Formal Methods: Never Too Young to Start

J Paul Gibson [1]

Le département Logiciels-Réseaux (LOR), Institut National des Télécommunications, Evry, France

Abstract

In many countries around the world, there is a crisis in the teaching of mathematics and computer science. Governments have tried to address the problem by investing in computers in schools; when they should have invested in teaching computer science in schools. Formal methods bridge the boundary between computing and mathematics in a natural way. Through our experience of teaching algorithmic thinking in schools, young children have been observed using concepts such as refinement, proof, abstraction, complexity, non-determinism, equivalence, etc... in their own reasoning about problems. We argue that this ability needs to be better leveraged in order to improve both the teaching of mathematics but also to improve childrens' understanding of computer science as a discipline in its own right. We give concrete examples of the type of formal methods teaching that succeeds.

Key words: Schools, Computer Science, Proof, Algorithms

1 Introduction: Why teach formal methods in schools?

The current state of mathematics teaching around the world is causing problems for the teaching of computer science. Children are taught "mathematics" at all levels of school education, yet computer science lecturers continue to struggle with undergraduate students whose mathematical abilities are poor. Furthermore, many of these students have been (mis)led in their schools to believe that computer science does not require mathematics. The recent introduction of computers into schools has not, in general, improved the situation: computers are mostly used as a tool to support the teaching of other subjects (including mathematics); whilst often giving a false impression that computer science is using a computer. Some schools, in an attempt to teach computer science, have introduced children to programming. We believe that this is a

[1] Email: `paul.gibson@int-evry.fr`

step forward, but it carries the risk that children think that programming is computer science, and that computer science is programming. We believe that the best way of introducing children to computer science does not require a computer: it requires the teaching of rigorous (formal) reasoning about computations and algorithms.

This paper reports on the teaching of formal methods to children as young as seven years old. It supports the well-established view that there are advantages in teaching mathematics constructively. For example Clements stated in 1990[4]: "Educational research offers compelling evidence that students learn mathematics well only when they construct their own mathematical understanding." The work reported is motivated by our own experiences in teaching computer science to young children: Java programming[8], the Computer Science Unplugged Tutorials[1], and problem based learning (PBL)[13,12]. It builds on research that suggests that formal software engineering concepts form the basis for how children learn to solve problems[9].

In "How to Solve It: A New Aspect of Mathematical Method", Polya[15] states: "A good teacher should understand and impress on his students the view that no problem whatever is completely exhausted." Our approach has been to take problems that one would normally see in university and to rework them for school children. The children are then encouraged to take the problems in whatever direction they wish, and as far as they want. In this way, young children quickly identify fundamental concepts in computer science.

This paper reports on some of the case studies with which we have had most success. We do not claim to have carried out a verifiable educational experiment: we report on observations that have been made over a number of years through interaction with hundreds of children (aged 7 to 18).

2 Learning Theory and Models

There are hundreds of well-published complementary, and competing, theories of learning. The highly cited review by Hilgard and Bower[11], published over half a century ago, is a good introduction to the foundations of learning theory. In this paper, we focus on the well-accepted theories that have had most influence on our own research into formal reasoning and problem solving.

Cognitive structure is the concept central to Piaget's theory. (See the work by Brainerd[2] for a good overview and analysis.) Piaget's most interesting experiments, with respect to the work presented in this paper, focused on the development of mathematical and logical concepts. His theory has guided teaching practice and curriculum design in primary (elementary) schools in the last few decades. However, his work predates the development of computer science as a discipline and it is therefore unsurprising that it does not make reference to any formal methods concepts. Piaget's theory is similar to other *constructivist* perspectives of learning (e.g., Bruner [3]), which model learning as an active process in which learners construct new concepts upon their current knowledge and previous experience.

Similarites can be seen between the constructivist view and the *theories of intelligence* such as proposed by Guildford's *structure of intellect* (SI) theory [10] and Gardner's *multiple intelligences*[7]. Typically, these theories structure the learning space in terms of skills, for example: reasoning and problem-solving, memory operations, decision-making, and language. These skills do not explicitly mention computation and proof, but one can see that there are strong overlaps.

Piaget's ideas also influenced the work by Seymour Papert in the specific domain of computers and education[14]. Papert argues that children can understand concepts best when they are able to explain them algorithmicaly through writing computer programs. We believe that they learn better when working at higher levels of abstraction, and that what they should really be doing is mathematical modelling.

Alan Schoenfeld argues that understanding and teaching mathematics should be treated as problem-solving [16]. He identifies four skills that are needed to be successful in mathematics: proposition and procedural knowledge, strategies and techniques for problem resolution, decisions about when and what knowledge and strategies to use, and a logical *world view* that motivates an individual's approach to solving a particular problem. In this respect, Schoenfeld has quite nicely identified fundamental aspects of formal methods in computer science and software engineering.

3 Learning Objectives: review of our sessions

In each of the sessions that we run in the schools, we are mindful that our goal is that the children learn fundamental concepts. Following the PBL philosophy, we do not explicitly teach the children about these concepts; we help the children to discover them through interaction with a specific problem. Each problem has the goal of the students discovering at least one of the following:

- *Proof* — the evidence or argument that compels one to accept an assertion as true.
- *Theorem* — a proposition that is true in all cases.
- *Conjecture* — an unproven proposition for which there is some sort of empirical evidence.
- *Constructive proof* — demonstrates the existence of a mathematical object with certain properties by giving a method (algorithm) for creating such an object.
- *Algorithm* — any procedure involving a series of steps that is used to find the solution to a specific problem.
- *A deterministic algorithm:* behaves predictably. Given a particular input, it will always produce the same output, and the underlying machine will always pass through the same sequence of states.

- *Correctness* — an algorithm can be proven to be correct with respect to a specification.
- *Refinement* — the verifiable transformation of an abstract (high-level) formal specification into a concrete (low-level) executable program. It guarantees the correctness of the program by construction.
- *Invariant* — an expression whose value does not change during algorithm execution; which can implement a required safety property in the specification
- *Computational Complexity* — the scaleability of algorithms with respect to the use of resources (typically time and space).

It is beyond the scope of this paper to analyse all of these learning objectives, and to demonstrate how our sessions help children to reach them. Rather, we give — in the next 3 sections — examples of sessions that we run with the youngest children (aged seven to nine). These illustrate how the formal methods concepts arise quite naturally out of the games that we play.

4 Parity: Algorithms, Verification and Proof

For this problem children need to be familiar with the concepts of even and odd numbers. They do not need to be able to add (although they think they do). We present the children with sums and ask if the answers are odd or even numbers. For example: *Is the answer to the sum 1 + 2 + 2 + 1 + 2 + 1 + 1 +1 + 1 + 2 +1 even or odd*? The children typically follow the algorithm:

```
add up the numbers
decide if the sum is odd or even (by looking at the last digit)
```

They then arrive at (hopefully) the answer 15 and then they say that 15 is odd (because 5 is odd). We then ask the children for more complicated sums and claim that we can perform the task faster than they can (even if they use a calculator): *1285 + 45362 + 12987 + 367235 + 12 + 887877 + ...* Of course, we have a trick, which we ask the students to try and work out.

We then bring in some (younger) students from a class that we have taught how to recognise odd and even numbers (represented as a string of digits). We give them a torch (flashlight). They then perform the following algorithm (after practising with us until we know they execute it correctly):

```
In the beginning the light (which can be switched) is off
For each number to be added:
 If light is off and the number odd  then switch
  else if light is on and the number even  then switch
   else do not switch
When no more numbers left
  If light is off then answer is even else the answer is odd.
```

The older children are amazed that the younger children — who cannot count (very well) — are quicker than them at doing the sum. Of course, they never really do the sum ... this is the trick that we want the older children to discover. We then ask the younger children to explain the algorithm to the older children. The younger children, in general, know the algorithm but do not understand what it is doing. The older children do not know the algorithm but know what it is supposed to do. Typically, they question the correctness of the algorithm by asking if the trick will always work:

"They did it once (for a very difficult example) so it must work", or
"Ask them to do it a few more times to make sure they are not just guessing", or
"I will not ever be convinced until I know exactly what they are doing"

In order to convince the sceptical students we ask them to consider the following table:

```
Odd + Odd = Even          Even + Even = Even
Odd + Even = Odd          Even + Odd = Odd
```

The students are then asked to construct a table for the addition of three numbers, and to see if they can use the two tables to find out something that may help us how to understand whether the algorithm works (is correct). Typically, they identify the associativity and commutativity of addition (over odds and evens). At this point we introduce some physical elements (toys) in order to help them play with the problem. In this case, we used different coloured bricks that can stick together. We demonstrate the addition of the numbers 1 and 2, as shown in figure 1.

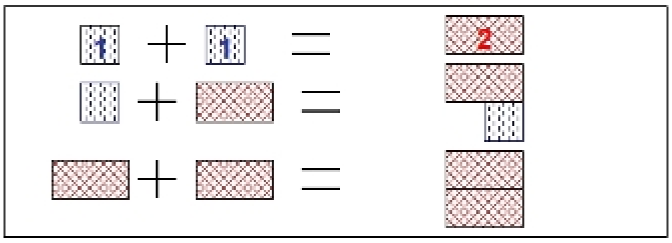

Fig. 1. Addition using Sticky Coloured Bricks

We ask the students to demonstrate/prove the elements of the addition table, for example: `Odd + Odd = Even`. In general, we see the sort of "proof" as illustrated in figure 2. Whether the physical manipulation of the sticky bricks constitutes a proof is open to question. What should not be questioned is that children are able to demonstrate why the parity addition trick works.

5 Primes: Algorithms, Correctness and Complexity

The goal of this exercise is to test whether the students can reason about the computational complexity of an algorithm. Furthermore, we would like them to see that checking the answer of a computation is often simpler than finding the correct answer. We use the classic problem of testing whether an integer is prime. The children do not need to know how to multiply or divide but they do need to know about rectangles. We use sweets (wrapped in paper)

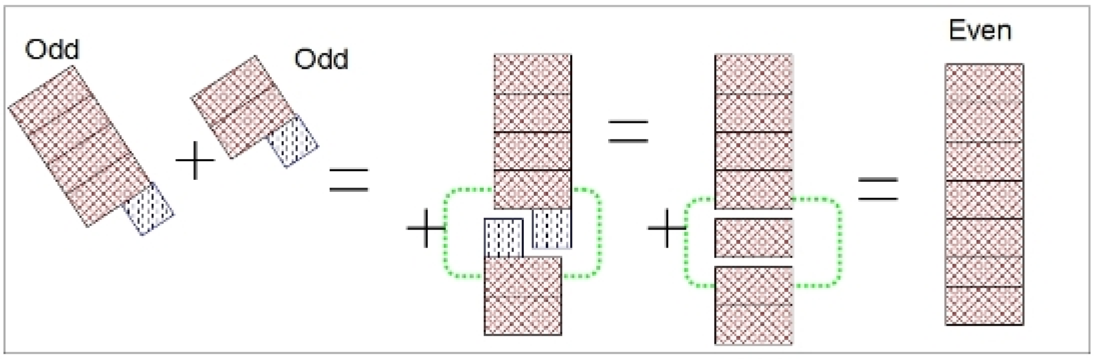

Fig. 2. Parity "Proof"

for the concrete representation of numbers (in unary notation). In figure 3, we see how the children check whether the numbers 7 and 6 are prime.

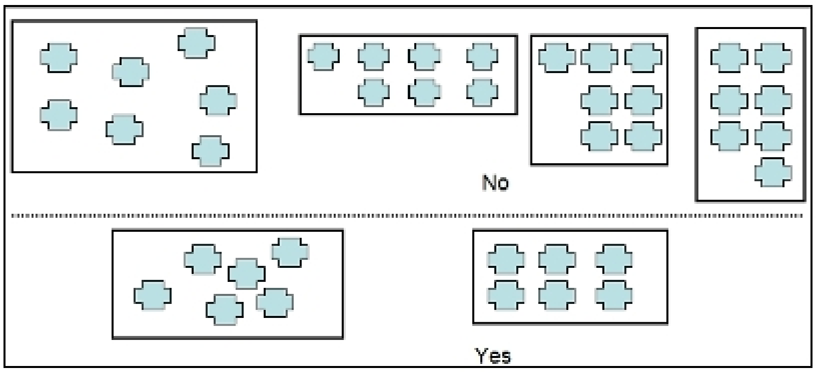

Fig. 3. Primes Algorithm: Can we make a rectangle?

As a game, we provide a larger number of sweets and ask the children to race against each other (often in teams) in order to construct a rectangle. The winning team gets to eat their sweets. As a challenge, we give them 17 sweets and see that they get quite frustrated. We explain that when they cannot make a rectangle then the number of sweets is said to be prime. When playing the game, some of the children do not trust us when we say that another team has found the rectangle before they have. Consequently, we tell them the width and height (multiples) and they see immediately that they are quicker at checking that the answer is correct than at trying to find the answer themselves. After playing this game, the older children identify classic algorithms for primality and even manage to execute them in parallel using different players in their teams to check different multiples. In order to win the sweets, the children try to speed up their algorithms. In general, they are too young to recognise that they do not need to check all rectangles (and that they can stop at the "square in the middle").

During this game there are a number of surprising things that can occur. The most interesting remark that we had was from an 8-year old who — on seeing that the number was prime — asked if he could have another sweet.

6 Searching and Sorting: Refinement

6.1 Searching and data refinement

In searching, we initially require only that a child can match a single piece of string with another piece of string in a collection. We demonstrate that we can hide a piece of string in a box, and place a number of pieces of string in a number of boxes (one per box). Finally, we hand them a piece of string and ask them to find the matching string in one of the boxes. However, they are told that they can open only one box at a time. Again, we play the children against each other, making alternate moves of a game. In this game, a move is looking in a box for the matching string. The first player to match the string wins the game. All other children act as spectators of each game; and observing the spectators is as insightful as observing the players.

We first observe the children selecting boxes in a random manner. The first interesting observation is when children realise that they have a better chance of winning if they never look in a box that they have already looked in. This observation usually arises from one of the child spectators shouting out that a player has already looked in a particular box and that they should choose another. In terms of software engineering, the children have quite naturally identified and communicated a process refinement At this point, we ask children to play against each other using the new, improved approach. However, we preclude the spectators from speaking during a game. Very quickly, it is observed that some of the children have problems remembering in which boxes they have already looked.

In the searching example, we have observed 3 types of data refinement which the children adopted as a specific way of overcoming the problem of having to remember which boxes had already been examined:

- Children searched the boxes in an ordered fashion (left to right, e.g.) and so had to remember only the last box searched.
- Children marked the boxes already searched (using a pencil, e.g.).
- Children moved the boxes examined into an *already examined pile.*

In the next phase, before we ask the children to play the game, we order the boxes based on the length of the strings within. Very quickly it is observed that not all the children realise that the strings in the boxes are ordered by length. The children who realise this are then observed playing in a more structured manner. Over a period of time, we observe that the children effectively refine their solution to a binary search where they do not always optimise the search by cutting the search space perfectly in two every time they make a guess. They know they need to look to the left or right of the current string box, based on the relative sizes of the search string and the string in the box.

The children are asked if it is possible to play *better*? Often they make quite solid arguments as to why their solution is optimal. At this stage, we play against them and we always find the string that is being looked for in the first guess. Without them knowing, we employ a perfect hashing function to map the string length directly to a particular box. They often accuse us of cheating; and only the more advanced students realise the trick.

6.2 Example: Sorting and process refinement

We propose the following algorithm for sorting a list of elements:

(i) Take the input list I and copy into list O.

(ii) If O is **ordered** then return this as output,
else swap two randomly chosen elements of O.

(iii) Goto step (ii)

We can specify this algorithm as a non-deterministic finite state automaton (NFSA). In the left hand side of figure 4, we illustrate sorting a list of three elements. The states are labelled with the list of integer elements. The transitions (all non-deterministic) are labelled by arcs and correspond to random swaps. Note: we have not represented the null transitions (where an element is swapped by itself) in the NFSA. The terminating state — where the list elements are ordered — is shaded.

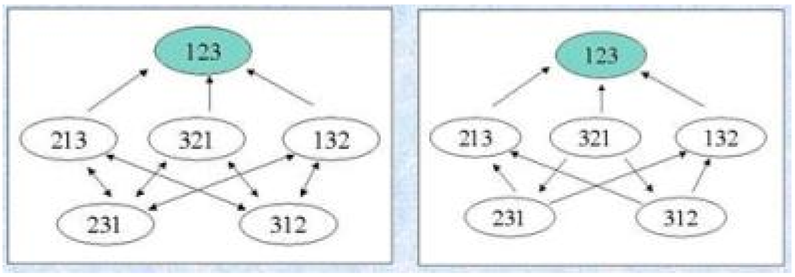

Fig. 4. *Left:* Random sorting of a 3-element list. *Right:* a sorting process refinement

We can see that from any starting state it is always possible that the terminating state will be reached, but that we may move in circles in an inefficient manner. Refinements that remove some of the non-determinism of the system (solution) can be used to generate a more efficient solution. Consider the first refinement, in the right of figure 4, where we have removed all swap transitions that exchange elements that are already in order. The removal of non-determinism, in this case, has transformed a *correct* solution into a better, more efficient, *correct* solution.

In order to see if the children can reason about refinement and correctness, we present the sorting problem using coloured balls (to be ordered as seen in the rainbow) hidden in shoe boxes. This is illustrated in figure 5. Through playing games, children naturally refine the inefficient solution and discover classic sorting algorithms in the process.

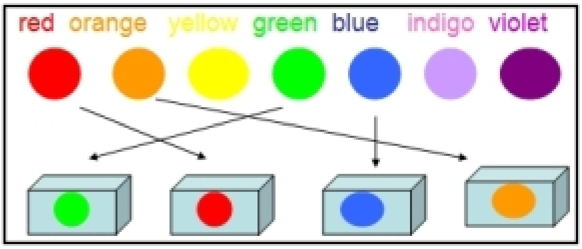

Fig. 5. Sorting with coloured balls

7 Some Interesting Observations

This paper is not reporting on a formal pedagogic experiment. We have made detailed notes concerning our sessions with the children and offer two interesting observations. With respect to algorithms, we were interested to note that, in general, a subset of students lose interest in a game as soon as the trick is explained by an algorithm. In early years (children as young as 7-years old) the percentage of children concerned is usually between 25 and 30. However, in later years (children in their late teens) this percentage usually grows to more than half the class. With respect to correctness, the yonger children are less interested in verifying their algorithms than the older children. In order to promote verification, we present the children with algorithms that are almost correct (but don't always work). Younger children appear to be disappointed in the fact that the algorithms can be broken; whilst the older (interested) children see that as being the real challenge.

8 Conclusions

We have demonstrated that it is possible to teach young children formal methods concepts through games and problem based learning. Children who are disinterested in mathematics regain an interest when they see that mathematical modelling can help them reason about algorithms and games. We do not make any claims about whether our sessions improve the childrens' mathematical ability. However, we do believe that this type of session is a good way of introducing computer science in schools. (We have also demonstrated that this problem-based learning approach can be used to teach university students[12] about computer science.)

As formal methods are foundational to computer science, it should not be any surprise that the rigorous, mathematical, analysis of algorithms and computations should be a major part of teaching computer science to children. This is not a new idea — it was discussed at the TFM workshop in Ghent in 2004[6], where daRosa presented research into the teaching of recursive algorithms as part of a high school mathematics course[5]. We already know that computer science education has need of mathematics; perhaps now the mathematics teachers can be persuaded to consider computer science (formal methods) as a good way of teaching mathematics.

References

[1] Bell, T., *A low-cost high-impact computer science show for family audiences*, 23rd Australasian Computer Science Conference **00** (2000), pp. 10–16.

[2] Brainerd., C., "Piaget's Theory of Intelligence," Prentice Hall, Englewood Cliffs, NJ, 1978.

[3] Bruner, J. S., "Toward a theory of instruction," Belknap Press of Harvard University, Cambridge, Mass,, 1966.

[4] Clements, D. H. and M. T. Battista, *Constructivist learning and teaching*, Arithmetic Teacher **38** (1982), pp. 34–35.

[5] da Rosa, S., *Designing algorithms in high school mathematics*, in: Dean and Boute [6], pp. 17–31.

[6] Dean, C. N. and R. T. Boute, editors, "Teaching Formal Methods, CoLogNET/FME Symposium, TFM 2004, Ghent, Belgium, November 18-19, 2004, Proceedings," Lecture Notes in Computer Science **3294**, Springer, 2004.

[7] Gardner, H., "Frames of mind: the theory of multiple intelligence," Basic Books, New York, 1983.

[8] Gibson, J. P., *A noughts and crosses java applet to teach programming to primary school children*, in: *PPPJ '03: Proceedings of the 2nd international conference on Principles and practice of programming in Java* (2003), pp. 85–88.

[9] Gibson, J. P. and J. O'Kelly, *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* (2005), pp. 87–97.

[10] Guilford., J. P., "The Nature of Human Intelligence," McGraw-Hill, New York, 1967.

[11] Hilgard, E. R. and G. H. Bower, "Theories of Learning," Prentice Hall, Englewood Cliffs, NJ, 1956.

[12] O'Kelly, J. and J. P. Gibson, *PBL: Year one analysis — interpretation and validation*, in: *PBL In Context — Bridging Work and Education*, 2005.

[13] O'Kelly, J. and J. P. Gibson, *Robocode and problem-based learning: a non-prescriptive approach to teaching programming*, in: R. Davoli, M. Goldweber and P. Salomoni, editors, *ITiCSE* (2006), pp. 217–221.

[14] Papert, S. and J. Sculley, "Mindstorms: children,computers, and powerful ideas," Basic Books, New York, 1980.

[15] Polya, G., "How to Solve It," Princeton University Press, 1971.

[16] Schoenfeld., A. H., "Mathematical Problem Solving," Academic Press, Orlando, Fla, 1985.