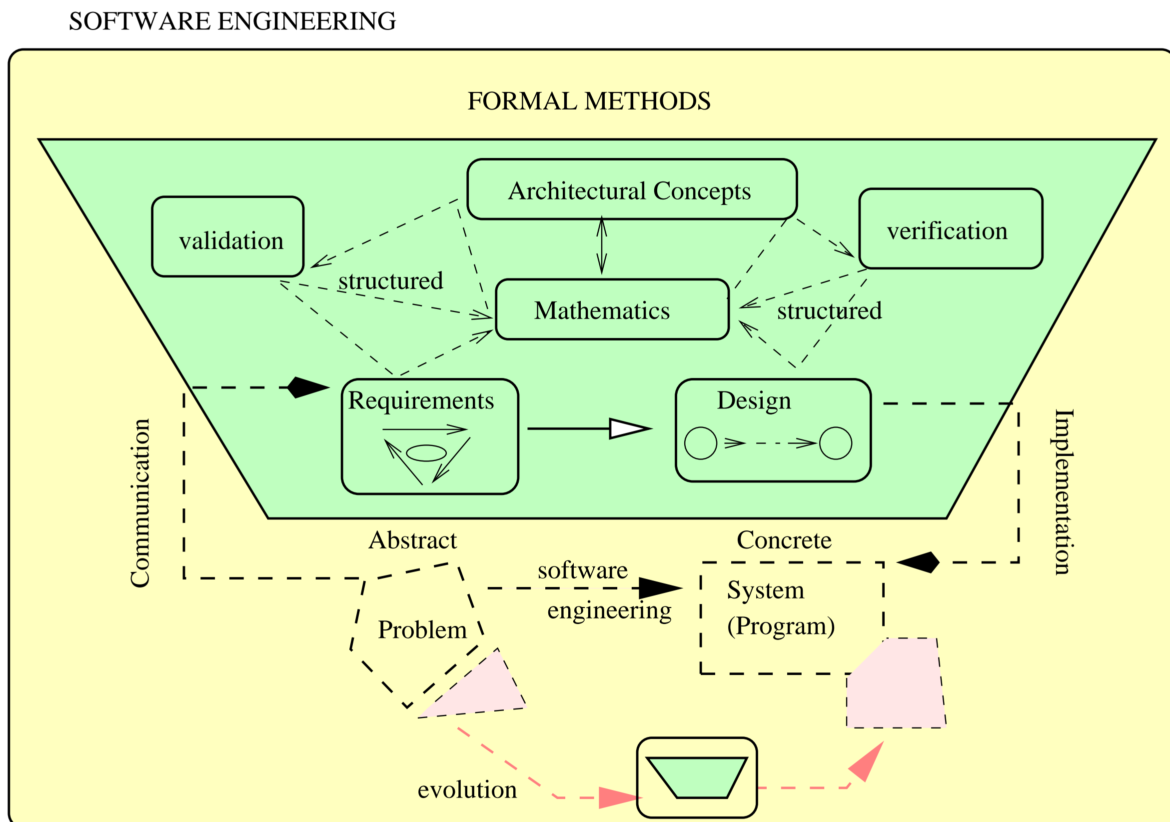

Fig. 1. Software Engineering and Formality

The figure 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 [1,2,6,5], 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.

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*: the application of techniques, based on mathematical theory, towards the construction of abstract machines as a means of solving well defined problems.

As a means of motivating the students, we mention a major study of the state-of-the-art in formal methods [4], carried out 5 years before, which concluded by stating:

> "... formal methods, while still immature in certain important respects, are beginning to be used seriously and successfully by industry to design and develop computer systems ..."

There are a wide and varied range of definitions of *formal method* which can be found in the majority of texts concerned with mathematical rigour in computer science. (A wide range of *formal methods* are considered in [7,23,21,17,24].) For the purposes of the course we propose the following definition:

> *A formal method is any technique concerned with the construction and/or analysis of mathematical models which aid the automation of the development of computer systems.*

To conclude our introduction, both to the students and in this paper, we mention the seminal article by Hall [11] and follow-up articles by Bowen and Hinchey [3] which explore the *myths* of formal methods. The students were asked to comment on each of the myths: leading to an entertaining and enlightening discussion. We concluded that *all* the myths were already firmly implanted in our students' minds. A new goal arose: to trust the students (and the ability of our teaching) to break down these myths in their own time and in their own way — an exam question based on these myths was also written!

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.

There has been a CTI workshop[1] concerned with this subject. In general, the workshop sessions addressed the problems of motivation, doing real proofs, and integrating formality with more graphical development methods. Specific problems of teaching with Z [24], one of the favoured teaching languages, and choosing teaching material were also examined.

A range of other papers explain the teaching of functional programming [18,19] logic [14,16] and discrete mathematics [15] to computing science students. (Hart et. al. [12] 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 [22] which illustrates the formal reasoning about programs, and comments on how this can be used to introduce formal methods. A large number of universities also provide information, on the internet, with regard to their *formal methods* courses.

3 Course Overview

The course was taught at Université Henri Poincari (Nancy I), France. It is part of the degree *Ingénierie Mathématiques 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.

3.1 Student background

There are 19 students: 4 females and 15 males, 13 who have computing experience and 6 who have not, 5 who have studied discrete mathematics at University and 14 with school level maths, 6 who have worked in industry full-time and 4 with part-time experience, and all, except 2, had no experience with *formal methods*.

3.2 Course Hours and Structure

The course was to be taught in 36 hours: 12 sessions of 3 hours each. It was for us to dynamically organise which sessions would be used for lectures, tutorials, practicals, discussions, and team case-studies: this was done flexibly and most sessions were a mixture of lectures and case studies. The 3 hour sessions were not ideal — the attention span of students is much less than this. However, a *lecture a little, question a little, practise a little, case study a little* technique reduced the potential for "boredom" (for want of a better word) and provided much needed feedback on the teaching process, which is vital when teaching formal methods for the first time.

[1] Workshop on Teaching Formal Methods, 12 Sept 1995, The University of Huddersfield, `http://www.hud.ac.uk/schools/comp+maths/events/cti.html`.

3.3 The Importance of Case Studies

As the course advanced we came to see the value of the case studies. When introducing new concepts, a small case study (often *toy* problems) were used to illustrate *why* formalism was needed, *what* sort of formalism could meet our needs and *how* to define and (re)use this formalism. Some of the studies took a few minutes whilst the longest, the specification of a lift, was set as course work to be examined. The most interesting cases are reviewed in the next section.

4 Case Studies

For each of the case studies, the following information is given:

- the source of the material,
- the time taken for the work,
- the goal of the study,
- a brief summary of the problem, and
- the lessons learned.

4.1 Study 1: Term Re-write Systems

The original inspiration for this example came from the classic text by Douglas Hofstadter [13], which was also recommended reading on the course. Two simple term-rewriting systems were studied during 1 hour. The goal was to introduce the following concepts, using as simple a mathematical model as possible: *formal system*, *calculability*, *termination*, *proof*, *theorem*, *decision procedure*, *meta-analysis*, *structural induction*, *necessary and sufficient*, *isomorphisms*, *meaning* and *inconsistency*. As the list above shows, even the simplest problems give rise to complex vocabulary.

A typographical re-write system (TRS) is a *formal system* based on the ability to generate a set of strings by following a simple set of syntactic rules. Each rule is *calculable* (since the generation of a new string from the old string by application of a rule *always terminates*). However, a TRS may be defined to produce an infinite number of different strings. The problem of *decidability* is deciding if any given string can be generated by the TRS.

The MUI TRS

```
The MUI system
--------------
Alphabet = {M,I,U}
Strings: any sequence of characters found in the alphabet
Axiom: MI
Generation Rules: forall strings x such that x is a String of MUI or x =""
```

```
1) xI    generates      xIU
2) Mx    generates      Mxx
3) xIIIx generates      xUx
4) xUUx  generates      xx
```

We say that a *theorem* of a TRS is any string which can be generated from the axioms (or any other theorem). We say that a *proof* of a theorem corresponds to the set of rules which have to be followed to generate that theorem. We asked the students to prove the theorem `MUIIU`. The following question was then posed:

> Can we automate the process by developing a function/machine for testing the *theoremhood* of given string, in a finite period of time?

Unsurprisingly, none of the students managed to *find* such a mechanism! Such a machine would be a *decision procedure* for `MUI`.

The following *solution* was proposed and the students asked their opinions:

> For `MUI` we construct a tree of strings, starting from the axiom (at the root). Any applicable rule constitutes a branch of the tree. To decide if a given string is a theorem it is sufficient to keep extending the tree until the string is found. We know that if the string is a theorem then the machine will terminate with result true. But, what happens if the string is not a theorem?

They all asked, as hoped, what happens if the procedure does not terminate.

Next we asked them to consider the string `IIIUUUIIIUUUI`. Is it a theorem of the system? The more observant reader would say *no* immediately. From looking at the axioms and rules we can *see* that all theorems must start with an `M`. However, we cannot prove this within the system: the only proof that the system allows is a sequence of re-write rules. Such a proof has to be done *out of* the system using a *meta-analysis*.

So, what sort of reasoning tells us that all `MUI` theorems start with an `M`. Most of the students were familiar with mathematical induction so the notion of *structural induction* was not difficult to explain.

The meta-property *all theorems start with an M* is called a *necessary* but not *sufficient* property of theorem-hood. It is necessary because if it is not true of a given string then the string cannot be a theorem. It is not sufficient because there are strings which start with an `M` which are not theorems.

The `MUI` TRS illustrates a *toy* formal system: it does not appear to offer any practical benefits with respect to real computation. The next TRS shows how we can use such a system to reason about *some* mathematical properties.

The pq- TRS

The pq- TRS

```
-----------
Alphabet     p q -
Axioms       for any x such that x is a  possibly empty sequence of "-"s,
             xp-qx- is an axiom
Rule 1)      for any x,y,z which are possibly empty sequences of "-"s,
             if xpyqz is a theorem then xpy-qz- is a theorem
```

We asked the students to define a decision procedure (a terminating boolean function) for this formal system. (The secret is that all the re-write rules lengthen the given string. Thus, we can generate a tree of theorems and we know that a string is a non-theorem when we start producing strings which are longer than the length of the required string.)

The interesting aspect of `pq-` is that it provides us with a formal model of a mathematical property: the addition of integers. For example:

```
--p---q-----  is a theorem and "2+3=5" is true
--p-q--       is a non-theorem and "2+1=2" is false
--p-p-q---p   is a non-theorem but "2+1+1=4" is true
```

If we intepret `p` as plus and `q` as equals, and a sequence of n `-`s as the integer n, then we appear to have a means of checking $x + y = z$ for all non-negative integers *x,y,z*. The third example, in the list above, shows the syntactic limitation of `pq-`: we cannot reason about the addition of more than two numbers.

We say that pq- is *consistent* (under the given interpretation) because all theorems are true after interpretation. We say that pq- is *complete* if all true statements (in the domain of interpretation) can be generated as theorems in the system. We say that the interpretation is *isomorphic* to the system if the system is both *complete* and *consistent*.

The pq- system is *isomorphic* to a very limited domain of interpretation. The students were asked how such a domain could be widened — following their suggestion, a new axiom was added: `xp-qx`. This new axiom lets us generate many more theorems. The students were asked to comment on the *completeness* and *consistency* issues. They identified a problem: the new system is not consistent with our previous interpretation. For example: `--p--q---` is now a theorem but "2+1=2" is not true. A *good* solution is to change the interpretation to regain *consistency*. For example, we may intepret `q` as ">=". Now, we have consistency, but ... we have lost completeness. For example, "2+5>=4" is true in our new domain of interpretation but `--p-----q----` is a non-theorem.

4.2 Lift: Informal vs Formal

The original motivation for this problem came from a study which was carried out when first testing LOTOS for specifying problems with an object oriented approach [8]. This problem was given as a course project (3 or 4 students in each group) which required, on average, 20 hours work for each

group. The problem was for them to specify (in whatever way they wished) the behaviour 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).

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 specifi cations 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 specifi cation — 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, informally:

> 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.

The operational groups clearly had no problems with the validation of their specifi cation, but did not understand the verifi cation 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*, and the integration of *logical* and *operational* views.

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 are to suggest ways in which lift systems can be composed from 2, or more, lift components.

4.3 Sets: Abstract to Concrete

The original idea for this study came from a French text on graph algorithms [20], where the author explained how the way in which sets where defi ned has a great influence in how they can be used

for graph problems: where graphs are specified as sets of nodes and arcs. This study took two hours to complete: half the time was spent developing an ADT specification. The other half was used to explain different implementation strategies. After the lift specification, the students seemed to understand the need for going from the abstract to the concrete. The goal of this case study was for them to see a development hierarchy as a step-by-step process towards implementation.

The figure below illustrates the hierarchy which we examined:

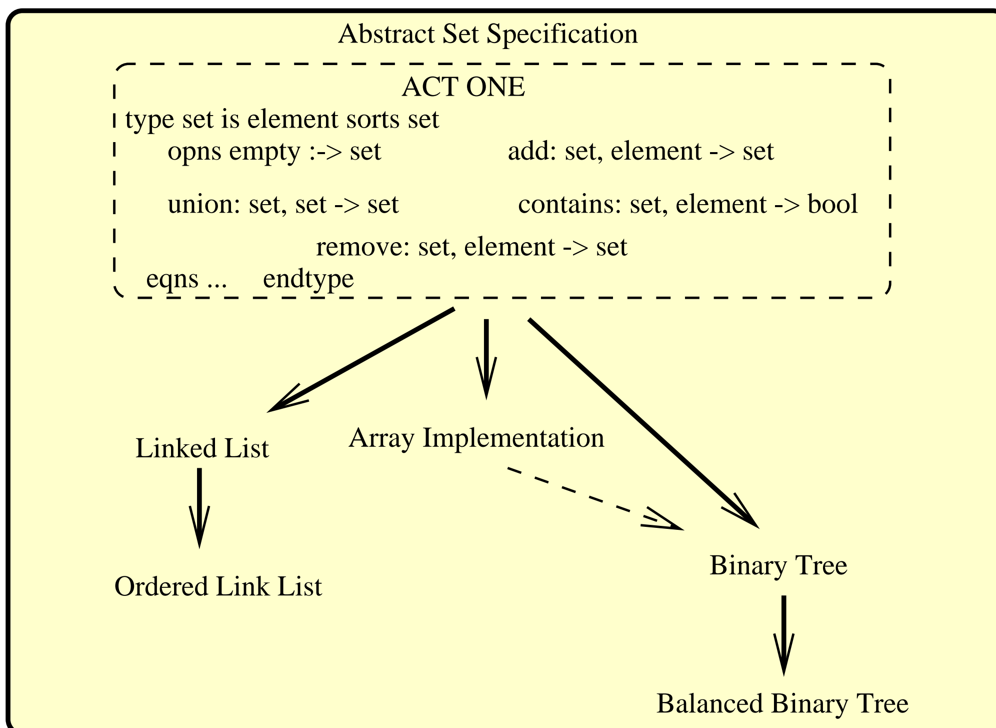

Fig. 2. Designing a Set

This case study brought up some unforseen 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 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*.

- In the diagram, we see five different implementation strategies. Each arrow represents a design step in which the internal structure of the set is changed to *improve performance*. The question which I faced was:

  Why do we say that one model is more concrete (or abstract) than another if they are *equivalent*.

  Intuitively, there is something more concrete about a balanced binary tree than a linked list, but clearly from a purely functional point of view they provide the same behaviour. How do we formalise this notion of *abstraction level*?
- In the hierarchy diagram, there is a dotted link between the array implementation and the binary tree: it is *easy* to implement a binary tree using an array whilst it is not so *easy* to do this implementation with a linked list. The students wanted to know if this concept of *easy to implement using something* can be formalised.

After this case study we realised the need to look at the notion of *equivalence* in more detail, and the need to re-examine the notion of *abstraction level* from the point of view of nondeterminism.

4.4 Graphs: Equivalence and 'Function Follows Form'

The original idea for this study came from working on graph algorithms in Caml, a functional programming language, with our first year students. The study took 1 hour. The goals were to examine the importance of structure in specifications and show how equivalent specifications could have different structures. Different structures aid the specification of certain behaviours whilst some structures hinder the specification. The notion that *function follows form* was to be fundemental. Figure 3 illustrates the problem:

The question posed was as follows:

Using the lists and cartesian products in Caml, represent the graph `G` as shown in the diagram.

The four most interesting representations are shown to the right of the diagram. 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 *isomorphic mappings*. The result was that they posed the following questions:

- Using the first notation, what graph is represented by `[(1,[(a,2)]),(3,[(b,1)])]`[2]?
- 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.

[2] This problem arose from a typing error when testing their functions.

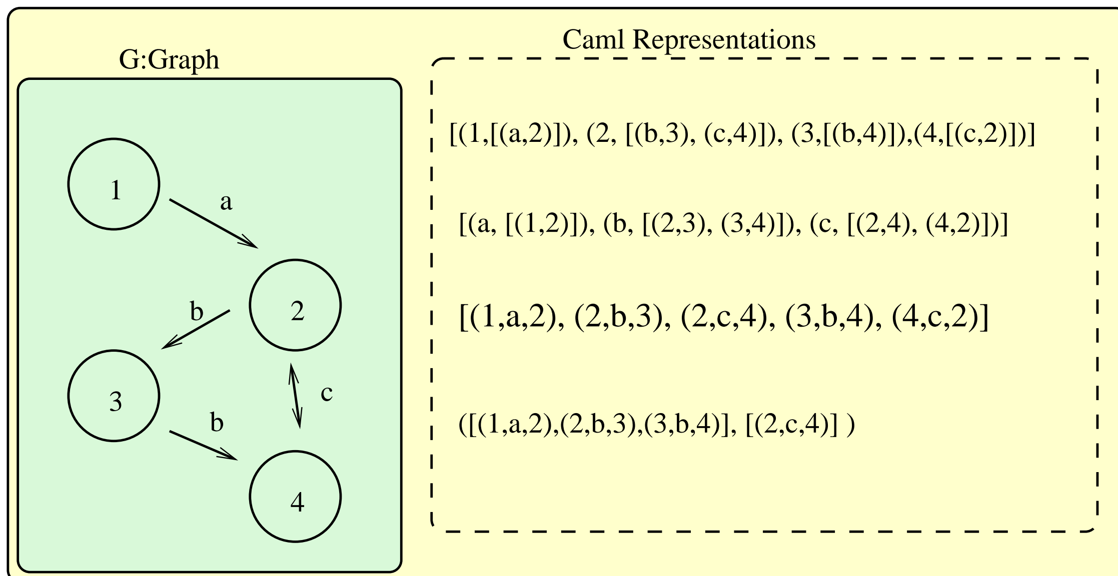

Fig. 3. Graph Representations

– In all the notations, the order of the elements in the lists is unimportant: is this equivalence?
– In the final notation, the second list (with one 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?

Unintentionally, the graph example had shown the need for *invariants* in specifications. 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, the representations are *incomplete*. The students successfully found a *complete* representation which did not need an invariant.

The second part of the case study involved specifying functions and properties on the graphs. The functions *alphabet* and `nodes`, for calculating the set of arc names and set of node labels, illustrated the inter-dependency between function and form. The final key in the invariant puzzle was to get them to specify a function for adding a new arc between two specified nodes. Incredibly, all groups managed to see the importance of maintaining the invariant and specified their new operation accordingly, albeit with different techniques: the first would not add an arc if the two nodes were not already present, the second added the nodes if they were not already there. We come back to this *invariant preserving* when we consider subclassing in object oriented specification.

4.5 Communicating Queues: Objects and Processes

We firmly believe in the integration of object oriented and formal methods [9]. The students were introduced to the notion of an ADT encapsulating the functional behaviour of an object behind an interface. Then, this idea was extended to consider systems of concurrent objects as communicating processes. At this stage we had already spent a few hours with LOTOS. The example of communicating queues is taken directly from our fair objects paper [10] where we examined the specification of nondeterminism in object models. The case study took 1 hour and the goal was to try and get the students to relate informal graphical representations with high level re-usable formal components: this would be a proper design task. The problem is illustrated in the figure below:

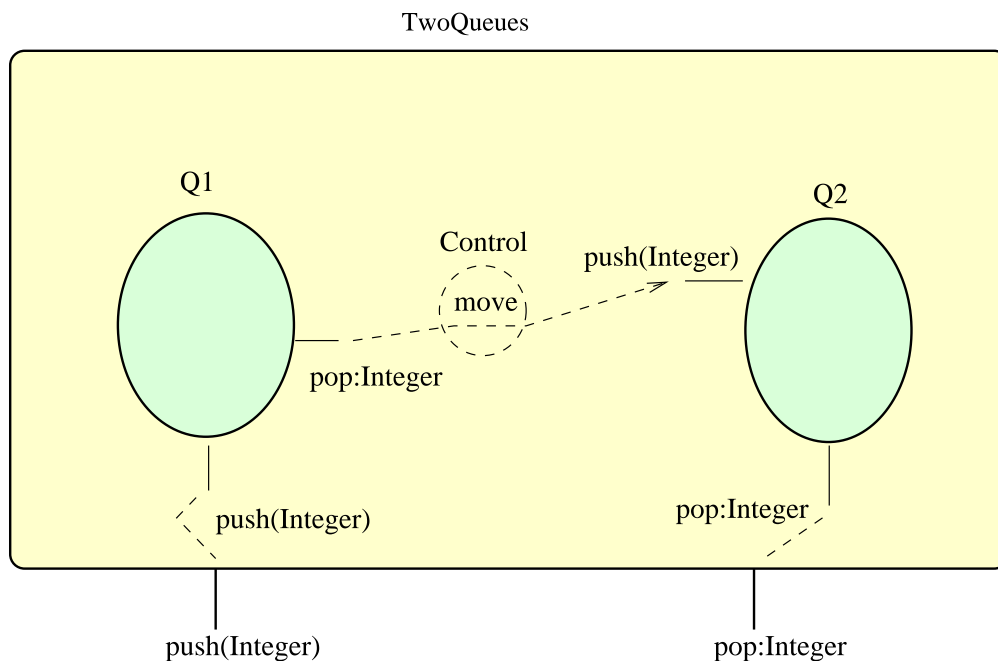

Fig. 4. High level formal design

The question posed was as follows:

> Given the LOTOS process specification of a `Queue` specify a system `TwoQueues` with the following services — `push` an element onto the first queue and `pop` an element of the second queue. Furthermore, we require elements to `move` nondeterministically between the queues. The problem requires a co-ordinating `Control` process for moving the elements from the first queue to the second.

This was one of the more successful studies; however, as usual, the students were placing unforseen demands on their teacher! We were not prepared, at this stage, to explain temporal logic but their questions, listed below, forced us to improvise:

- Is the `TwoQueues` just an implementation of a `Queue`?
- How can we force the items to be moved from Q1 to Q2 without enforcing implementation decisions during design?
- Is it possible to prove equivalence (provided that moves happen) between this system and a queue; they seem the same *intuitively*.
- We can re-use the `Queues` but how do we re-use the composition mechanism?

At this point we examined the object oriented models and methods which are becoming prominent in software development. The semantics of composition were well understood, but the next case study was their first introduction to inheritance and subclassing.

4.6 Squares and Rectangles: Subclassing Formalised

This example was inspired from a long running 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. We spent 1 hour on this study, directly after working on the formal concepts of *extension* and *specialisation* as subclassing relations. Our goal was to see if the students would themselves discover that the question is unfair as it depends on the unspecified functionality of the program in which the shapes will be found.

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. 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 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. With a bit of pushing, they managed to say that `stretching` a shape may cause a problem because after you stretch a square it may longer be a square.

The lesson to be learned was the importance of the invariant in the square specification, which states that all sides are of equal length. Provided 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`.

5 Formal Methods and Tools

Throughout the case studies we emphasised the need for tools. We believe that it is very diffi cult to teach formal methods without a good tool support.

The most *difficult* question for an advocate of formal methods used to be *why formalise*? We propose the following response: *without formality it is impossible to automate*. Computers are very powerful automation tools for performing repetitive tasks which are beyond human capablilities: the problem is not in the complexity of any particular task, it is in the scale of the repetition. We formalise to structure complex problems in such a way as they can be solved through highly repetitive automation. Formal methods application depends on the tools for performing such automation. The spectrum of tools ranges from:

- Fully automated
- Highly automated, requiring some human interaction to help the machine to complete its task
- Partly automated, where the automation is there primarily to help a human structure their own behaviour in order to complete a task

It is important that students get to work with such a range of tools.

Students should have a means of learning through using. Lectures and tutorials do not give the students a chance to learn from practice, at their own rate. When programming, students like to be able to write their own programs (usually, very quickly) and see spectacular results (usually, immediately). The students could be said to be rewarded through their efforts. With formal methods, such an effort-reward cycle is harder to achieve and hence students do not have as much motivation to self-learn.

We would have liked to extend our use of tools: our goal is to provide one coherent framework for: interactive specifi cation, validation, transformation and verifi cation. Unfortunately, we had to split up our efforts for student-tool interaction. The following shows which parts of the course were taught using which formalisms:

- Specifying Structure — OMT and OO LOTOS
- Specifying Requirements — Abstract Data Types
- Validation — Animation of OO LOTOS Specifi cations
- Verifi cation — Completeness and consistency of algebraic specifi cations
- Transformation — Functional programs
- Verifi cation — Invariant proofs using PVS

The tool sets which we employed were: SMILE for LOTOS and ADT specifi cation, Atelier B and PVS for invariant specifi cation and proof, Object Geode for the graphical specifi cation, CAML for functional specifi cation and some *in-house* O-LSTS tools (written in JAVA) for animating our formal object oriented specifi cations.

6 Conclusions

Our approach to teaching formal methods tries to give an overall picture rather than concentrating on any one method, language or tool. We believe in letting the students discover the concepts and principles themselves, wherever possible. Our methods require *interactive* teaching between the lecturer and the students, flexible course structure, group work and frequent case studies. We emphasise the practical application of formal methods and try to teach the mathematics *on demand*.

This paper is intended to motivate a more collaborative approach to teaching formal methods. This is the first year we have taught such a course and all feedback is welcome. Formal methods need to be taught and they need to be taught well. By reporting on our imperfect attempts we hope to help others in their teaching of this difficult subject, and to help ourselves by better understanding the subject being taught and how it should be taught.

References

1. R.L. 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.
2. T. Bolognesi. Fundamental results in the verification of observational equivalence: a survey. In H. Rudin and West C.H., editors, *Protocol Specification, Testing and Verification VII*. North-Holland, 1988.
3. J. Bowen and M. Hinchley. Seven more myths of formal methods. *IEEE Software*, 12(4):34–41, 1995.
4. Dan Craigen, Susan Gerhart, and Ted Ralston. An international survey of industrial applications of formal methods. Nistgcr 93/626, U.S. Department of Commerce, Technology Administration, National Institute of Standards and Technology, Computer Systems Lab., Gaithersburg, MD 20899, 1993.
5. E. Cusack. Refinement, conformance and inheritance. In *Open University workshop on the theory and practice of refinement*, 1989.
6. R. DeNicola. Extensional equivalence for transition systems. *Acta Informatica*, 24:211–237, 1987.
7. A. Diller. *An Introduction To Formal Methods*. John Wiley and Sons, 1990.
8. J. Paul Gibson. Formal object based design in LOTOS. Tr-113, University of Stirling, Computing Science Department, Stirling, Scotland, 1994.
9. J.Paul Gibson. *Formal Object Oriented Development of Software Systems Using LOTOS*. Tech. report csm-114, Stirling University, August 1993.
10. Paul Gibson and Dominique Méry. Fair objects. In *OT98 (COTSR)*, Oxford, May 1998.
11. Anthony Hall. Seven myths of formal methods. *IEEE Software*, 7(5):11–19, 1990.
12. Rich Hart, Maltas. Teaching discrete mathematics in grades 7 –12. In *Mathematics Teacher*, volume 83, pages 362–367, 1990.
13. Douglas R. Hofstadter. *Godel, Escher, Bach: An eternal golden braid*. Penguin, 1987.
14. Reinfelds J. Logic in first courses for computing science majors. In *World Conference on Computers in Education*, pages 467–477, 1995.
15. Reinfelds J. A three paradigm first course for cs majors. *Proceedings of 26th ACM Tech. Symposium SIGCSE Bulletin*, 27(1):223–227, 1995.

16. Reinfelds J. A logical foundation course for cs majors. In *Australian Computer Science Education Conference*, pages 135–140, July 1996.
17. C. B. Jones. *Sytematic Software Development Using VDM*. Prentice-Hall International, 1986.
18. van der Hoeven G. Joosten S., van der Berg K. Teaching functional programming to first-year students. In *Journal of Functional Programming*, volume 3, pages 49–65, 1993.
19. Lambert. Using miranda as a first programming language. *Journal of Functional Programming*, 3(1):5–34, 1993.
20. Gérard Lévy. *Algorithmique Combinatoire: Méthodes Constructives*. DUNOD, 1994.
21. Helmut A. Partsch. *Specifi cation and Transformation of Programs: A Formal Approach To Software Development*. Springer-Verlag, 1990.
22. Reeves and Goldson. The calculator project - formal reasoning about programs. In Purvis M., editor, *Proceedings Software Engineering Conference SRIG-ET*, pages 166–173. IEEE Computer Society Press, 1995.
23. K.J.T. Turner. *Using FDTS: An Introduction To ESTELLE, LOTOS and SDL*. John Wiley and Sons, 1993.
24. J. Woodcock and J. Davies. *Using Z Specifi cation, Refi nement, and Proof*. Prentice Hall International Series in Computer Science. Prentice Hall, 1996.