Sculpturing Event-B Models with Rodin: *Holes and Lumps* in Teaching Refinement through Problem-Based Learning

J. Paul Gibson, Eric Lallet, Jean-Luc Raffy

Le département Logiciels-Réseaux (LOR), Telecom & Management SudParis,
9 rue Charles Fourier, 91011 Évry cedex, France
(L'Unité Mixte de Recherche -SAMOVAR- UMR 5157)
`pgibson@it-sudparis.eu`

Abstract. We present a Problem-Based Learning (PBL) approach to teaching formal methods, using Event-B and the Rodin development environment. This approach has arisen out of a gradual adoption, over a period of 3 years, of Rodin as the main teaching tool. Just as the concept of refinement is fundamental to what we are trying to teach, we demonstrate that it is also fundamental to the teaching process. Through analysis of a small number of PBL case-studies we argue that the changes to our teaching, supported by Rodin, have started to have a positive impact on our students meeting the specified learning objectives (course requirements). However, we also argue that much more work needs to be done in order to improve our teaching of formal methods. Inspired by the analogy between software design and sculpture, we conclude by proposing that formality holds the key to mastering the harmony between the "holes" and the "lumps" in our models.

Sculpture is the art of the hole and the lump.
[**Auguste Rodin, 1840 – 1917**]

1 Background: teaching Event-B with Rodin

The Event-B Rodin development environment[1] is central to our teaching of formal methods. The openness of the platform, combined with our research experience, motivated us to adopt it as early as possible in our teaching. Before Rodin, we had experience of teaching a variety of formal methods; with more recent teaching using B[2] and the Atelier-B tools. Another motivating factor is that our students are all familiar with the Eclipse[3] platform, on which Rodin is built, and this helps them overcome initial feelings of unfamiliarity which often arise from using formal methods tools for the first time.

In this paper we focus our discussion on the impact of our adoption of Rodin within a single optional module called *Langages formels et applications.* (We note that the problems have also been used with other student groups and in other institutions.) Within this module, 21 hours of teaching (direct contact between lecturer and students) is programmed specifically for teaching Event-B, and the students are also required to carry out a similar amount of self-study in their own time. The class is small — typically between 8 and 16 students. All students have already studied at least 1 year of computer science/software engineering, including foundational mathematics and programming.

In figure 1, we show how we have gradually adopted Rodin over a period of three years. In the first year, we prepared lecture slides based on Event-B case studies and tutorials that were available at the Rodin website. We also incorporated case studies inspired by formal methods research in different problem domains, for example: E-voting systems[4, 5], Distributed reference counting algorithms[6], Tree-structured File Systems[7] and the IEEE 1394 leader election protocol[8].

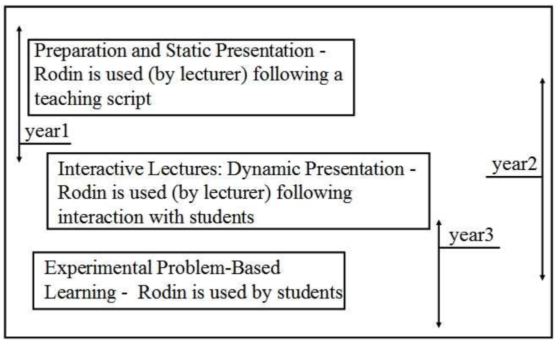

Preparation and Static Presentation - Rodin is used (by lecturer) following a teaching script
year1
Interactive Lectures: Dynamic Presentation - Rodin is used (by lecturer) following interaction with students
year2
year3
Experimental Problem-Based Learning - Rodin is used by students

Fig. 1. The Gradual Introduction of Rodin for teaching Event-B

We re-wrote previous practical work (moving from B to Event-B in the process) so that our models could better demonstrate the facilities provided by Rodin. We did not expect students to be able to use the Rodin tool: we were just becoming familiar with it ourselves. However, we did present Event-B models, refinement sequences, and proofs that had been prepared using Rodin. We followed a traditional teaching model where fundamental theory was presented before practical case studies. As a final step towards preparing for our second year of teaching with Rodin, a single lecture was presented in a more interactive style. The students were much more responsive to the lecturer "struggling to bend the tool to their way of thinking about the problems" (as they described it), rather than the lecturer presenting a solution that had been "baked earlier". Consequently, we decided that in the following year we would try to work on all the case study problems in a more interactive way.

In our second year, we continued to start the module with traditional lectures on foundational theory. However, the second half of the module was used to let the students interact with the Rodin tools (indirectly through the lecturer). Rather than the lecturer presenting models and proofs that had already been developed, the lecturer presented the problems to the students and then attempted to show the students how they could "test their solutions" by modelling them in

Event-B and analysing them using Rodin. At the end of the second year we ran a single PBL session with the students. They were given a problem where they had to design a solution in Event-B. They did not use the Rodin tool during the design problem; but after they had submitted their designs the lecturer demonstrated how the Rodin tool could have helped them to produce better (correct) designs. The students enjoyed the PBL approach — rather than learning some piece of theory from traditional lectures the students wrestled with a problem and discovered that they lacked some fundamental understanding that would help them to solve the problem. This approach is excellent for motivating the students, when the problems are well suited to the students meeting the learning objective. However, there is a great risk that the students do not learn simply from being exposed to the problem. Based on our previous work with PBL, we decided that the potential rewards of more fully adopting PBL outweighed the risk.

In the third year we did not start with any traditional lectures. On the first day the students installed Rodin and started to experiment with the tool. Initially, they built very simple models following the directions of the lecturer. However, quite quickly they stopped asking questions of the form "what would happen if we did this instead" to trying to find out, using Rodin, the answers themselves. At the end of each class (of duration 3 hours) the students would be advised to read material that would explain how/why the tool was reacting to their experimentation. They would also be given (optional) practical work that forced them to reflect on what they had learned during the session.

As we write this paper, we are half-way through the 3rd year of teaching Rodin. It is too early to analyse the global impact of our pedagogic changes: B to Event-B, using Rodin and PBL. In fact, as the three changes are very much interdependent — and there are many other more noisy parameters to take into account — it will be difficult to draw specific conclusions as to what causes any improvement (or deterioration) in our students' learning.

2 The PBL Teaching Method

Although there are many different definitions of PBL, the common factor in every one of them is that the problem acts as the catalyst that initiates the learning process. It is said that this way of learning encourages a deeper understanding of the material, rather than surface learning. As the problem is such a critical component of the learning process it is imperative that one uses *good* problems. In 2001, Duch identified five characteristics of what makes a PBL problem *good*[9]: (1) Effective problems should engage the students' interest and motivate them to probe for deeper understanding. (2) PBL problems should have multiple stages. (3) Problems should be complex enough that group co-operation will be necessary in order for them to effectively work towards a solution. (4) Problem should be open-ended. (5) Learning objectives of the course should be incorporated into the problems.

One of the major stumbling blocks to the implementation of PBL within any discipline is the lack of a good set of problems[10]. However, the discipline of software engineering and formal methods has many well-understood problems that have arisen out of industrial and research projects. Lecturers must be encouraged to use these problems (or parts of them) in their teaching. A recent proposal for weaving formal methods, through a software engineering programme, using problem-based learning (PBL)[11] provides good background and motivation for such a teaching approach in a software engineering programme, based on observations on how students solve problems using foundational software engineering techniques[12].

We note that the 5th of Duch's characteristics for a good PBL problem is defined in terms of learning objectives. Without explicit statement of learning objectives it is difficult, if not impossible, to evaluate and analyse the effectiveness of PBL, in general, and specific problems, in particular.

3 Learning Objectives

A main weakness when teaching design is that students fail to understand that design is a dynamic process and not just a sequence of models. This is particularly important when teaching formal methods through design problems.

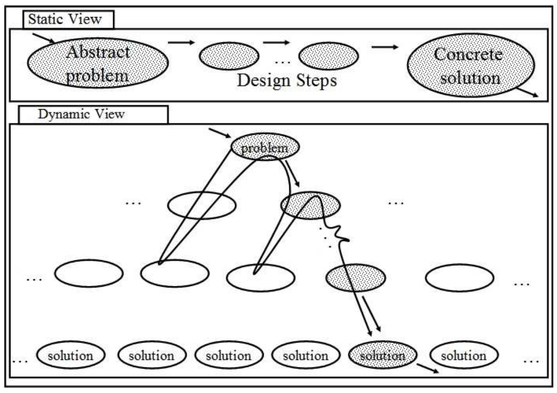


Fig. 2. Design as A Dynamic Process: The Learning Objectives

Figure 2 provides a graphical view of our main learning objectives when treating design as a process: (1) to be able to build models at different levels of abstraction, (2) to be able to prove that models at all stages of development are well defined, (3) to be able to validate that models capture precisely the needs of the client, (4) to be able to verify that each design step (from abstract to concrete) is correct, and (5) to be able to manage the process of rolling back a design decision.

We note that these objectives are generic in the sense that the modelling language and modelling tools are unspecified. Our secondary learning objectives are that the students are able to meet the main objectives when modelling with Event-B, and that they are able to use the Rodin tool for (automated) support.

3.1 Model at different levels of abstraction

Students are expected to already know about nondeterminism. We expect them to be able to build requirements models that use nondeterminism to facilitate implementation freedom. Our main objective is for them to be able to remove such nondeterminism through refinement. Further, they should learn how to reverse engineer a concrete model to something more abstract.

3.2 Well-defined Models

Students are expected to already know, in general, how theorem provers work. They are also expected to be able to carry out simple mathematical proofs by hand. Our learning objective is for them to be able to apply this knowledge when using the automated theorem proving support provided by Rodin. Our goal is to provide a problem through which the students learn about well-definedness and also learn, through experimentation, how Rodin can be used to prove that Event-B models (and parts of the models) are well-defined.

3.3 Validation — testing understanding of requirements (formally)

Students are expected to already understand that many problems occur in software engineering because of poorly understood requirements[13]. They are also expected to know that most common validation techniques involve testing an executable system (often a prototype) with the client; and that there are weaknesses to this approach. Our goal is to show that formal models offer an approach to validation that is complementary to executing code. We aim to provide them with a problem where the students quite naturally test their understanding of requirements through the formulation of theorems to be proved.

3.4 Correct Design

Students are expected to already have studied design (usually with a modelling language like the UML[1]). They must understand the role of design in bridging the gap between the problem (requirements) and the solution (implementation). Our goal is that students learn what it means for a design to be correct[14], and that they can use RODIN to prove three fundamental properties of a machine: (1) Invariants are respected. (2) Termination (where required). (3) Deadlock freeness (for interactive systems).

3.5 Design As A Process

A key objective is that students understand that design is a dynamic process, represented by a tree of decisions and compromises. (We also hope that the

[1] It is pleasing to note, for potential future exploitation in our teaching, the research and development of a Rodin plug-in for integrating Event-B and the UML[1]

students see that this tree representation is an abstraction of what happens in real software development.) The design documentation should not just be a record of the sequential final path in this tree that links the problem to the particular chosen solution: it should be a record of every design decision that was taken and why (including the decisions that were changed, i.e. rolled back). Further, the students must learn that there is as much value in the links in the design tree (which represent the correctness of the design steps) as there is in the nodes (the models).

4 Problems Presented

In this section we review a subset of the problems that have been presented to the students in order to meet specific learning objectives. We comment on students' behaviour whilst interacting with the problems, with particular emphasis on their use of the Rodin tools.

4.1 Well-defined Models: The Purse Problem

In our search for interesting problems, it was noted that the notion of a wallet (of money) had proven to be a good pedagogic case study[15]. This inspired us to consider a purse as the basis for our PBL case study. The following requirements were presented to the students:

1. A purse contains coins.
2. Coins are positive integers, but not all integers have a corresponding coin.
3. We wish to start with an empty purse, containing no coins.
4. We allow 3 operations: (a) initialise a purse to being empty (containing no coins), (b) add a coin, and (c) pay a certain (integer) sum by removing an appropriate number of coins from the purse.

Figure 3 shows a graphical representation of the problem that was presented to the students to complement the textual requirements.

It is interesting to note how the students tried to model the `Purse` using Event-B. Firstly, we witnessed the problem of confusing sets with bags as discussed by Habrias[15]. Once students realised that the problem required more than a set of coins (represented as integers) most of them they quickly defined a `Purse` to be a total function from coins to integers. (Some students also chose to specify `Purse` as a partial function from coins to integers, arguing that if a coin was not in the domain then there were no coins of that value in the `Purse`.) The students struggled to specify a generic `Purse`, parameterised by any set of coins. They knew that this type of specification should be possible but had to be shown how to specify this using an abstract `COIN` set. It was pleasing to see that many of the students then specified the notion of an empty purse in a similar, generic fashion, as shown in figure 4.

Most students then thought about the operation for paying a certain sum and decided that it was too difficult to specify directly. They were encouraged to

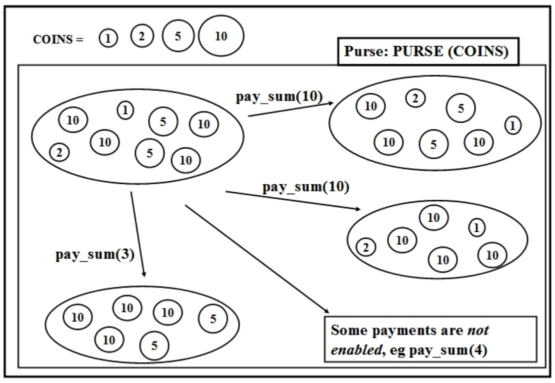
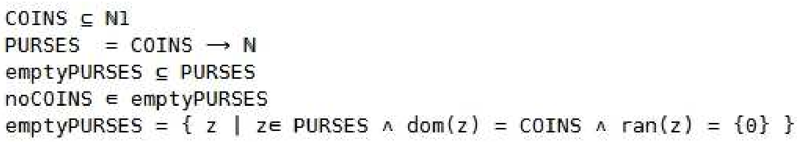


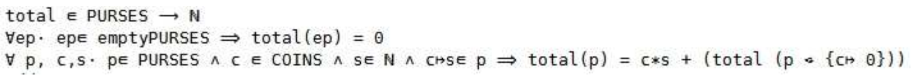
Fig. 3. The Purse **pay_sum** Behaviour



```
COINS ⊆ N1
PURSES  = COINS → N
emptyPURSES ⊆ PURSES
noCOINS ∈ emptyPURSES
emptyPURSES = { z | z∈ PURSES ∧ dom(z) = COINS ∧ ran(z) = {0} }
```

Fig. 4. A generic specification of an empty purse

think about it in an abstract, nondeterministic, fashion. However, most of them thought that this meant decomposing the payment into component parts. One of the most common ways of doing this was for students to specify the notions of “total” and “remove” (two key terms found in the textual requirements). An example of how a student specified `total` is shown in figure 5.

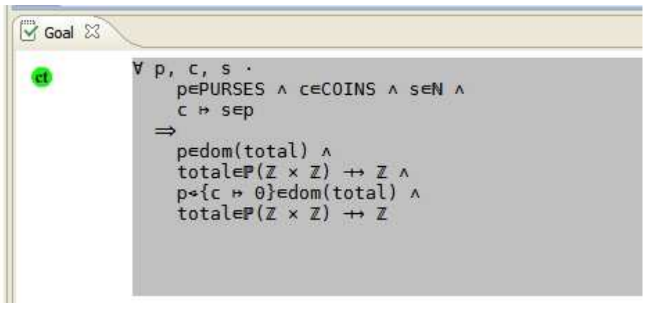


```
total ∈ PURSES → N
∀ep· ep∈ emptyPURSES ⇒ total(ep) = 0
∀ p, c,s· p∈ PURSES ∧ c ∈ COINS ∧ s∈ N ∧ c↦s∈ p ⇒ total(p) = c∗s + (total (p  {c↦ 0}))
```

Fig. 5. The introduction of a function to calculate the total

At this stage, the lecturer pointed out that the Rodin tool was generating proof obligations with regard to the well-definedness of their `total` specifications, as shown in figure 6.

The students experimented with the Rodin tool in order to see which proofs were discharged automatically and which required interaction. Although they did not know how the prover worked (and had received no lectures on the subject)

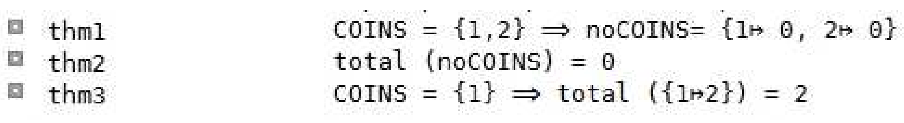


```
∀ p, c, s ·
    p∈PURSES ∧ c∈COINS ∧ s∈ℕ ∧
    c ↦ s∈p
  ⇒
    p∈dom(total) ∧
    total∈ℙ(ℤ × ℤ) ⇸ ℤ ∧
    p<+{c ↦ 0}∈dom(total) ∧
    total∈ℙ(ℤ × ℤ) ⇸ ℤ
```

Fig. 6. The proof obligation generated for the specification of total

they were able to carry out some proofs simply by "randomly" clicking and instantiating.

By encouraging students to examine different specifications of `total` it often arises that students ask how they can test that their specifications "really work". In essence, they are asking how they can validate that the meaning of `total` corresponds to the requirements. At this stage the lecturer suggests that they formulate simple use cases. The students are able to express the fact that they want to test, for example: (1) the total of any empty purse must be 0, and (2) the total of a purse containing two 1c coins should be 2. It was surprising (to us) that, in general, students manage to express these as theorems only after receiving help from the lecturer, as in figure 7. The main problem was due to us not having yet covered the foundational material explaining fundamental concepts such as axiom, theorem, proof, completeness, consistency, etc ...



```
thm1        COINS = {1,2} ⇒ noCOINS= {1↦ 0, 2↦ 0}
thm2        total (noCOINS) = 0
thm3        COINS = {1} ⇒ total ({1↦2}) = 2
```

Fig. 7. Using theorems to validate understanding and specification

In the next example, of the family tree, we expect the students to follow a similar validation process.

4.2 Validation: The Family Tree Problem

The problem of specifying relations between people in order to identify families has been used by a large number of lecturers. Our goal is to use the same example but to force the students to work within a particular view that needs to be validated: we consider only relations between humans that are alive, and we wish to enforce that each person can have 0,1 or 2 parents that are still alive. We present the problem by the tree shown in figure 8.

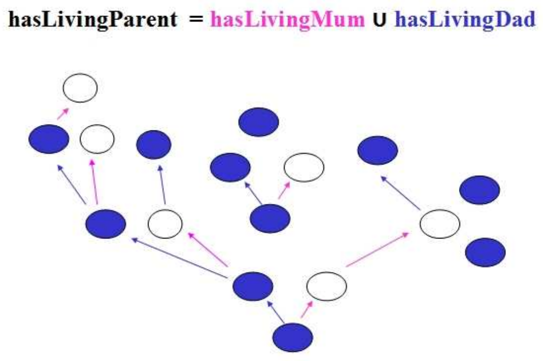
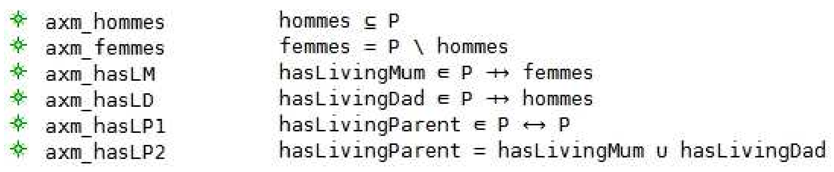


Fig. 8. Graphical representation of the required Parent behaviour

The students, having learned from their experience of the `Purse` problem validate their parent specifications — an example is given in figure 9 — with simple theorems.



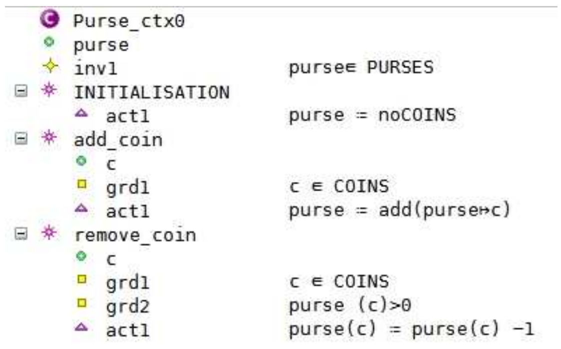
axm_hommes	hommes ⊆ P
axm_femmes	femmes = P ∖ hommes
axm_hasLM	hasLivingMum ∈ P ⤀ femmes
axm_hasLD	hasLivingDad ∈ P ⤀ hommes
axm_hasLP1	hasLivingParent ∈ P ↔ P
axm_hasLP2	hasLivingParent = hasLivingMum ∪ hasLivingDad

Fig. 9. Formal Specification of Parent Requirements

It was interesting to note that the students went on to specify relations like `brother`, `cousin`, `aunt`, etc .... However, none of them validated (or attempted to validate) that their family tree did not contain any cycles!

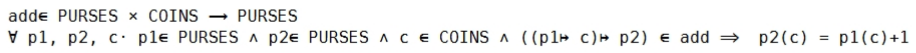
4.3 Correct Design: The Purse Revisited

In the process of specifying the `Purse` behaviour we noted that the first design step — of pairing a machine with a context — led to some interesting design decisions. For example, we saw two different specification styles — see figure 10 — for events that update the state of the purse, by adding and removing coins. Some students used a style where the state updates of the machine were specified axiomatically in the context, for example: the `add_coin` event uses an `add` function that has been specified in the context `Purse_ctx0` (see figure 11). Whilst others used a more operational style (as for event `remove`).



```
Purse_ctx0
purse
inv1                    purse∈ PURSES
INITIALISATION
    act1                purse ≔ noCOINS
add_coin
    c
    grd1                c ∈ COINS
    act1                purse ≔ add(purse↦c)
remove_coin
    c
    grd1                c ∈ COINS
    grd2                purse (c)>0
    act1                purse(c) ≔ purse(c) −1
```

Fig. 10. Adding and Removing a coin from a Purse



```
add∈ PURSES × COINS ⟶ PURSES
∀ p1, p2, c· p1∈ PURSES ∧ p2∈ PURSES ∧ c ∈ COINS ∧ ((p1↦ c)↦ p2) ∈ add ⇒  p2(c) = p1(c)+1
```

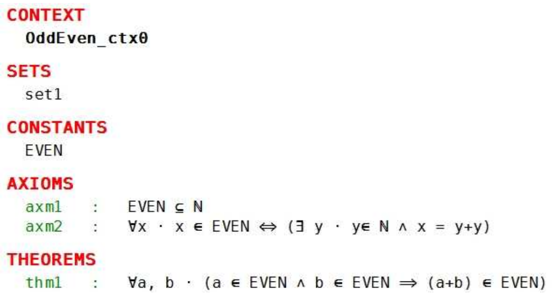
Fig. 11. The add function defined in the Purse context

Without going into details, this approach requires additional work when proving the correctness of the context, but leads to a simple proof that the invariant is respected by the `add_coin` event (in the machine). Contrastingly, the `remove_coin` event's action is specified directly (without an additional "worker" function from the context). This approach means that the proof that `remove_coin` respects the invariant cannot re-use any properties of `remove` that could have been specified in the context.

4.4 Design As A Process: The OddEven Problem

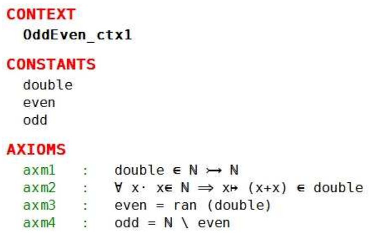
We present the students with the problem of specifying `Odd` and `Even` numbers. We tell them that these specifications will be required in a later machine, but do not tell them precisely how. Even for such a simple problem, students typically produce different specifications. For example, see figures 12, 13 and 14.

We observed that the students, whilst in the process of proving properties of the subsequent machine, chose to change the way in which they specified `Odd` and `Even`. When asked about this, one of the more interesting replies was: "We



```
CONTEXT
  OddEven_ctx0
SETS
  set1
CONSTANTS
  EVEN
AXIOMS
  axm1   :   EVEN ⊆ N
  axm2   :   ∀x · x ∈ EVEN ⇔ (∃ y · y∈ N ∧ x = y+y)
THEOREMS
  thm1   :   ∀a, b · (a ∈ EVEN ∧ b ∈ EVEN ⇒ (a+b) ∈ EVEN)
```

Fig. 12. A first specification of Odd and Even



```
CONTEXT
  OddEven_ctx1
CONSTANTS
  double
  even
  odd
AXIOMS
  axm1   :   double ∈ N ↣ N
  axm2   :   ∀ x· x∈ N ⇒ x↦ (x+x) ∈ double
  axm3   :   even = ran (double)
  axm4   :   odd = N \ even
```

Fig. 13. A second specification of Odd and Even

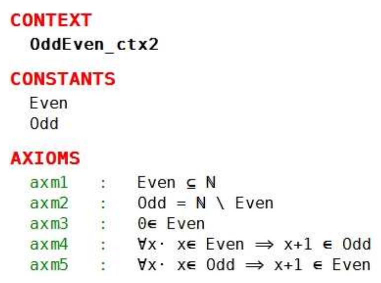
have learned the importance of structuring our code to make it easier to test; so why not restructure our specifications to make them easier to verify?".

4.5 Refinement and Design: The "Centralised Leader Election" Problem

The very last problem that we presented to the students had the objective of testing whether they were able to reason about the correctness of different designs through refinement of an abstract machine. The problem presented to them was a simplification of leader election:

> Given a team of players, there must be a single unique captain. There is a single event which corresponds to changing the captain. Propose alternative designs to implementing this simple system. Use the Rodin toolset to reason about the correctness of the designs.

We have observed four different types of "solution" to the problem: (i) Initial Machine too concrete and no refinement, (ii) Initial Machine too concrete and a correct refinement, (iii) Initial Machine at appropriate level of abstraction but unable to specify design as a refinement, and (iv) Initial Machine at appropriate level of abstraction and alternative correct designs specified as refinements.



```
CONTEXT
  OddEven_ctx2
CONSTANTS
  Even
  Odd
AXIOMS
  axm1   :   Even ⊆ ℕ
  axm2   :   Odd = ℕ \ Even
  axm3   :   0∈ Even
  axm4   :   ∀x· x∈ Even ⇒ x+1 ∈ Odd
  axm5   :   ∀x· x∈ Odd ⇒ x+1 ∈ Even
```

Fig. 14. A third specification of Odd and Even

Approximately half the class started with machines that were too concrete in the sense that they precluded some reasonable implementations. In all but one of these cases, the students who started with a concrete design were unable to refine it. Consequently, they reasoned about the design correctness informally. None attempted to reverse engineer a more abstract machine.

In a single case, the students started with a team of players where each player had an integer counting the number of times they had been selected as captain. They specified the (invariant) requirement that no player could have a `count` more than 1 larger than any other player's `count`. Then, in the abstract machine they specified that the change captain event never picked a captain whose `count` was already bigger than any other player's `count`. They then refined this event by introducing a captain `queue` where selecting a new captain popped the current captain from the front of the `queue` and pushed this player onto the back of the `queue` — so that the sequence in which captains were chosen would follow a repetitive cycle. This machine is a refinement of the first as it removes the nondeterminism in the event which chooses the captain (after the first cycle).

The remainder of the class specified the initial abstract machine at an appropriate level of abstraction. Two groups chose to specify simple solutions where the position of captain always alternated between the same 2 team members. Although this is, perhaps, the simplest design, neither of these groups were able to demonstrate that this design is correct. They attempted to prove the more concrete machine to be a refinement of the abstract machine, but failed.

The best solutions offered alternative designs and demonstrated their correctness. None of these offered a sequence of refinement steps (but this was not explicitly asked of the students). It was disappointing that only about one third of the students were able to address the problem in this way.

5 Refining our Teaching: what needs to be changed?

In order to improve our teaching formal methods[2] it is important that we learn from the students[16]. With PBL, in particular, one must take great care when

[2] More generally, student feedback is crucial in improving the teaching in any discipline.

using quantitative and qualitative analysis to evaluate the effectiveness of the problems[17].

Much like software engineers who refine their models for implementation, formal methods lecturers need to refine their teaching models. When there is a mismatch between what is required and a proposed solution then there are three possibilities: (1) the solution is correct with respect to the requirements specification yet the specification misrepresents the requirements, or (2) the specification correctly represents the requirements but the solution is not correct with respect to the specification, or (3) a combination of the two previous possibilities. In general, when a solution is acceptable it is because the initial requirements were correctly specified and the implementation was correct with respect to these requirements. (In theory, it may be possible that the solution meets the requirements despite the fact that the specification is incorrect. However, this situation is not desirable even though the client may be happy in the short term!)

There are several options when a problem is not meeting a specific learning objective: (1) Replace the problem with something completely different. (2) Fix the problem by making minor changes. (3) Change the learning objective.

This feedback into our teaching is critical in PBL but it is problematic because: (1) The frequency of change is usually tied to the academic calendar. (2) The mapping relation between learning objectives and problems is not (usually) a bijection, though it should be a total surjection. (3) Analysis of the effectiveness of problems should be done using more than 1 class of students. (4) Developing new problems is time-consuming. The simplest way to overcome these issues is to share and re-use problems between different lecturers and programmes.

Once a problem has been developed that is deemed to be effective, it is very important that one does not break its effectiveness through making change. Lecturers would greatly appreciate a formal notion of refinement with respect to their teaching material. Thus, they could make (verifiable) changes to existing problems knowing that such changes do not compromise their effectiveness (at meeting the learning objectives). Of course, this is currently beyond the state-of-the-art in educational research! However, as teachers we must aspire to achieving such refinements of our problem designs: it will improve our teaching and reduce our workload.

6 Holes and Lumps in our Event-B Models

Event-B models can be judged on a number of different criteria: (1) Well-definedness, which can be checked without knowing the intended purpose of the model. (2) Fitness for purpose, which can be checked against required behaviour. (3) Level of abstraction, which reflects whether design decisions are being taken too early/late in the development process. (4) Maintainability/Reusability, which represents how much of the modelling work (including the proofs) can be re-used if requirements change.

Design is not a prescriptive process. Students need to learn that building good designs (in Event-B) requires experience, good judgement and good fortune. Many of the students produce poor designs because their models are too rich in detail. They miss the importance of keeping things simple. A key insight is that the best students are quite comfortable with leaving details to later stages in the development process. These "holes" correspond to abstraction. Subsequent refinements may (partially) fill in the holes. The most valuable Event-B designs are those that bridge the gap between the requirements specifications that are relatively easy to model, using lots of nondeterminism, and the determinisitic models that are directly implementable, using traditional programming languages. Our experience shows that students can quite easily produce Event-B models at these extremes of the abstraction continuum but find it very difficult to produce the intermediate design steps (that are necessary in establishing correctness).

A second issue that needs to be addressed is one of composition. Event-B, due to its refinement mechanism, has proven to be successful in teaching correctness-by-construction. However, we have fears that it is not so well-suited to reasoning about composition of systems. In fact, our students often commented on having difficulties in adding functionality (features) to already developed (and verified) machines. Further, they observed that they would like to be able to synchronise machines[3] through some form of shared events. They also claimed to miss the high level composition mechanisms that they were used to having in their favourite OO programming languages.

Design is also about knowing what needs to be added and where. Modelling and managing these "lumps" is a learning objective that we have yet to try and meet (when teaching Event-B with Rodin). There has been research in extending Event-B with richer composition mechanisms, but we are not yet ready to use them in the classroom.

7 Conclusions

We believe that our PBL case-studies, using Rodin, are improving the way in which we teach formal methods: (1) Students are happy to experiment with their models and proofs. (2) Students are more motivated by working on problems — and often spend much more time than required on self-study. (3) The students were able to better understand the foundational material presented to them (in traditional lecture format) as they could relate the theoretical concepts to the operation of the Rodin tools — with particular interest in how the provers work when discharging obligations automatically, and how to best carry out proofs interactively.

However, we need to build a more extensive problem set, and improve our feedback mechanisms for evaluating and improving problems. A major issue is the specification of our learning objectives: if we want to share problems then we need to be able to find common agreement on learning objectives. Such agreement would be complementary to the development of a formal methods body of knowledge[19]. Inspired by the analogy between software design and sculpture, we conclude by proposing that formality holds the key to mastering the harmony between the "holes" and the "lumps" in our models.

[3] This is similar to the integration of CSP and B[18].

References

1. Abrial, J.R., Butler, M., Hallerstede, S., Voisin, L.: A roadmap for the Rodin toolset. In: Abstract State Machines, B and Z, First International Conference ABZ 2008. Volume LNCS 5238. (September 2008) 347–351
2. Abrial, J.R.: The B Book - Assigning Programs to Meanings. Cambridge Univ. Press (1996)
3. des Rivières, J., Wiegand, J.: Eclipse: A platform for integrating development tools. IBM Systems Journal **43**(2) (2004) 371–383
4. Cansell, D., Gibson, J.P., Méry, D.: Formal verification of tamper-evident storage for e-voting. In: Software Engineering and Formal Methods (SEFM 2007), IEEE Computer Society (2007) 329–338
5. Cansell, D., Gibson, J.P., Méry, D.: Refinement: A constructive approach to formal software design for a secure e-voting interface. Electr. Notes Theor. Comput. Sci. **183** (2007) 39–55
6. Cansell, D., Méry, D.: Formal and incremental construction of distributed algorithms: On the distributed reference counting algorithm. Theor. Comput. Sci. **364**(3) (2006) 318–337
7. Damchoom, K., Butler, M., Abrial, J.R.: Modelling and proof of a tree-structured file system. In: ICFEM 2008. Volume LNCS 5256., Springer (October 2008) 25–44 Springer LNCS 5256.
8. Rehm, J., Cansell, D.: Proved development of the real-time properties of the ieee 1394 root contention protocol with the event b method. In Ameur, Y.A., Boniol, F., Wiels, V., eds.: ISoLA. Volume RNTI-SM-1 of Revue des Nouvelles Technologies de l'Information., Cépaduès-Éditions (2007) 179–190
9. Duch, B. In: Writing Problems for Deeper Understanding. Stylus Publishing, Sterling, Virginia (2001) 47–53
10. Tien, C., Chu, S., Lin, Y.: Four phases to construct problem-based learning instruction materials. In: PBL In Context: Bridging work and Education, Tampere University Press (2005) 117–133
11. Gibson, J.P.: Weaving a formal methods education with problem-based learning. In: 3rd Int. Symposium on Leveraging Applications of Formal Methods, Verification & Validation. Volume 17., Springer-Verlag, Berlin Heidelberg (2008) 460–472
12. Gibson, J.P., O'Kelly, J.: Software engineering as a model of understanding for learning and problem solving. In: ICER'05: Proceedings of the 2005 international workshop on Computing Education Research, ACM (2005) 87–97
13. Gibson, J.P.: Formal requirements engineering: Learning from the students. In: Australian Software Engineering Conference, IEEE Comp. Soc. (2000) 171–180
14. Gibson, J.P., Lallet, E., Raffy, J.L.: How do I know if my design is correct? In: Formal Methods in Computer Science Education (FORMED). (March 2008) 59–69
15. Habrias, H.: Teaching specifications, hands on. In: Formal Methods in Computer Science Education (FORMED). (March 2008) 5–15
16. Gibson, J.P., Méry, D.: Teaching formal methods: Lessons to learn. In Flynn, S., Butterfield, A., eds.: IWFM. Workshops in Computing, BCS (1998)
17. O'Kelly, J., Gibson, J.P.: PBL: Year one analysis — interpretation and validation. In: PBL In Context — Bridging Work and Education. (2005)
18. Butler, M.J., Leuschel, M.: Combining csp and b for specification and property verification. In Fitzgerald, J., Hayes, I.J., Tarlecki, A., eds.: FM. Volume 3582 of Lecture Notes in Computer Science., Springer (2005) 221–236
19. Oliveira, J.N.: A survey of formal methods courses in European higher education. In Dean, C.N., Boute, R.T., eds.: TFM'04. Volume 3294 of Lecture Notes in Computer Science., Springer-Verlag (2004) 235–248