Teaching Them Early: Formal Methods in School

Faron Moller(✉), Liam O'Reilly, Stewart Powell, and Casey Denner

Swansea University, Swansea, UK
{F.G.Moller,L.P.OReilly}@swansea.ac.uk,
{Stewart.Powell,Casey.Denner}@technocamps.com

Abstract. In this paper, we describe a programme of school engagement aimed at instilling a discipline of computational thinking within pupils before they embark on a university course. The workshops we deliver are designed mainly to increase the pipeline of school leavers going on to study computer science or software engineering, specifically by changing perceptions on what this means amongst the vast majority – particularly girls – who think it is just a geeky topic for boys.

Over the past number of years, student enrollment has been increasing dramatically in our university's undergraduate computer science and software engineering degree programmes. Also, the performance of the students on first-year formal methods modules – which has historically been poor – has risen substantially. Whilst there are many influences contributing towards these trends, we present evidence that our efforts with school engagement has to a non-trivial extent contributed towards these: both through the way the undergraduate programme has been adapted to incorporate the Technocamps approach, and through providing a pipeline of students who understand the principles of computational thinking.

Keywords: Formal methods · School engagement · Computer science education · Pedagogy

1 Introduction

A typical 1st-year undergraduate student likes writing computer programs as this provides instant gratification: the computer does what you tell it to do. This is often why they choose to do computer science at university. As they proceed through their undergraduate education, they learn how to be more and more creative and to get the computer to do more and more exciting things.

For most of these students, however, stopping to think about whether the things that they make the computer do are in fact the *right* things to do – both technically as well as ethically – is often unattractive. Unwelcome digressions into mathematics are required to learn how to make your programs do what you want them to do – and to even formulate the specifications of what they

A. Cerone and M. Roggenbach (Eds.): FMFun 2019, CCIS 1301, pp. 173–190, 2021.
https://doi.org/10.1007/978-3-030-71374-4_9

should do. Unwelcome digressions into philosophy are required to understand the ethical implications of the programs that they write.

It is generally accepted that the typical modern computer science student is less mathematically minded than a generation ago, and the reasons for this are now understood. Moller and Crick [12] give a detailed account of the history of computing education in UK schools: from a strong position in the 1980s with the introduction of the BBC Micro into every school along with a curriculum for teaching the fundamentals of programming including hardware, software, Boolean logic and number representation; through the 1990s and beyond where the emergence of pre-installed office productivity software led to the computing curricula being permeated – and overwritten – by basic IT skills; "death-by-PowerPoint" became a common epithet for the subject. Beyond the arguments and references provided in [12], we can note a trend towards omitting mathematics as a prerequisite subject for studying computer science: of the 164 undergraduate computer science programmes offered by 105 universities in the UK, over 60% of these do not require mathematics as a school prerequisite [7].

There is a recognised digital skills shortage providing a high demand for computer science graduates [8], and an eagerness on the part of universities to fill places. However, with ever more students declaring on their applications that they are choosing to study computer science due to an affection for digital devices rather than an affection for the subject – and thus ever less prepared for the intellectual, logical and mathematical problem-solving challenges this entails – it can be a challenge in making some of the mathematical content (the formal methods) of the curriculum palatable. This is especially true in the current climate where student satisfaction is a key indicator which universities are required by law in the UK to publish in their recruitment and marketing.

Our thesis is simple: if we instil within pupils in schools the discipline of *computational thinking and problem solving* before and alongside their learning of how to write programs, we can deter them from forming a hacker's mentality of "program first, think later", and thus prevent habits forming which invite failures in software quality due, for example, to unintended consequences. Finding a means of doing this, however, is not straightforward; engaging the pupils in thought experiments before they get onto the computer requires an approach which is inspiring, creative and fun.

In this paper, we report on how we have addressed this issue in schools, and the impact that this has had on our university programme – both in the nature of the students entering the programme as well as on how we teach the syllabus. The structure of the paper is as follows. In Sect. 2 we reflect further on the background to the issues we address. In Sect. 3 we describe our programme of school and pupil engagement, in particular reflecting on our computational thinking and problem solving workshops. In Sect. 4 we describe how first year formal methods has changed in our university since we've started our school engagement activity. Finally, in Sect. 5 we provide some concluding remarks, and identify related activities.

2 Background

The nature of computer science education is changing, reflecting the increasing ubiquity and importance of its subject matter. In the last decades, computational methods and tools have revolutionised the sciences, engineering and technology. Computational concepts and techniques are starting to influence the way we think, reason and tackle problems; and computing systems have become an integral part of our professional, economic and social lives. The more we depend on these systems – particularly for safety-critical or economically-critical applications – the more we must ensure that they are safe, reliable and well designed, and the less forgiving we can be of failures, delays or inconveniences caused by the notorious "computer glitch."

Unlike for traditional engineering disciplines, the mathematical foundations underlying computer science are often not afforded the attention they deserve. The civil engineering student learns exactly how to define and analyse a mathematical model of the components of a bridge design so that it can be relied on not to fall down, and the aeronautical engineer learns exactly how to define and analyse a mathematical model of an aeroplane wing for the same purpose. However, software engineers are typically not as robustly drilled in the use of mathematical modelling tools. In the words of the eminent computer scientist Alan Kay [9], "most undergraduate degrees in computer science these days are basically Java vocational training." But computing systems can be at least as complex as bridges or aeroplanes, and a canon of mathematical methods for modelling computing systems is therefore very much needed. "Software's Chronic Crisis" was the title of a popular and widely-cited Scientific American article from 1994 [6] – with the dramatic term "software crisis" coined a quarter of a century earlier by Fritz Bauer [14] – and, unfortunately, its message remains valid a quarter of a century later.

University computer science departments face a sociological challenge posed by the fact that computers have become everyday, deceptively easy-to-use objects. Today's students – born directly into the heart of the computer era – have grown up with the Internet, a billion dollar computer games industry, and mobile phones with more computing power than the space shuttle. They often choose to study computer science on the basis of having a passion for using computing devices throughout their everyday lives, for everything from socialising with their friends to enjoying the latest films and music; and they often have less regard than they might to the considerations of what a university computer science programme entails, that it is far more than just *using* computers. In our experience, many of these students are easily turned off the subject when first faced with formal methods through a traditional course in discrete mathematics.

This has motivated us as a university department to reflect on our presentation of first-year formal methods, as well as explore means by which we can inform and educate pupils in schools as to the true nature of computer science before they become university students.

3 The Technocamps School Engagement Programme

Technocamps[1] is a pan-Wales schools outreach programme based at Swansea University but with hubs in the computer science department of every university across Wales. It was founded in 2003 to address the issues of computer science education in the context of the specific challenges posed in Wales. The portfolio of activities carried out by Technocamps is described and discussed in detail in [4], framed by the key educational challenges that exist in Wales, along with an evaluation of Technocamps interventions. In this paper, we will consider specifically the ways in which Technocamps workshops introduce and reinforce computational thinking and problem solving; how this has impacted on the uptake of computing; and how it has influenced the way in which the subject is delivered in our undergraduate programme.

Within classrooms throughout Wales, teachers are struggling to deliver the current computer science curriculum. This is unsurprising given that less than 40% of the teachers leading these classes have any training in ICT let along computer science [5]. The result is that pupils typically experience a lacklustre delivery focused on basic coding to solve the problem specified on the scheme of work in a very specific way according to the teacher's limited understanding, rather than exploring generic computational problem solving strategies to break down the problem and develop a solution.

As part of the varied offerings of the Technocamps programme, we have developed and delivered a series of *computational problem solving* workshops which explore the fundamentals of computational thinking – abstraction, algorithms, pattern recognition and decomposition – providing an accessible (if somewhat covert) introduction to formal methods. Suitable workshops are provided to the whole range of school classes from early primary through to late secondary. Our workshops for the youngest participants, which we deliver as part of our *Playground Computing* programme for primary schools, are mainly "unplugged" workshops – i.e., not involving a computer – typically carried out in the school gymnasium. Figure 1 depicts a scene from a Playground Computing workshop where the children are following instructions, whilst blindfolded, to solve tasks. By being blindfolded, they readily understand the need for absolute precision both in specifying solutions as well as in the instructions for carrying out these solutions.

Within these workshops – be they Playground Computing workshops for primary children or Technocamps workshops for late secondary students – pupils are challenged to approach problem solving in a way that is very different to what they have experienced. Rather than exploring problems through a series of steps which translate directly into lines of code, we use problems that are derived from puzzles and riddles, and have the pupils model the problems using state transition systems as a formalism. Again, these are very much unplugged exercises, though computer software is ultimately used to facilitate the modelling. We present here two classic riddles that feature in Technocamps Workshops.

[1] technocamps.com.

Fig. 1. A Technocamps *Playground Computing* Workshop in action

3.1 The Man-Wolf-Goat-Cabbage Riddle

The following riddle was posed by Alcuin of York in the 8th century, and more recently tackled by Homer Simpson in a 2009 episode of The Simpsons titled Gone Maggie Gone (which provides the ideal way to introduce the problem).

> *A man needs to cross a river with a wolf, a goat and a cabbage. His boat is only large enough to carry himself and one of his three possessions, so he must transport these items one at a time. However, if he leaves the wolf and the goat together unattended, then the wolf will eat the goat; similarly, if he leaves the goat and the cabbage together unattended, then the goat will eat the cabbage. How can the man get across safely with his three items?*

Pupils are challenged to first think logically about how to solve the problem, often through trial-and-error, which we enable through a collection of supportive tools[2] – developed in Scratch[3] – which allow pupils to explore the puzzles in an interactive way. A more systematic approach is then presented by suggesting to pupils that they should "model" the scenario by abstracting away the non-important information and presenting the problem as a sequence of states. Pupils are encourage to think about what constitutes a state, and what actions might occur that would result in a transition from one state to another.

Figure 2 gives a taste of how this is presented to the class. Having introduced the problem, it is represented by a picture which captures the essential information (which side of the river each of the four entities lies). The participants are then encouraged to consider what actions may occur, and how these actions would change the picture – that is, how the state of the world would change. This introduces and reinforces the notions of abstraction – identifying the relevant information and disregarding anything irrelevant – and decomposition –

[2] bit.ly/Technothink.
[3] https://scratch.mit.edu.

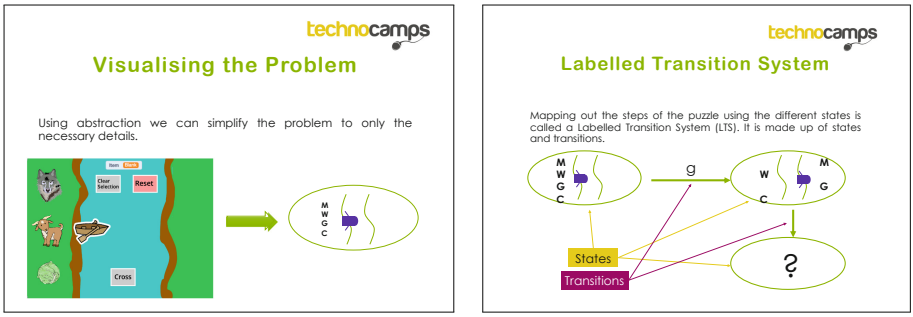


Fig. 2. Introducing modelling using transition systems

breaking down a problem and solving it by solving smaller problems. Getting the participants to depict the occurrence of actions by arrows between states, they are naturally introduced to the notion of a labelled transition system (LTS). Through exploring transitions – by hand and using simulation tools – the participants are asked to find a sequence of actions which will solve the problem. Figure 3 shows this problem being solved in a workshop.

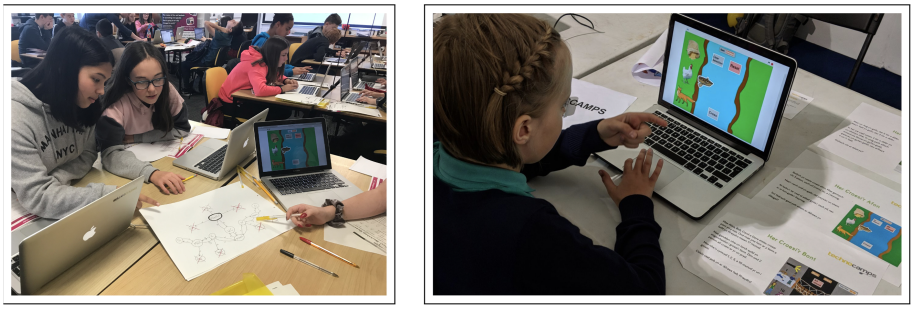


Fig. 3. LTS modelling in the classroom

The labelled transition system which the students are developing is depicted in Fig. 4. A state of the LTS represent the current position (left or right bank) of the four entities (man, wolf, goat, cabbage); and there are four actions representing the four possible actions that the man can take:

- m = the man crosses the river on his own;
- w = the man crosses the river with the wolf;
- g = the man crosses the river with the goat; and
- c = the man crosses the river with the cabbage.

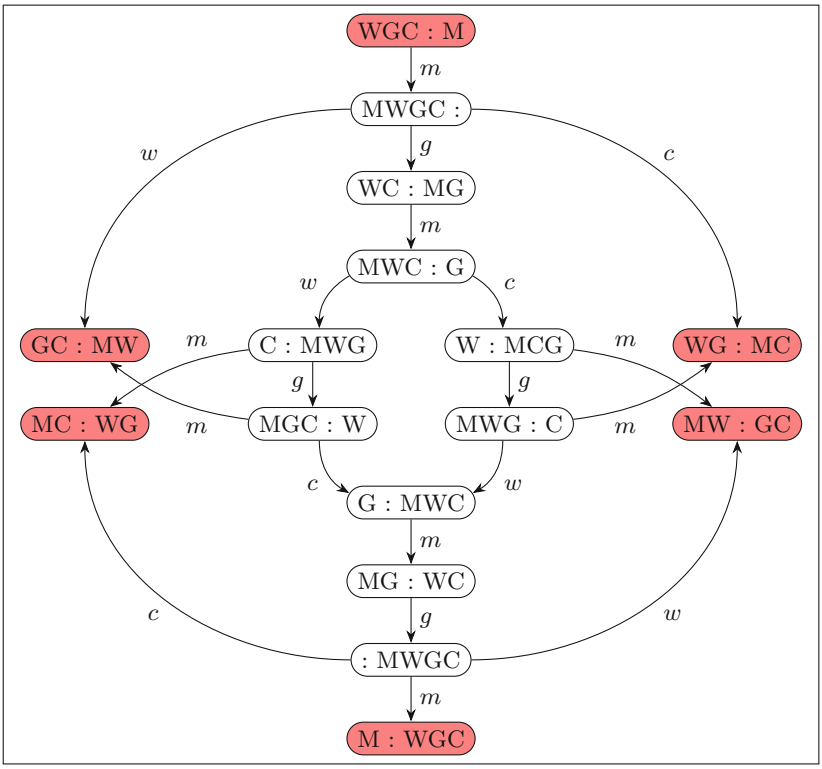


Fig. 4. The Man-Wolf-Goat-Cabbage LTS.

The initial state is (MWGC:) (meaning all are on the left bank of the river), and the goal is to find a sequence of actions which will lead to the state (:MWGC) (meaning all are on the right bank of the river). However, we want to avoid going through any of the six dangerous (red) states:

(WGC:M) (GC:MW) (WG:MC) (MC:WG) (MW:GC) (M:WGC)

There are several possibilities (all involving at least 7 crossings), for example:

$$g,\ m,\ w,\ g,\ c,\ m,\ g.$$

3.2 The Water Jugs Riddle

In the 1995 film Die Hard: With a Vengeance, New York detective John McClane (played by Bruce Willis) and Harlem dry cleaner Zeus Carver (played by Samuel L. Jackson) had to solve the following problem in order to prevent a bomb from exploding at a public fountain. (Again, this provides the ideal means to introduce the problem to a class.)

Fig. 5. Solving the water jug puzzle in Computational Thinking workshops

> *Given only a five-gallon jug and a three-gallon jug, neither with any markings on them, fill the larger jug with* exactly *four gallons of water from the fountain, and place it onto a scale in order to stop the bomb's timer and prevent disaster.*

This riddle – and many others like it – was posed by Abbot Albert in the 13th Century, and can be solved using an LTS. A state of the system underlying this riddle consists of a pair of integers (i, j) with $0 \leq i \leq 5$ and $0 \leq j \leq 3$, representing the volume of water in the 5-gallon and 3-gallon jugs A and B, respectively. The initial state is $(0, 0)$ and the final state you wish to reach is $(4, 0)$.

There are six moves possible from a given state (i, j):

- $(0, j) \xrightarrow{fillA} (5, j)$
- $(i, 0) \xrightarrow{fillB} (i, 3)$
- $(i, j) \xrightarrow{emptyA} (0, j)$ if $i > 0$
- $(i, j) \xrightarrow{emptyB} (i, 0)$ if $j > 0$
- $(i, j) \xrightarrow{AtoB} \big(\max(0, i + j - 3), \min(3, i + j)\big)$ if $i > 0$ and $j < 3$
- $(i, j) \xrightarrow{BtoA} \big(\min(5, i + j), \max(0, i + j - 5)\big)$ if $i < 5$ and $j > 0$

Drawing out the LTS (admittedly a daunting task in this instance yet a useful exercise), we get the following 7-step solution:

$$(0,0) \xrightarrow{fillA} (5,0) \xrightarrow{AtoB} (2,3) \xrightarrow{emptyB} (2,0) \xrightarrow{AtoB} (0,2) \xrightarrow{fillA} (5,2) \xrightarrow{AtoB} (4,3) \xrightarrow{emptyB} (4,0).$$

In Fig. 5 we can see a school workshop in action. We use blue sand rather than water in these workshops to avoid the obvious risk of creating a wet chaos. Through experimenting, the participants inevitably stumble upon a solution; but charged with the task of explaining their solution step-by-step, they naturally arrive at a solution which they describe using the language and notation of labelled transition systems. Arriving at a complete solution does not require the

class to find and express the most general rules as presented above. However, for older groups, finding these rules provides an interesting challenge in numeracy.

These types of riddles and puzzles allow pupils to easily grasp and understand the powerful concept of labelled transition systems. After seeing only a few examples, they are able to model straightforward systems by themselves using LTSs. Once an intuitive understanding has been established, the task of understanding the mathematics behind LTSs becomes less foreboding.

3.3 Feedback from the Workshops

Technocamps has been successfully delivering these workshops to school groups since 2003, on university campuses and in schools as well as elsewhere in the community. In particular, in the *Learning in Digital Wales* project for Welsh Government's Department of Education, Technocamps delivered an average of 9.8 h of interactive workshops across every secondary school throughout Wales over an 18-month period during 2014–2016. For the purposes of this paper, we reflect on a recent programme of engagement.

During the Summer term of 2019, Technocamps delivered its computational problem solving workshops to 424 pupils, aged between 12–15, within the South Wales region as part of a series of STEM Enrichment Programmes. Of those who answered the feedback questionnaire, feedback was significantly positive with over 86% of pupils rating the workshop overall as Great/Good as well as its subject content.

The Technocamps goal of changing perceptions about computer science as a subject worth studying is reflected in its activity. Since 2011, 50,000 young people – over 7% of the Welsh population who are today aged 5–24 – have participated in Technocamps Workshops; a full 43% of these have been girls, and these girls are 25% more likely than the boys to return for follow-on workshops.

4 First-Year Formal Methods

We have replicated the Technocamps approach to introducing formal methods for our first-year university computer science students. Our efforts in this direction have been nothing short of remarkable. By adopting and adapting our approach over the past twenty years from a traditional starting point, we have substantially increased the success rate – and substantially decreased the failure rate – of our students. Figure 6 shows how the percentage of students attaining a 1st-class grade (a grade over 70%) rose from 2% in 2000–2001 to over 60% in 2017–2018 and 2018–2019, whilst those failing the course (by attaining a grade under 40%) dropped over the same time frame from 56% to under 2%. The figure also shows the class sizes which have more than tripled over the most recent five years which explains a noticeable dip in attainment which, as we explain below, was remedied by further tweaking of our delivery model. The fact that this success is based on our approach is borne out by reflecting on annual student feedback for the various modules which students take across their programme of study; our

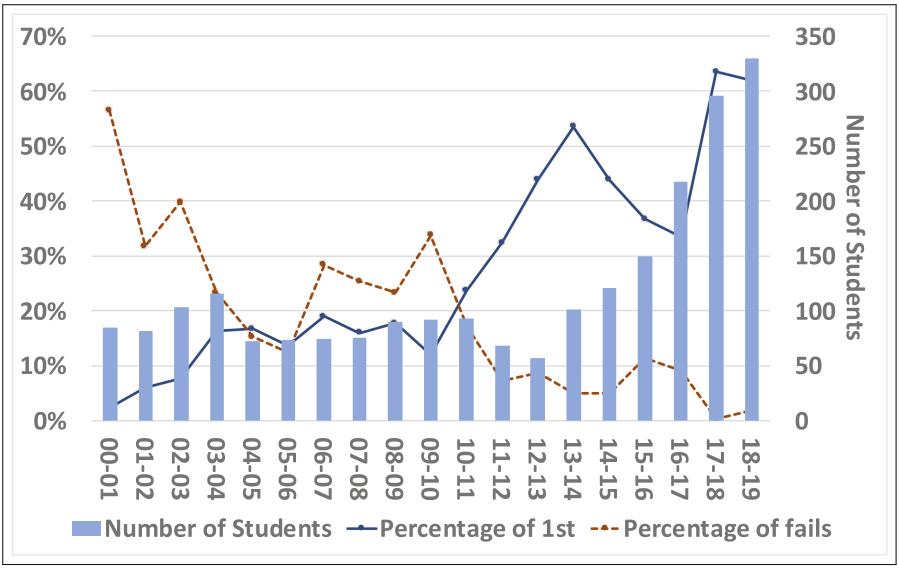


Fig. 6. Trends of students achieving 1st-class and failing results; and class sizes.

delivery model is contrasted favourably against traditional approaches used in other modules taken by the same students, and recorded attendance (and hence engagement) is highest in this module.

Through years of reflecting on how to successfully present formal methods to beginning computer science students, we have identified the following key considerations, all of which we have gleaned – and from which we have learned – from student feedback.

- *Do not call it (discrete) mathematics.* A simple change of name from *"discrete mathematics for computer science"* to *"modelling computing systems"* in 2010–2011 was enough for us to witness a substantially increased level of engagement and attainment with the course, as made evident in Fig. 6. There was no other change that year to add to the cause of this effect.
- *Do not formalise early on.* The standard approach to, e.g., propositional logic is to present the formal syntax and semantics of the logic and emphasise the precise form and function of the connectives. The approach we have adopted is to stress the careful use of English, and to introduce logical symbols as mere shorthand for writing out English sentences. Formalism becomes far easier to adapt to if and once the students are comfortable with working with the concepts.
- *Exploit riddles and games.* As described above, riddles and games provide an effective way to instil the rigours of computational thinking. These were incorporated more and more from 2010 onwards, resulting in the year-on-year improvement in attainment reflected in Fig. 6.

- *Use regular interactive small-group problem sessions.* We supplement three hours of weekly whole-class lectures with a one-hour small-group problem session (of 30–50 students) in which the emphasis is on the students carrying out computational problem-solving tasks, typically in pairs. We are confident in our thesis that this matters, as tweaking the sizes and regularity of these groups through the years coincides with peaks and dips in the attainment graphs. In particular, see the next consideration.
- *Keep these problem session groups small.* As can be seen in Fig. 6, attainment dropped between 2014 and 2017 as class sizes grew, but more than recovered in 2017–2018 despite a huge increase in the overall class size. This was due to an increase in the number of problem session groups; whilst the whole-class lectures became far less personable due to the huge numbers, the decrease in the sizes of the problem session groups resulted in much better results. Again, this being the only substantive change to delivery, we are confident in attributing the positive effect to this.

It is worth stressing that throughout the years, entrance requirements have not changed to admit only stronger applicants. On the contrary, pressures to increase student numbers (i.e., fees income) have meant that academically-weaker students (those with lower school grades) are being admitted in greater numbers. Also, neither the content of the course nor the way it is assessed has gotten easier. Again, quite to the contrary, the topics covered in the first-year formal methods modules have expanded to include the coinductive concept of bisimulation equivalence, a topic which even postgraduate research students find challenging, but which we successfully present as outlined in the next section.

4.1 Verification via Games

Having introduced a formalism for representing and simulating (the behaviour of) a system, the next question to explore is: *Is the system correct?* In its most basic form, this amounts to determining if the system matches its specification, where we assume that both the system and its specification are given as states of some LTS. For example, consider the two vending machines V_1 and V_2 depicted in Fig. 7, where V_1 is taken to represent the specification of the vending machine while V_2 is taken to represent its implementation. Clearly the behaviour of V_1 is somehow different from the behaviour of V_2: after *twice* inserting a 10p coin into V_1, we are *guaranteed* to be *able* to press the coffee button; this is *not* true of V_2. The question is: *How do we formally distinguish between processes?*

4.2 The Formal Definition of Equivalence

A traditional approach to this question relies on determining if these two states are related by a *bisimulation relation*, which is a binary relation R over its states in which whenever $(x, y) \in R$:

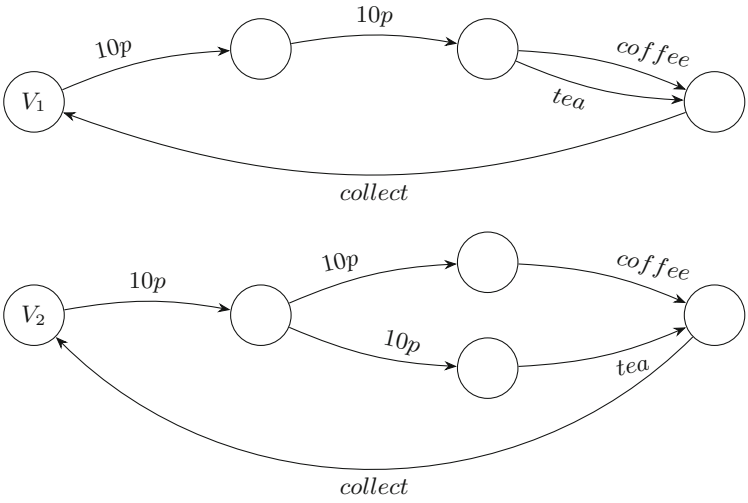


Fig. 7. Two Vending Machines.

- if $x > ax'$ for some x' and a, then $y > ay'$ for some y' such that $(x', y') \in R$;
- if $y > ay'$ for some y' and a, then $x > ax'$ for some x' such that $(x', y') \in R$.

Simple inductive definitions already represent a major challenge for undergraduate university students; so it is no surprise that this coinductive definition of a bisimulation relation is incomprehensible even to some of the brightest postgraduate students – at least on their first encounter with it. It thus may seem incredulous to consider this to be a first-year discrete mathematics topic, even if it is a perfect application for exploring equivalence relations as taught earlier in the course. However, there is a straightforward way to explain the idea of bisimulation equivalence to first-year students – a way which they can readily grasp and are happy to explore and, indeed, play with. The approach is based on the following game.

4.3 The Copy-Cat Game

This game is played between two players, typically referred to as Alice and Bob. We start by placing tokens on two states of an LTS, and then proceed as follows.

1. Alice moves *either* of the two tokens forward along an arrow to another state; if this is impossible (that is, if there are no arrows leading out of either node on which the tokens sit), then Bob is declared to be the winner.
2. Bob must move the *other* token forward along an arrow which has *the same label* as the arrow used by Alice; if this is impossible, then the Alice is declared to be the winner.

This exchange of moves is repeated for as long as neither player gets stuck. If Bob ever gets stuck, then Alice is declared to be the winner; otherwise Bob is declared to be the winner (in particular, if the game goes on forever).

Alice, therefore, wants to show that the two states holding tokens are somehow different, in that there is something that can happen from one of the two states which cannot happen from the other. Bob, on the other hand, wants to show that the two states are the same: that whatever might happen from one of the two states can be copied by the other state.

It is easy to argue that two states should be considered equivalent exactly when Bob has a winning strategy in this game starting with the tokens on the two states in question; and indeed this is taken to be the definition of when two states are equal, specifically, when an implementation matches its specification.

As an example, consider playing the game on the LTS depicted in Fig. 8.

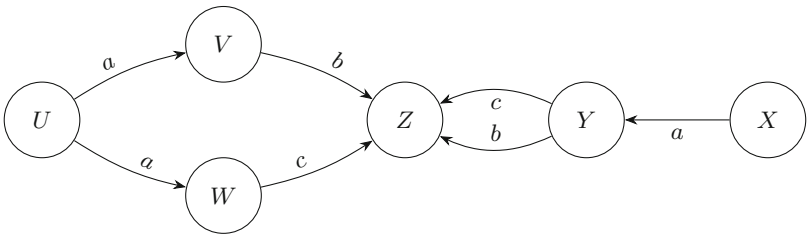


Fig. 8. A simple LTS.

Starting with tokens on states U and X, Alice has a winning strategy:

- Alice can move the token on U along the a-transition to V.
- Bob must respond by moving the token on X along the a-transition to Y.
- Alice can then move the token on Y along the c-transition to Z.
- Bob will be stuck, as there is no matching c-transition from V.

This example is a simplified version of the vending machine example; and a straightforward adaptation of the winning strategy for Alice will work in the game starting with the tokens on the vending machine states V_1 and V_2. We thus have an argument as to why the two vending machines are different.

4.4 Relating Winning Strategies to Equivalence

Whilst this notion of equality between states is particularly simple, and even entertaining to explore, it coincides precisely with the complicated coinductive definition of when two states are bisimulation equivalent. Seeing this is the case is almost equally straightforward.

- Suppose we play the copy-cat game starting with the tokens on two states x and y which are related by some bisimulation relation R. It is easy to see that Bob has a winning strategy: whatever move Alice makes, by the definition of a bisimulation relation, Bob will be able to copy this move in such a way that

the two tokens will end up on states x' and y' which are again related by R; and Bob can keep repeating this for as long as the game lasts, meaning that he wins the game.

- Suppose now that R is the set of pairs of states of an LTS from which Bob has a winning strategy in the copy-cat game. It is easy to see that this is a bisimulation relation: suppose that $(x, y) \in R$:
  - if $x > ax'$ for some x' and a, then taking this to be a move by Alice in the copy-cat game, we let $y > ay'$ be a response by Bob using his winning strategy; this would mean that Bob still has a winning strategy from the resulting pair of states, that is $(x', y') \in R$;
  - if $y > ay'$ for some y' and a, then taking this to be a move by Alice in the copy-cat game, we let $x > ax'$ be a response by Bob using his winning strategy; this would mean that Bob still has a winning strategy from the resulting pair of states, that is $(x', y') \in R$.

We have thus taken a concept which baffles postgraduate research students, and presented it in a way which is well within the grasp of first-year undergraduate students.

4.5 Determining Who Has the Winning Strategy

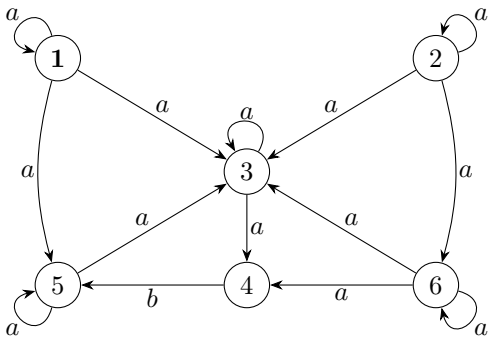
Once the notion of equivalence is understood in terms of winning strategies in the copy-cat game, the question then arises as to how to determine if two particular states are equivalent, i.e., if Bob has a winning strategy starting with the tokens on the two given states. This isn't generally a simple prospect; games like chess and go are notoriously difficult to play perfectly, as you can only look ahead a few moves before getting caught up in the vast number of positions into which the game may evolve.

Here again, though, we have a straightforward way to determine when two states are equivalent. Suppose we could paint the states of an LTS in such a way that any two states which are equivalent – that is, from which Bob has a winning strategy – are painted the same colour. The following property would then hold.

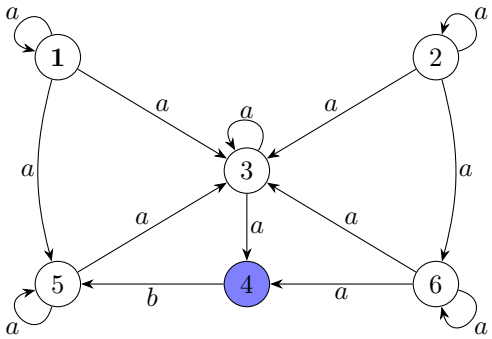
> *If any state with some colour C has a transition leading out of it into a state with some colour C', then* every *state with colour C has an identically-labelled transition leading out of it into a state coloured C'.*

That is, if two tokens are on like-coloured states (meaning that Bob has a winning strategy) then no matter what move Alice makes, Bob can respond in such a way as to keep the tokens on like-coloured states (ie, a position from which he still has a winning strategy). We refer to such a special colouring of the states as a *game colouring*.

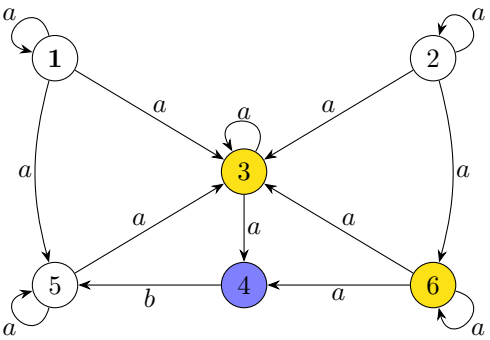
To demonstrate, consider the following LTS.



At the moment, all states are coloured white, and we might consider whether this is a valid game colouring. It becomes readily apparent that it is not, as the white state 4 can make a b-transition to the white state 5 whereas none of the other white states (1, 2, 3, 5 and 6) can do likewise. In fact, in any game colouring, the state 4 must have a different colour from 1, 2, 3, 5 and 6. Hence we paint it a different colour from white; say blue:



We again consider whether this is now a valid game colouring. Again it becomes apparent that it is not, as the white states 3 and 6 have a-transitions to a blue state, whereas none of the other white states 1, 2 and 5 do. And in any game colouring, the states 3 and 6 must have a different colour from 1, 2 and 5. Hence we paint these a different colour from white and blue; say yellow:



We again consider whether this is now a valid game colouring. This time we find that it is, as every state can do exactly the same thing as every other state of the same colour:

- every white state has an a-transition to a white state and an a-transition to a yellow state;
- every yellow state has an a-transition to a yellow state and an a-transition to a blue state;
- every blue state has a b-transition to a white state.

At this point we have a complete understanding of the game, and can say with certainty which states are equivalent to each other. This is an exercise which first-year students can happily carry out on arbitrarily-complicated LTSs, which again gives testament to the effectiveness of using games to great success in imparting difficult theoretical concepts to first-year students – in this case the concept of partition refinement.

While students take turns playing each other in the copy-cat game, they develop an intuitive understanding of winning strategies: that the first player must play correctly, and the second player – no matter how well they play – can never win. They even have fun doing it! This allows them to argue when two systems are different (or the same) and even paves the way for other more advanced formal verification techniques such as observational equivalence.

5 Conclusions

As with any topic, teaching formal methods – even to school children – is most successful when done in a way which nurtures their willingness to engage. Appealing to their existing understand of how the world works, using puzzles as a medium, students can quickly become comfortable using mathematical concepts such as labelled transition systems. A similar lesson is learnt when it comes to teaching verification: starting with the formal definition of bisimulation (or similar) is an uphill battle from the start, even for postgraduate research students. However, starting from games like the copy-cat game, such topics become immediately accessible.

We have used this approach for over a decade to teach discrete mathematics incorporating the modelling and verification of computing systems as part of our first-year undergraduate programme. With the fine-tuning of our approach, and abiding by the considerations outlined in Sect. 4, we have succeeded in maximising attainment levels of the students through active and interested engagement.

Of course, problem solving through recreational mathematics – which is ultimately what we are exploiting in our approach – has very many proponents, and there is a long and extensive history of books marketed towards the mathematically-inquisitive. We are by no means alone in recognising the power of applying recreational mathematics to the development of computational problem solving skills; as relevant exemplars we note Averbach and Chein's *Problem Solving Through Recreational Mathematics* [1], Backhouse's *Algorithmic Problem Solving* [2], Levitin and Levitin's *Algorithmic Puzzles* [10]; and Michalewicz and Michalewicz's *Puzzle-Based Learning* [11]. What we propose in particular is an embedding of the approach from before a student's undergraduate journey, in particular to engage them in a topic – discrete mathematics – that they typically struggle with, both academically and in terms of recognising its relevance in the subject. In this sense, we are closely related to the various approaches that have been developed of late for introducing school-aged audiences to computational thinking. In this vein we note the CS Unplugged[4] and the CS4FN[5] initiatives.

The "informal" way in which we approach the teaching of formal methods has many parallels with Morgan's *(In)Formal Methods: The Lost Art* [13]. The course described in this report is for upper-level computer science students who are already adept at writing programs who are studying software development methods. Nonetheless, many of its findings – in particular as reflected in the student feedback – are replicated in our own activity, where positive feedback is provided on: the interactive and hands-on approach; the amusing exercises and assignments; the class room style teaching; the overall teaching methodology with dedicated tutors; and the means by which the relevance of the course is stressed.

As a final note, many of the considerations that we have identified as being important in teaching mathematics to computing students are reflected by Betteridge et al. [3] as being useful and thus adopted in their novel approach to teaching computing to mathematics students.

References

1. Averbach, B., Chein, O.: Problem Solving Through Recreational Mathematics. Dover, Mineola (1980)
2. Backhouse, R.: Algorithmic Problem Solving. Wiley, New York (2011)
3. Betteridge, J., et al.: Teaching of computing to mathematics students. In: Proceedings of the 3rd Conference on Computing Education Practice, CEP 2019, Durham, UK, 9 Jan 2019, pp. 12:1–12:4 (2019)

[4] csunplugged.org.
[5] cs4fn.org.

4. Crick, T., Moller, F.: Technocamps: advancing computer science education in wales. In: Proceedings of WiPSCE: The 10th Workshop in Primary and Secondary Computing Education, pp. 121–126. ACM (2015)
5. Education Workforce Council (EWC): Annual statistics digest (2019). https://www.ewc.wales/site/index.php/en/documents/research-and-statistics/annual-statistics-digest/archived-annual-statistics-digests/1895-2017.html
6. Gibbs, W.W.: Software's chronic crisis. Sci. Am. **271**(3), 86–95 (2004)
7. Higher Education Statistics Agency (HESA): Recruitment data for computer science courses in the UK (2019). https://www.hesa.ac.uk
8. House of Commons Science and Technology Committee: Digital skills crisis: Second Report of Session 2016–2017 (2016)
9. Kay, A.: A conversation with Alan Kay. ACM Queue **2**(9), 20–30 (2004)
10. Levitin, A., Levitin, M.: Algorithmic Puzzles. Oxford University Press, New York (2011)
11. Michalewicz, Z., Michalewicz, M.: Puzzle-Based Learning. Hybrid Publishers, Melbourne (2010)
12. Moller, F., Crick, T.: A university-based model for supporting computer science curriculum reform. J. Comput. Educ. **5**(4), 415–434 (2018)
13. Morgan, C.: (In-)Formal methods: the lost art. In: Liu, Z., Zhang, Z. (eds.) SETSS 2014. LNCS, vol. 9506, pp. 1–79. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-29628-9_1
14. Naur, P., Randell, B. (eds.): Software Engineering: Report of a Conference Sponsored by the NATO Science Committee, Garmisch, Germany, 7–11 Oct 1968. NATO Scientific Affairs Division (1969)