Formal Methods for First Years

Faron Moller Liam O'Reilly

Department of Computer Science
Swansea University

{F.G.Moller|L.P.OReilly}@swansea.ac.uk

13 April 2013

Abstract

In this report, we describe the underlying concept and contents of a new textbook *"Modelling Computing Systems: The Mathematics of Computer Science."* This book will be published by Springer in Autumn 2013, and is aimed at first-year university computer science students as a novel approach to introducing them in an engaging way to formal methods at the very start of their education. In fact, the approach to formal modelling based on labelled transition systems promoted by the book has been successfully adapted to workshops delivered by Technocamps, a schools outreach programme aimed at secondary school pupils.

1 Why a New Book on Formal Methods

Computer Science is a relatively young discipline. University Computer Science Departments are rarely more than a few decades old. They will typically have emerged either from a Mathematics Department or an Engineering Department, and until recently a Computer Science degree was predominantly about writing computer programs (the mathematical software) and building computers (the engineering hardware). Textbooks typically referred to programming as an "art" or a "craft" with little scientific basis compared to traditional engineering subjects, and many computer programmers still like to see themselves as part of a pop culture of geeks and hackers rather than as academically-trained professionals.

However, the nature of Computer Science is changing rapidly, 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 traditional engineering disciplines which are solidly rooted on centuries-old mathematical theories, the mathematical foundations underlying Computer Science are younger, and Computer Scientists have yet to agree on how best to approach the fundamental concepts and tasks in the design of computing systems. The Civil Engineer knows 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 knows exactly how to define and analyse a mathematical model of an aeroplane wing for the same purpose. However, Software Engineers have few universally-accepted mathematical modelling tools at their disposal. In the words of the eminent Computer Scientist Alan Kay, "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, and, unfortunately, its message remains valid two decades later.

University Computer Science Departments face a sociological challenge posed by the fact that computers have become everyday, deceptively easy-to-use objects. A single generation ago, new Computer Science students typically had teenage backgrounds spent writing Basic and/or Assembly Language programs for their early hobbyist computers. A passion for this activity is what drove these students into University Computer Science programmes, and they were not disappointed with the education they received. Their modern-day successors on the other hand – 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 downloading the latest films, 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.

There is a universal trend of large numbers of first-year students transferring out of Computer Science programmes and into related programmes such as Media Studies or Information Studies. This trend, we feel, is often unjustified, and can be reversed by a more considered approach to modelling and the mathematical foundations of system design, one which the students can connect and feel at home with right from the beginning of their University education. This was the motivation behind producing a modern textbook, to be published by Springer in Autumn 2013, aimed at teaching first-year undergraduate students the essential mathematics and modelling techniques for computing systems in a novel and relatively light-weight way.

The book is divided into two parts. Part I, subtitled *Mathematics for Computer Science*, introduces concepts from Discrete Mathematics which are in the curriculum of any University Computer Science programme, as well as much which often is not. This material is typically taught in service modules by mathematicians, and new Computer Science students often find it difficult to connect with the material presented in a purely mathematical context. In this book, this material is presented in an engaging and motivating fashion as the basis of computational thinking.

Part II of the book, subtitled *Modelling Computing Systems*, develops a particular approach to modelling based on state transition systems. Such transition systems have always featured in the Computer Science curriculum, but traditionally (and increasingly historically) only within the study of formal languages. Here they are introduced as general modelling devices, and languages and techniques are explored for expressing and reasoning about system specifications and (concurrent) implementations. Although Part I covers twice as many pages as Part II, much of the Mathematics presented in Part I itself is used directly for modelling systems, and forms the basis on which the approach developed in Part II is based.

The main benefit of mathematical formalisation is that systems can be modelled and analysed in precise and unambiguous ways; but formal precision can also be a major pitfall in modelling since it can compromise simplicity and intuition. In this book, therefore, the starting point is intuition and examples, and precise concepts are developed from that basis. How and when to be precise is certainly not less important to learn than precision itself: the ability to give mathematical proofs often does not depend on knowing precise formal definitions and foundations. One can, for example, write down recursive functions without having a precise formal concept in mind.

There is a long standing tradition in disciplines like Physics to teach modelling through little artifacts. The fundamental ideas of computational modelling and thinking as well can better be learned from idealised examples and exercises than from many real world computer applications. This book builds on a large collection of logical puzzles and mathematical games that require no prior knowledge about computers and computing systems; these can be much more fun and sometimes much more challenging than analysing a device driver or a criminal record database. Also, computational modelling and thinking is about much more than just computers!

In fact, games play a far more important role in the book: they provide a novel approach to understanding computer software and systems which is proving to be very successful both in theory and practice. When a computer runs a program, for example, it is in a sense playing a game against the user who is providing the input to the program. The program represents a strategy which the computer is using in this game, and the computer wins the game if it correctly computes the result. In this game, the user is the adversary of the computer and is naturally trying to confound the computer, which itself is attempting to defend its claim that it is computing correctly, that is, that the program it is running is a winning strategy. (In Software Engineering, this game appears in the guise of *testing*.) Similarly, the controller of a software system that interacts with its environment plays a game against the environment: the controller tries to maintain the system's correctness properties, while the environment tries to confound them.

This view suggests an approach to addressing three basic problems in the design of computing systems:

1. ***Specification*** refers to the problem of precisely identifying the task to be solved, as well as what exactly constitutes a solution. This problem corresponds to the problem of defining a winning strategy.

2. ***Implementation*** or ***Synthesis*** refers to the problem of devising a solution to the task which respects the specification. This problem corresponds to the problem of implementing a win-

ning strategy.

3. ***Verification*** refers to the problem of demonstrating that the devised solution does indeed respect the specification. This problem corresponds to the problem of proving that a given strategy is in fact a winning strategy.

This analogy between the fundamental concepts in Software Engineering on the one hand, and games and strategies on the other, provides a mode of computational thinking which comes naturally to the human mind, and can be readily exploited to explain and understand Software Engineering concepts and their applications. It also motivates the thesis that Game Theory provides a paradigm for understanding the nature of computation.

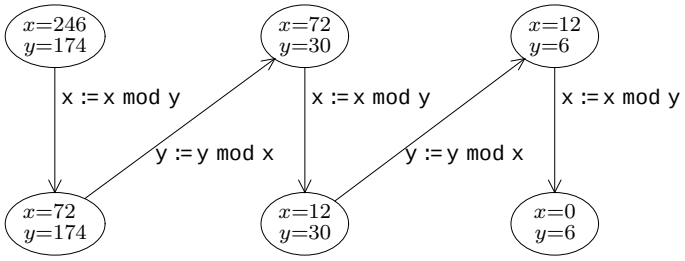
There are over 200 exercises presented throughout the book, all of which have complete solutions at the back of the book; as well as over 200 further exercises at the ends of the chapters whose solutions are not provided. The exercises within the chapters are often used to explore subtleties or side-issues, or simply to put lengthy arguments into an appendix. The material in this book has been used successfully for over a decade in first-year Discrete Mathematics and Systems Modelling modules. Countless eyes have passed over the text, and a thousand students have solved its exercises.

2 Labelled Transition Systems for Problem Solving

Consider the following presentation of Euclid's algorithm for computing the greatest common divisor of two numbers `x` and `y`:

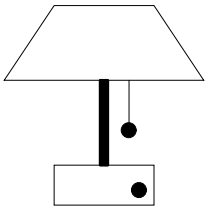
```
forever do
    x := x mod y;
    if x=0 then return y;
    y := y mod x;
    if y=0 then return x
od
```

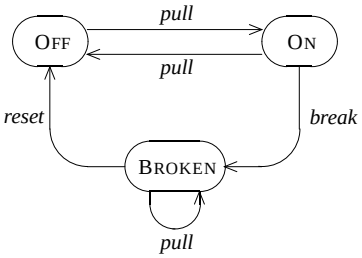
To understand this program, you can hand-turn it, keeping track of the state of the variables:



In general, a computation – or more generally a process – can be represented by a Labelled Transition System (LTS), which consists of a directed graph, where the vertices represent states, and the edges represent transitions from state to state, and are labelled by events. As shown above, an LTS is typically presented pictorially, with the states represented by circles and the transitions by arrows between states labelled by actions.

As a further example, consider the following lamp process:





The lamp has a string to pull for turning the light on and off, and a reset button which resets the circuit if a built-in circuit breaker breaks when the light is on.

At any moment in time the lamp can be in one of three states:

- OFF – in which the light is off (and the circuit breaker is set);
- ON – in which the light is on (and the circuit breaker is set); and
- BROKEN – in which the circuit breaker is broken (and the light is off).

In any state the string can be pulled, causing a transition into the appropriate new state (from the state BROKEN, the new state is the same state BROKEN). In the state ON, the circuit breaker may break, causing a transition into the state BROKEN in which the reset button has popped out; from this state, the reset button may be pushed, causing a transition into the state OFF.

These simple examples demonstrate the simple, but effective, use of LTSs as a means of modelling computing problems and real world objects. Of course, LTSs are not limited to such primitive forms. They can be extended in a variety of ways to add further information, for example, notions of time and space can be represented within states so that real-time and hybrid systems can be described. In this respect, LTSs can be regarded as a general formalism for modelling any kind of system, be it a computing system, a real world object, or a concurrent real-time system with multiple components.

Introducing LTSs with Puzzles

Whilst the definition of a labelled transition system is surprisingly straightforward for such a powerful formalism, getting students to engage with it requires some ingenuity. Fortunately, this is equally straightforward by resorting to well-known recreational puzzles.

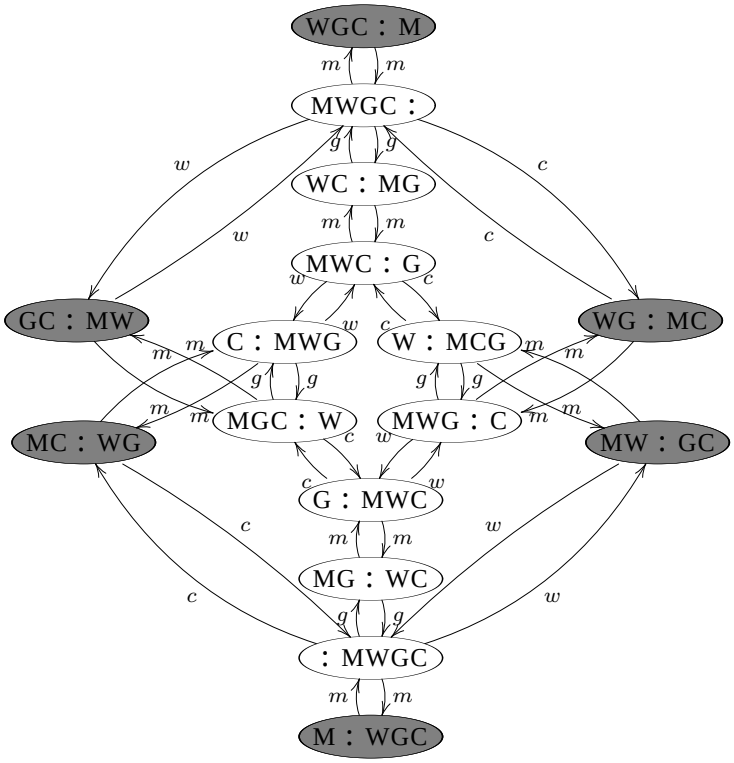
The Man-Wolf-Goat-Cabbage Riddle

> *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?*

This 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.

This puzzle can be solved by modelling it as an LTS. A state of the LTS will represent the current position (left or right bank) of the four entities (man, wolf, goat, cabbage); and there will be 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.



The initial state is $\boxed{\text{MWGC :}}$ (meaning all are on the left bank of the river).

We wish to find a sequence of actions which will lead to the state $\boxed{\text{: MWGC}}$ (meaning all are on the right bank of the river).

However, we want to avoid going through any of the six dangerous states:

$$\boxed{\text{WGC : M}} \quad \boxed{\text{GC : MW}} \quad \boxed{\text{WG : MC}}$$
$$\boxed{\text{MC : WG}} \quad \boxed{\text{MW : GC}} \quad \boxed{\text{M : WGC}}$$

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

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

The Missionaries and Cannibals Riddle

> *Three missionaries are travelling with three cannibals when they come upon a river. They have a boat, but it can only hold two people. The river is filled with piranha, so they all must eventually cross in the boat; no one can cross the river by swimming. The problem is: should the cannibals ever outnumber the missionaries on either side of the river, the outnumbered missionaries would be in deep trouble. Each missionary and each cannibal can row the boat. How can all six get across the river safely?*

Similarly to the Man-Wolf-Goat-Cabbage riddle, this puzzle can also be solved using an LTS, as depicted in Figure 1. Each state of the LTS records the positions of the people (which banks they are on) and which side holds the boat. The groups on the two banks are depicted side-by-side divided by wiggly lines representing the river, with the group holding the boat enclosed in parentheses. We only consider the safe states where the cannibals do not outnumber the missionaries.

There are five possible actions:

- m (a missionary crosses alone);
- mm (two missionaries cross together);
- c (a cannibal crosses alone);
- cc (two cannibals cross together); and
- mc (a missionary and a cannibal cross together).

Notice that all of the transitions are drawn bi-directionally, as every transition can clearly be reversed.

The group start in the top-left state in which the whole group is on the left bank, and they wish to get to the bottom-right state in which they are all on the right bank. It is not hard to find a such path through the LTS which involves 11 crossings.

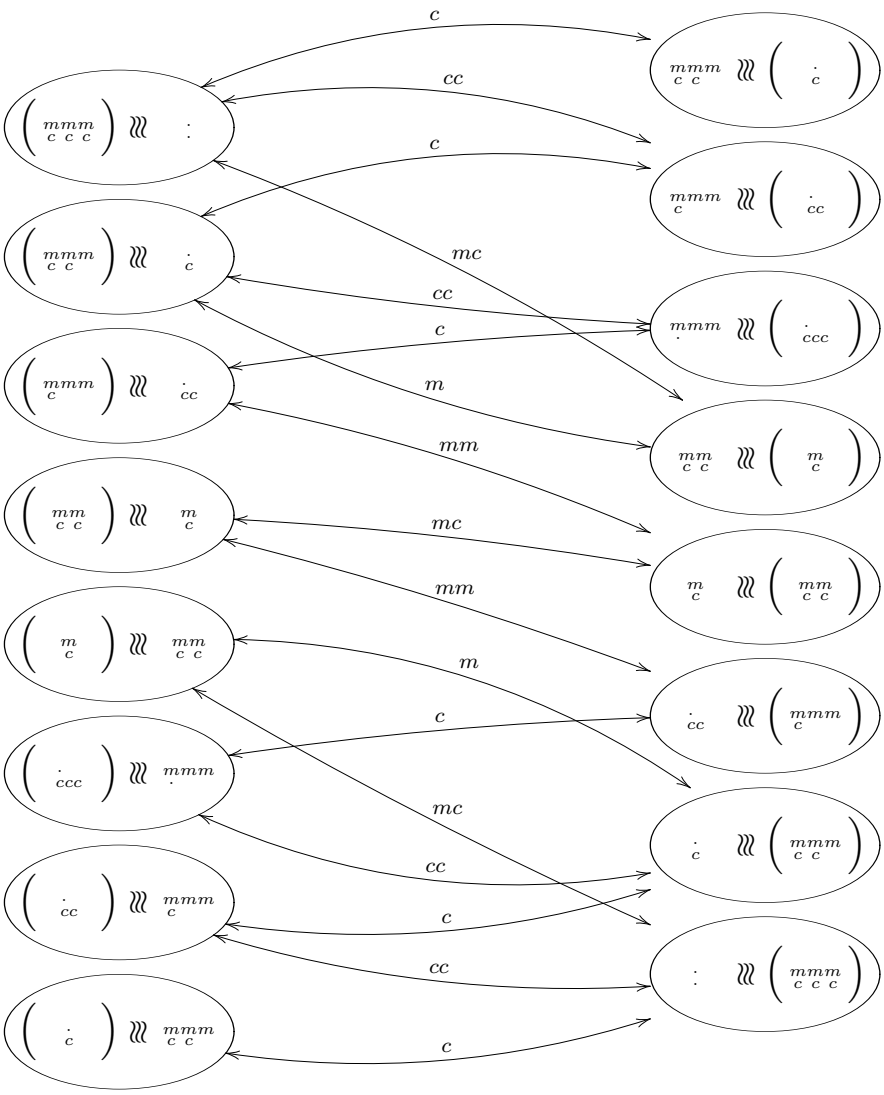


Figure 1: The Missionaries and Cannibals riddle as an LTS.

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.

> *Given only a five-gallon jug and a three-gallon jug, neither with any markings on them, they had to 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.*

How did they manage this feat?

This riddle was posed by Abbot Albert in the 13th Century.

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):

$$
\begin{array}{lll}
\text{(i,j)} \xrightarrow{\textit{fillA}} & (5, j) & \left(\textit{if } i{=}0\right) \\
\text{(i,j)} \xrightarrow{\textit{fillB}} & (i, 3) & \left(\textit{if } j{=}0\right) \\
\text{(i,j)} \xrightarrow{\textit{emptyA}} & (0, j) & \left(\textit{if } i{>}0\right) \\
\text{(i,j)} \xrightarrow{\textit{emptyB}} & (i, 0) & \left(\textit{if } j{>}0\right) \\
\text{(i,j)} \xrightarrow{\textit{AtoB}} & \left(\max(0, i{+}j{-}3), \min(3, i{+}j)\right) & \left(\textit{if } i{>}0 \textit{ and } j{<}3\right) \\
\text{(i,j)} \xrightarrow{\textit{BtoA}} & \left(\min(5, i{+}j), \max(0, i{+}j{-}5)\right) & \left(\textit{if } i{<}5 \textit{ and } j{>}0\right)
\end{array}
$$

Drawing out the LTS, we get the following 7-step solution:

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

These simple riddles and puzzles allow students 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 Bisimulation for Dummies

Beyond having a formalism for representing and simulating (the behaviour of) a system, we want to be able to determine if the system is 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 models of a vending machine V_1 and V_2 depicted in Figure 2, 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?*

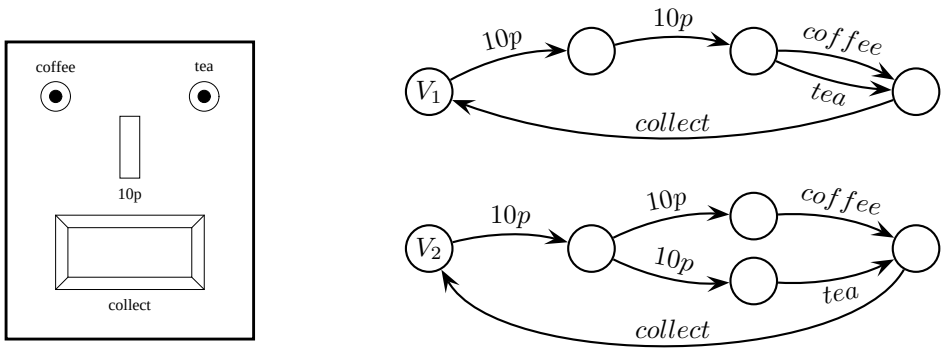


Figure 2: Two Vending Machine models

The formal definition of bisimilarity

A traditional approach to this question relies on determining if these two states are related by a *bisimulation relation* R as defined as follows.

> *A binary relation R over states of an LTS is a bisimulation relation if, and only if, whenever $(x, y) \in R$:*
>
> - *if $x \xrightarrow{a} x'$ for some x' and a, then $y \xrightarrow{a} y'$ for some y' such that $(x', y') \in R$; and*
> - *if $y \xrightarrow{a} y'$ for some y' and a, then $x \xrightarrow{a} x'$ 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. However, there is a straightforward way to explain the idea of bisimilarity 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.

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. The first player (Alice) chooses one of the two tokens, and moves it 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 the second player (Bob) is declared to be the winner.

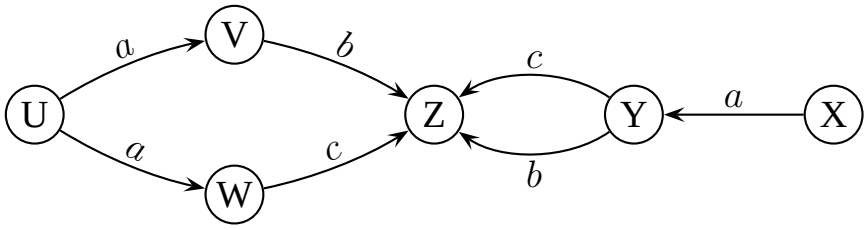
2. The second player (Bob) must move the *other* token forward along an arrow which has *the same label* as the arrow used by the first player; if this is impossible, then the first player (Alice) is declared to be the winner.

This exchange of moves is repeated for as long as neither player gets stuck. Note that Alice gets to choose which token to move *every time it is her turn*; she does not have to keep moving the same token. If Bob ever gets stuck – ie, cannot copy a move made by Alice – 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 the Copy-Cat 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 following LTS.



Starting with tokens on states U and X, the ***first*** player (Alice) has a winning strategy:

- Alice can move the token on U along the a-transition to V.
- Bob must match this 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 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 states V_1 and V_2. We thus have an argument as to why the two vending machines are different.

Relating winning strategies to bisimilarity

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 bisimilar. Furthermore, seeing this is the case is almost equally straightforward.

- Suppose we play the Copy-Cat Game starting with the tokens on two states E and F 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 E' and F' 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 $x \xrightarrow{a} x'$ for some x' and a, then taking this to be a move by Alice in the Copy-Cat Game, we let $y \xrightarrow{a} y'$ 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 $y \xrightarrow{a} y'$ for some y' and a, then taking this to be a move by Alice in the Copy-Cat Game, we let $x \xrightarrow{a} x'$ 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.

Determining who has the winning strategy

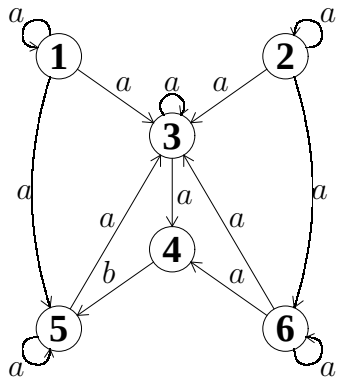
Once the notion of equivalence is understood in terms of winning strategies in the Copy-Cat Game, the question then arises as to how to determine if two particular states are equivalent, ie, if Bob has a winning strategy starting with the tokens on the two given states. This isn't generally a simple 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.

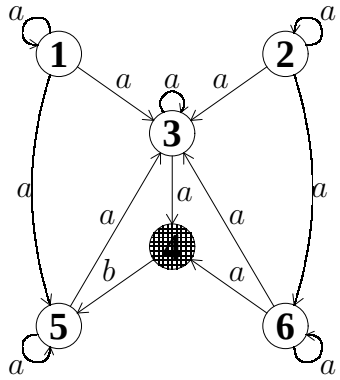
> *If any state with some colour C has a transition leading out of it into a state with some colour C', then* every *state with colour C has an identically-labelled transition leading out of it into a state coloured C'.*

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 a *game colouring*.

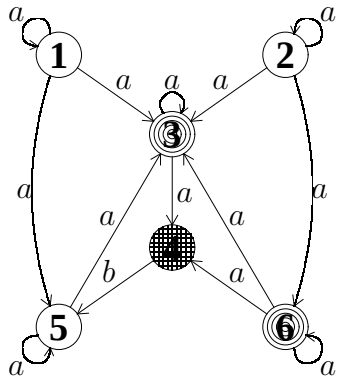
To demonstrate, consider the following LTS.



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; in order to present this example in black-and-white, we shall paint the state 4 with the colour "checkered."



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 checkered 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 checkered; we shall choose the colour "swirly."



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 swirly state;
- every swirly state has an a-transition to a swirly state and an a-transition to a checkered state;
- every checkered 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.

4 Conclusion

Students can quickly and easily understand the modelling of computing systems if it is done in a suitable way. Starting with some formal semantics and real world examples, in our experience, makes the task very daunting, difficult and generally unpleasant for students. However, appealing to their existing understanding of how the world works, using puzzles as a medium, students can quickly become comfortable using mathematical concepts such as LTSs. 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, whereas starting from games like the Copy-Cat Game, life is much easier for everyone.

We have used these approaches for over a decade to successfully teach the modelling and verification of computing systems to first-year students of our undergraduate course. This has eventually lead to the production of our new modern textbook, to be published by Springer in Autumn 2013, aimed at teaching first-year undergraduate students the essential mathematics and modelling techniques for computing systems in a novel and relatively light-weight way.

Appendix

Modelling Computing Systems: The Mathematics of Computer Science

Table of Contents