Antonio Cerone
Markus Roggenbach (Eds.)

Communications in Computer and Information Science 1301

Formal Methods – Fun for Everybody

First International Workshop, FMFun 2019
Bergen, Norway, December 2–3, 2019
Revised Selected Papers

Springer

Communications in Computer and Information Science 1301

Antonio Cerone · Markus Roggenbach (Eds.)

Formal Methods – Fun for Everybody

First International Workshop, FMFun 2019
Bergen, Norway, December 2–3, 2019
Revised Selected Papers

Springer

Editors
Antonio Cerone
Nazarbayev University
Nur-Sultan, Kazakhstan

Markus Roggenbach
Swansea University
Swansea, UK

ISSN 1865-0929 ISSN 1865-0937 (electronic)
Communications in Computer and Information Science
ISBN 978-3-030-71373-7 ISBN 978-3-030-71374-4 (eBook)
https://doi.org/10.1007/978-3-030-71374-4

Preface

The largest transformations that universities make to industrial practices is through releasing legions of graduates every year. These graduates challenge established processes and pave ways for new approaches. The standard computer science or software engineering graduate leaves university with either no knowledge of Formal Methods or a hatred for Formal Methods. Unless this situation is changed, Formal Methods will never be accepted in industry.

The First International Workshop "Formal Methods – Fun for Everybody" (FMFun 2019) explored ways to utilize this pathway to transformation for spreading Formal Methods. In current practice, Formal Methods is often taught by theoreticians, who (ab) use their Formal Methods courses to teach theoretical concepts rather than putting Formal Methods in a software engineering context. The vision of this workshop series is that Formal Methods ought to be taught in such a way that every student can have fun with it.

The 2019 two-day workshop included participants from Formal Methods as well as from Education who exchanged their views and perspectives. The innovative format of the workshop consisted of two keynote talks by Peter Csaba Ölveczky and Magne Haveraaen, 11 contributed presentations and a number of open discussion sessions. These sessions were characterised by an open atmosphere in which participants listened to each other's views, provided feedback and were inspired to develop ideas further. The workshop also featured a living lab on teaching Formal Methods with Fun, which was split in two parts: on the first day participants made available and presented their teaching materials and discussed them in small groups; on the second day the living lab session merged with the open discussion sessions and contributed to the formulation of the white paper published in these proceedings. This joint paper, which is co-authored by most of the workshop participants and some of the workshop authors who could not participate in the physical event, collects the outcomes of discussion and activities at the workshop, further revised via email in an intense collaborative effort that extended over several months. This paper provides examples of good practice in Formal Methods teaching as well as general recommendations on curriculum development which we intend to circulate to appropriate educational bodies.

The workshop received 15 full paper submissions and two presentation paper submissions. Each full paper submission was reviewed for quality, correctness, originality and relevance by at least three Program Committee members. A final discussion among the Program Committee members was carried out using EasyChair. Ten full paper contributions and one presentation paper contribution were accepted for presentation at the workshop. This volume contains the white paper, two papers by the workshop keynote speakers and revised versions of the nine full paper contributions that were accepted for publication. The published contributed papers were further reviewed after the workshop.

We would like to thank all the Program Committee members for their valuable and timely efforts. We are also grateful to the General Chair, Volker Stolz, and the Workshops Chairs, Violet Ka I Pun and Martin Leucker. Finally, we would like to thank all the workshop attendees for their active participation in discussions and for the feedback they provided to the authors.

October 2020

Antonio Cerone
Markus Roggenbach

Organization

Program Committee

Luis Barbosa	University of Minho, Portugal
Hubert Baumeister	Technical University of Denmark, Denmark
Antonio Cerone	Nazarbayev University, Kazakhstan
Ming Chai	Beijing Jiaotong University, China
Tom Crick	Swansea University, UK
Hans de Nivelle	Nazarbayev University, Kazakhstan
Elsa Estevez	Universidad Nacional del Sur, Argentina
Sabine Glesner	TU Berlin, Germany
Jan Friso Groote	Eindhoven University of Technology, The Netherlands
Stefan Gruner	University of Pretoria, South Africa
Klaus Havelund	Jet Propulsion Laboratory, USA
Magne Haveraaen	University of Bergen, Norway
Paddy Krishnan	Oracle, Australia
Karl Lermer	Zurich University of Applied Sciences, Switzerland
Carlos Gustavo Lopez Pombo	Universidad de Buenos Aires and CONICET, Argentina
Bas Luttik	Eindhoven University of Technology, The Netherlands
Kathy Malone	Nazarbayev University, Kazakhstan
Faron Moller	Swansea University, UK
Lucia Rapanotti	The Open University, UK
Steve Reeves	University of Waikato, New Zealand
Markus Roggenbach	Swansea University, UK
Kristin Yvonne Rozier	Iowa State University, USA
Holger Schlingloff	Fraunhofer FOKUS and Humboldt University, Germany
Gerardo Schneider	Chalmers — University of Gothenburg, Sweden
Siraj A. Shaikh	Coventry University, UK
Benjamin Tyler	Nazarbayev University, Kazakhstan
Janis Voigtländer	University of Duisburg-Essen, Germany
Ayman Wahba	Ain Shams University, Egypt
Peter Ölveczky	University of Oslo, Norway

Additional Reviewers

Arcuschin Moreno, Iván
Martinez Suñé, Agustín Eloy

Contents

Rooting Formal Methods Within Higher Education Curricula for Computer Science and Software Engineering — A White Paper —

Antonio Cerone[1](✉), Markus Roggenbach[2], James Davenport[3], Casey Denner[2], Marie Farrell[4], Magne Haveraaen[5], Faron Moller[2], Philipp Körner[6], Sebastian Krings[7], Peter Csaba Ölveczky[8], Bernd-Holger Schlingloff[9], Nikolay Shilov[10], and Rustam Zhumagambetov[1]

[1] Nazarbayev University, Nur-Sultan, Kazakhstan
antonio.cerone@nu.edu.kz
[2] Swansea University, Swansea, UK
m.roggenbach@swansea.ac.uk
[3] University of Bath, Bath, UK
[4] University of Manchester, Manchester, UK
[5] University of Bergen, Bergen, Norway
[6] Heinrich-Heine-Universität, Düsseldorf, Germany
[7] Niederrhein University of Applied Sciences, Krefeld, Germany
[8] University of Oslo, Oslo, Norway
[9] Humboldt-Universität zu Berlin, Berlin, Germany
[10] Innopolis University, Kazan, Russia

Abstract. This white paper argues that formal methods need to be better rooted in higher education curricula for computer science and software engineering programmes of study. To this end, it advocates

– improved teaching of formal methods;
– systematic highlighting of formal methods within existing, 'classical' computer science courses; and
– the inclusion of a compulsory formal methods course in computer science and software engineering curricula.

These recommendations are based on the observations that

– formal methods are an essential and cost-effective means to increase software quality; however
– computer science and software engineering programmes typically fail to provide adequate training in formal methods; and thus
– there is a lack of computer science graduates who are qualified to apply formal methods in industry.

This white paper is the result of a collective effort by authors and participants of the *1st International Workshop on Formal Methods – Fun for Everybody* which was held in Bergen, Norway, 2–3 December 2019.

https://doi.org/10.1007/978-3-030-71374-4_1

As such, it represents insights based on learning and teaching computer science and software engineering (with or without formal methods) at various universities across Europe.

1 Introduction

The greatest contribution that universities make to industrial practices is through releasing legions of graduates every year. When properly equipped with a scholarly education, these graduates challenge established processes and pave the way for new approaches. In the increasingly-digital world we live in, the scope for this is arguably greatest in the software industry, particularly given that the public perception – and indeed the reality – is that software is inherently unreliable.

Advances in digital technology take place at an astronomical rate, unfettered by regulations which would hinder progress in other scientific endeavours. There are generally few established principles in place to ensure that new software systems are as reliable as, say, a new vaccine. Software engineers demonstrate success in their company by releasing systems which, for *almost* all intents and purposes, *appear* to work. Because of the benefits these advances offer society, the public are generally accepting of – and, indeed, used to – software failures.

This situation persists in spite of the fact that computer science and software engineering research has developed a multitude of design principles which could help to improve software quality [Bar11]. It has been over half a century since Robert Floyd's seminal paper [Flo67] set out the means by which computer programs could be analysed to determine their functional correctness, and formal methods for developing correct software have been steadily devised and refined ever since. The typical computer science or software engineering graduate, however, leaves university with little or no knowledge of formal methods, and even a dislike for whatever formal methods they have encountered in their studies. Thus, rather than opening doors for formal methods in (software) industry, university education seems to have a detrimental effect.

Due to their ubiquity, software failures are overlooked by society as they tend to result in nothing more serious than delays and frustrations. We accept as mere inconvenience when a software failure results in a delayed train or an out-of-order cash machine or a need to repeatedly enter details into a website. However, the problems of systems failures become more serious (costly, deadly, invasive) as automatic control systems find their way into virtually every aspect of our daily lives. This increasing reliance on computer systems makes it essential to develop and maintain software in which the possibility, and probability, of hazardous errors is minimised. Formal methods offer cost-efficient means to achieve the required high degree of software quality.

A major reason that students (and, in turn, software engineers) have a negative attitude towards formal methods is that these are not introduced with due care during the early stages of higher education. Left to the theoretical computer science professor, such courses often start with fearful terms like state machine, logical inference, mathematical semantics, etc., without providing elementary explanations of the basic notions which relate these to the practice of software development. In their defence, formal methods professors often find it difficult to deliver the subject due to students' scepticism [Zhu20], which arises from the generally limited or non-existent exposure to formal methods in the rest of the curriculum. Boute [Bou09] and Sekerinski [Sek06] observe that limited references from other subjects and isolated use are the main factors leading to students' low opinion. Even worse, students perceive formal methods to be unsuitable for actual software engineering [BDK+06] or even an "additional burden" [BLA+09].

In this white paper we analyse what hinders a successful formal methods education, and make constructive suggestions about how to change the situation. We are convinced that such changes are a prerequisite for formal methods to become widely accepted in industry. We analyse the current situation of formal methods teaching and explore ways which we think will be engaging for students and practitioners alike. Our vision is that formal methods can be taught in such a way that both students and lecturers will enjoy formal methods teaching.

This white paper is the result of a collective effort by authors and participants at the 1st International Workshop "Formal Methods – Fun for Everybody", which was held in Bergen, Norway, 2–3 December 2019. At the workshop, there were several discussion sessions. Based on these, the two lead authors devised a paper outline, which was subsequently "populated" with text snippets written by all authors. The resulting draft was carefully edited, and agreed upon by all authors. By its very nature, this white paper offers a spectrum of opinions, in particular in the personal statements. What unites us are the following beliefs:

- Current software engineering practices fail to deliver dependable software.
- Formal methods are capable of improving this situation, and are beneficial and cost-effective for mainstream software development.
- Education in formal methods is key to progress things.
- Education in formal methods needs to be transformed.

In Sect. 2, we analyse the challenges in teaching formal methods. In Sect. 3, we collect ideas about how to teach formal methods – the fun way. In Sect. 4, we discuss how to increase the visibility of formal methods throughout the curriculum. In Sect. 5, we suggest a syllabus for a compulsory formal methods course. Finally, we discuss how to assess such teaching efforts in Sect. 6, before making concluding remarks in Sect. 7.

2 Challenges in Teaching Formal Methods

Teaching of formal methods faces a number of challenges. Currently, as a knowledge area, formal methods are virtually absent from curricula in computer science or software engineering. Formal Methods barely appear in the ACM/IEEE 2014 Software Engineering Curriculum, and indeed the development of formal specifications is explicitly deemed to be inappropriate for a capstone project [ACM15, p. 56]. Moreover, many students have an incorrect perception of what formal methods are about. Formal methods neither make the headlines nor are a popular topic in social networks, nor are they visibly used by industry. It is also the case that colleagues as well as students have misguided ideas concerning the mathematical background required to utilise formal methods. In the following, we elaborate on these topics. The section concludes with personal statements.

We begin our discussion by providing a working definition, cf. [RCS+21], of what a formal method might be.

Definition 1. *A formal method M can be seen to consist of the three elements syntax, semantics, and method:*

- *Syntax: the precise description of the form of objects (strings or graphs) belonging to M.*
- *Semantics: the 'meaning' of the syntactic objects of M, in general by a mapping into some mathematical structure.*
- *Method: algorithmic ways of transforming syntactic objects, in order to gain some insight about them.*

A typical example of a formal method is the process algebra CSP: its syntax is given in form of a grammar; there are various formal semantics (operational, denotational, and axiomatic ones); and there are proof methods for refinement via model checking and theorem proving.

Applying this definition, e.g., to the programming language Pascal, we see that it also qualifies as a formal method. It has a defined syntax and formal semantics; and each compiler and static analyser provides a method, the Hoare calculus would be another instance of a method.

UML on the other hand does not qualify as a formal method. The syntax is largely fixed via meta models, and there are various methods available, e.g., for code generation (e.g., from class diagrams or state machines). However, proposed semantics for UML contain several critical "variation points" and has – to the best of our knowledge – never been fully formalised.

2.1 On the Absence of Formal Methods from Computer Science and Software Engineering Curricula

Anecdotal evidence suggests that current computer science and software engineering curricula rarely cover formal methods to a large extent. We exemplify this observation by providing an historic perspective on programming education, an element central to all curricula.

In the late 1980s, Pascal was a dominant teaching language for beginning programming students. Pascal is a small, structured programming language with a syntax designed to be easy to parse [ISO90]. Most textbooks of the time presented the Pascal language using syntax diagrams, alerting the students to the idea of context free grammars, e.g., [CC82]. The element of syntax was taught as an integral part of programming. Some textbooks included the entire ISO Pascal standard, thus making the students aware of language definition documents.[1] For those specifically interested, Pascal had a widely available formal semantics [HW73]. Robust programming, i.e., checking preconditions, was an essential part of programming courses. Some universities would even have space for a formal methods course, typically based on Hoare logic, in their undergraduate curriculum: i.e., a formal method was taught.

About 20 years ago, Pascal was superseded by Java as the dominating teaching language. Java is a much more complex language than Pascal; it supports object-oriented development, and it has large support libraries. Thus, in the transition to Java, precise syntax and semantics was replaced by a more example-driven approach, e.g., [DD07], where the first half contains similar material to [CC82]. Verification tools such as Java Pathfinder[2] rarely made it into the syllabus of a programming course. Instead, students needed to learn more methodology, such as object-orientation, test-driven design and agile methods. All of this reduces the students' exposure to formality, such as formal syntax or precise semantics[3], making the gap to formal methods larger. Further, the pragmatics of software development take up more of the curriculum, leaving less space for a formal methods course in the core curriculum. Dewar and Schonberg support this critical assessment: "It is our view that Computer Science education is neglecting basic skills, in particular in the areas of programming and formal methods. We consider that the general adoption of Java as a first programming language is in part responsible for this decline [DS18]."

In recent years, Python has emerged into a common teaching language for programming. The move towards Python represents a change back to a much smaller language than Java. The Python reference document is just 160 pages, and its formal grammar is only four pages [vRtPdt20]. This should make it possible to at least expose the students to formal syntax and a standardisation document. However, the typing and semantic model of Python remains complex, and is not easily formalised.

Thus, while current programming education based on Java often fails to provide foundations for formal methods by discussing syntax and semantics, the move towards Python provides the silver lining that the element of syntax might again become a part of standard education in programming.

[1] Pattis [Pat94] even suggested teaching Extended Backus-Naur Form (EBNF) as the first topic in computer science. Not to facilitate presenting the syntax of a programming language, but because EBNF is a microcosm of programming. With no prerequisites, students are introduced to a variety of fundamental concepts in programming: formal systems, abstraction, control structures, equivalence of descriptions, the difference between syntax and semantics, and the relative power of recursion versus iteration.

[2] https://github.com/javapathfinder/jpf-core/wiki.

[3] The recent *The Java®Language Specification, Java SE 14 Edition* is 800 pages [GJS+20] and not easily digestible.

2.2 Students' Perception of Formal Methods

The reduced exposure to formal approaches, as described in Sect. 2.1, supports university students' misconception that formal methods are a difficult topic with little or no practical relevance. This keeps students away from formal methods during their undergraduate studies. Even worse, it leads them to embrace the common belief that mathematics and computer science are two independent, fully distinct disciplines. Computer science is rather identified with programming, which, in turn, is seen more like an art rather than a scientific activity [CL20]. Interestingly, this view has even been supported not only by the pragmatic evolution of programming languages outlined in the previous paragraphs, but also by some academic publications claiming that rigorous mathematical knowledge is not necessary for computer science practitioners [Gla00]. Finally, this view has been paradoxically encouraged by the introduction of computer science in high schools. In fact, although in several schools computer science has been introduced as a stand-alone subject, it is not connected with mathematics but, instead, it is presented as a 'service subject' intrinsically tied to the use of computers. Scope of the subject is to provide tools that facilitate students in carrying out their homework and class projects [Cer20,Gib08].

Although we can say that, on average, a typical computer science student tends to have a negative perception of formal methods, in reality lecturers observe a lot of variation between students, as well as changes of perceptions in one direction or the other. Variations in students can be observed starting from the first programming courses. A slightly exaggerated categorisation goes as follows. On the one hand, there are students who tackle programming in a purely 'artistic way' by sitting down at the computer and writing code immediately, using debugging rather than problem solving to reach the solution. On the other hand, there are students who start analysing the problem using pen and paper, then draw diagrams, possibly write pseudo-code, test their solution on paper and, only when they are confident in their solution, they sit in front of a computer and convert their solution into a program. Obviously, it is the latter approach what lectures suggest. Normally, the former group of students tend to have a negative perception of formal methods, whereas the latter group tend to have a positive one. This partition of the students in two groups appears more evident once recursion is introduced in the programming course. The former group of students will tend to hate recursion, the latter group will tend to love it.

These two opposite perceptions obviously occur in several degrees. Moreover, they are not static but, at least potentially, dynamic and may be either encouraged or hindered in various ways throughout the course of undergraduate studies. The common absence of formal semantics among the topics of programming courses definitely keeps students away from an early exposure to formal methods and prevents them from really understanding what formal methods are. Being exposed to some basic operational semantics could actually help students to better understand conditional and iterative constructs, which are normally serious challenges for first year students. Furthermore, recursion could be better understood, thus providing the basis for a future interest in formal methods.

Concerning senior students, although for some of them their perception of formal methods may have been strongly oriented towards the negative side, there is hope to shift them towards the positive side. Senior students tend to be very pragmatic and their minds are dominated by the goal of entering the job market and the industrial world. Therefore they will build a positive perception of formal methods when presented with their pragmatic and industry-oriented aspects.

2.3 Limited Visibility of Formal Methods in Media and Industry

How students perceive a knowledge area has many drivers, such as personal success, like/dislike of certain academic teachers, their grades, etc. But maybe ‘coolness’ is the dominant factor. During their studies, students want to do something cool, maybe work with AlphaZero[4] or participate in a hackathon such as Google’s Hash Code. Students also strive to get ‘cool jobs’, e.g., with Google, Facebook, Amazon, and the like. Currently, what one might want to call the ‘coolness factor’ of formal methods is rather low. Formal methods make neither the headlines nor are prominent in social media, nor are they visibly used by industry.

Besides studying, quite a number of students work on the side for companies. In these jobs, students often see only small parts of the overall job profile of a professional computer scientist or software engineer. Many of these side jobs deal with having a quick and dirty solution for some pressing problem, adapting software according to customer requests, or building prototypes in order to find out whether some concept works out. In contrast, mature students, coming back from industry and getting into university education again, know about the importance of quality assurance. But as they usually were not exposed to formal methods in their jobs, they are often reluctant to study them.

Luckily, there is some serious uptake of formal methods in industry. The classic case of a safety-critical industry is railway signalling, as described e.g. in [GM13]. Ligne 14 of the Paris Métro had software built using the B method [GM13] and has now run for over 20 years without a bug being reported. The “High Integrity Systems” unit of Altran develops systems for, e.g., the railway signalling industry and air traffic control, as well as tools and methodologies, such as the SPARK subset of Ada [MC15]. SPARK 2014 uses contracts to describe the specification of components in a form that is suitable for both static and dynamic verification.

Outside the safety-critical industry, a few ‘enlightened’, large information technology companies are beginning to use formal methods:

- Google is developing an ecosystem for formal analysis tools [SvGJ+15].
- Facebook uses “advanced static analysis” as described in [DFLO19].
- Amazon’s use of formal methods is discussed in [NRZ+15,BBC+19]. There is a more technical description of one component in [CCC+18].

[4] AlphaZero is the descendant of AlphaGo, the AI that became known for defeating Lee Sedol, the world’s best Go player, in March of 2016.

If we look at Facebook, [DFLO19] reports that, in many cases, "we have gravitated toward a 'diff time' deployment, where analyzers participate as bots in code review, making automatic comments when an engineer submits a code modification". For their Infer tool, which has its origin in the separation logic work of [CDOY11], they aim "for Infer to run in 15–20 min on a diff on average".

Similarly, at Altran, an attempt to check source code into the main repository (the equivalent of `git push`) generates a requirement to prove the appropriate contracts, and the verification conditions that ensure, for example, no numeric overflow. An important requirement here is that this verification be "reasonably fast". [BS12] describes their work here as "this changes the qualitative time band for a large scale industrial project from 'Nightly' to 'Coffee'." Both Facebook and Altran argue that the primary purpose of this time requirement is to avoid 'context switch' in the developer's brain.

Further changes could be initiated by academics. "Two-hundred-terabyte maths proof is largest ever" reported Nature in May 2016[5] and wrote: "Three computer scientists have announced the largest-ever mathematics proof: a file that comes in at a whopping 200 terabytes, roughly equivalent to all the digitized text held by the US Library of Congress. The researchers have created a 68-gigabyte compressed version of their solution – which would allow anyone with about 30,000 hours of spare processor time to download, reconstruct and verify it – but a human could never hope to read through it." The results that triggered this media interest concerns the Pythagorean Triples Problem. "We consider all partitions of the set $\{1, 2, \dots\}$ of natural numbers into finitely many parts, and the question is whether always at least one part contains a Pythagorean triple (a, b, c) with $a^2 + b^2 = c^2$. For example when splitting into odd and even numbers, then the odd part does not contain a Pythagorean triple (due to odd plus odd = even), but the even part contains for example $6^2 + 8^2 = 10^2$. We show that the answer is yes when partitioning into two parts, and we conjecture the answer to be yes for any finite size of the partition." [HK17] Such results triggering media interest could possibly change the situation. Another approach could be to organise, say, verification competitions at a student level. They would need to provide a stimulating social environment by being accessible to all students, and could be supported by elements such as cool prizes and free pizza.

2.4 Students' Mathematical Background

The seeming need for a solid mathematical background is often an argument against teaching formal methods. However, reflecting on the three elements of a formal method, grasping the syntax of a formal method is not more involved than understanding the syntax of a programming language: both are given by grammars. Grammars for formal methods are usually smaller than those for programming languages. However, students learn programming languages by trial and error on a computer, where the compiler/interpreter provides feedback on syntax errors. As discussed in Sect. 2.1, standard programming courses mostly

[5] Nature, 26 May 2016.

take an example-driven approach to syntax. In contrast, in formal methods students are often presented with a grammar for the syntax. For students, this often provides the first mathematical hurdle[6]. The challenge in formal methods teaching therefore lies in adopting a more example-driven style when it comes to syntax.

The semantics of a formal method is inherently mathematical in nature: in logic it is given in terms of the satisfaction of a formula by a model, process algebra utilizes structural operational semantics or denotational semantics, etc.

However, in a basic course focused upon the application of formal methods, it would be enough to point out that such formal semantics exists and to hint at its nature. The teaching challenge lies in providing an explorative approach to semantics via tools. In logic, this could follow ideas such as Tarski's world. In process algebra, one can explore processes by simulating them. In such a set-up, students could develop their own formal models and explore them, i.e., tools provide students with a similar feedback like running a computer program. Another idea would be to use a semantics compatible with the programming languages students are using. For instance in axiom-based testing, the 'axioms' can be interpreted as code in the programming language, thus utilising the students' programming background.

In an advanced course, in addition to such an explorative approach, the formal semantics itself needs to be presented. This will require a good mathematical background from the students.

Finally, the method aspect of a formal method is best presented through the use of a tool that automates the analysis in which one is interested. Running a tool would not require any mathematical background at all. Understanding the result of a method applied to a concrete example is usually immediate. An advanced course would address the mathematical details of why a method is sound.

These considerations refute the common prejudice that teaching formal methods requires students to have a profound mathematical background. An explorative teaching approach can make formal methods accessible even to students who like to program the 'artistic way'. This is supported by experience reports such as: "Engineers from entry level to principal have been able to learn TLA+ from scratch and get useful results in two to three weeks" [NRZ+15].

2.5 Personal Statements

In the order in which they were contributed, we present a number of personal statements by the co-authors.

SK. One challenge in teaching formal methods is to spark an initial interest. This is the case, because links are weak between formal methods and the current hot topics in computer science. Many students steer towards what currently is

[6] This is not eased by the often poor error messages provided by formal method tools.

perceived to dominate the future: data science and artificial intelligence, to name a just a few.

To overcome this, the formal methods community should strive to demonstrate its relevance, beyond 'classical' topics such as railway engineering. Correctness is as relevant in the new, upcoming areas of computer science as it is in the classical ones.

PK. A similar thought adding to SK: many students do not even have a clear idea of what formal methods are! They have heard of other areas such as machine learning, databases, operating systems, computer networks, compiler construction, and have an idea what is going on there. It's hard to encounter many aspects of formal methods in daily programming life, especially for a student with a limited view. So, why exactly would they pick a 'no-name' course such as "formal methods" or "model checking" over the other choices?

CD. The name of a course makes a big difference: students tend to avoid courses that already sound complicated (i.e. anything math or formal) in contrast to courses that sound 'useful' or 'applicable' or even just trendy. As a student, I had a course named "Modelling Computer Systems" that was on discrete mathematics. If it had been called "Discrete Mathematics", I'm sure it would have put several students on edge to begin with. Courses with names that contain tech buzzwords may also sound more appealing to students, such as cyber security, software testing, machine learning, artificial intelligence etc. We should consider these trendy subjects and adjust formal methods to be just as appealing, even if it means slightly adjusting course names.

MF. The lack of reliable tools that are suitable for teaching formal methods, as well as are scalable enough to demonstrate interesting and realistic use cases, creates a barrier for students. Throughout our course, we used a number of freely available formal methods and students struggled to understand the error messages and other feedback from the tools [FW20]. This kind of ambiguous feedback causes the students to lose interest and prevents them from engaging with the tools in a positive, constructive way. Furthermore, this usability issue also hinders the uptake of these tools in industry. This is somewhat of a vicious circle. Admittedly, most formal method tools are academic in nature and thus often are aimed at being good for publication. Better error messages and the like are often not prioritized that way. This causes the industrial uptake to miss, which decreases the focus again.

2.6 A Student's Personal Statement

RZ. My first introduction to formal methods was during my second year (right after introductory programming courses but before software engineering) in the GPU computing course. We used Petri nets for modelling the classic dining philosophers' problem. One of the motivations for using them was to avoid software failures. By providing a mathematical proof with Petri nets, so the professor claimed, we would be on course for success. At that time formal methods looked to me like an advanced technique in software development and a usual practice. My illusions were shattered later when another professor pointed out that it takes numerous assumptions for formal methods to work in the real world, and that often these assumptions do not apply.

3 Teaching Formal Methods — the Fun Way

In this section we collect a number of personal views and ideas on how teaching formal methods can be done the fun way. While some authors, see, e.g. [CRS+15], have written systematic accounts of the topic, here we present a number of personal statements in the order in which they were contributed.

MF. Games can be useful when it comes to teaching formal methods in the initial stages. However, to adequately demonstrate the importance of formal methods there must also be an emphasis on building and verifying software and not just on solving a puzzle, as entertaining as that may be. Of course, computer science students will find enjoyment in building systems, otherwise they would not be studying the subject. So, perhaps setting them the task of developing and verifying a simple, but realistic, model of a system would also be beneficial while encouraging them to have fun with formal methods. In this setting, games would ideally be placed at the beginning of the course as a light-weight and fun introduction.

JD. It is often difficult to motivate formal methods. Most students will not go into the construction of safety-critical systems, important though they are. Also, the specialist safety-critical companies tend to do their own training (though they would really like to have to do less!). It is perhaps easier to motivate formal methods with more common examples. The Chromium Project[7] is one example of ‘mainstream’ software, viz. browsers, and shows that the Chromium team is moving ‘more formal’.

SK. Usually, what makes any course interesting is the applications and the transfer of knowledge from classroom to reality. However, most formal method courses rely on examples that, while interesting, are far away from what students can experience and experiment with. We often rely on examples from industry and spend quite a lot of time explaining what a particular model is supposed to achieve exactly. I feel this often distracts students. Rather than focusing on what formal methods have to offer, we get lost in technical details. This is not the case with games, especially if considering well-known ones. Usually, the rules are known and (mostly. . . .) agreed upon already and we can focus on how a formal method can help us to get them right in our application.

Again, I strongly believe we should get away from the purely theoretical approach to teaching formal methods to beginners. At least for me, the theoretical

[7] https://www.chromium.org/Home/chromium-security/memory-safety.

advances in formal methods have always been a means to an end. In order to appreciate them, one has to experience what it means to try and reach the same end without them. This however falls short in programming education in general. Students proceed from smallish group projects to other smallish group projects, while only seldom have to experience larger refactoring, legacy code, etc. In an environment like this, formal methods are less useful. Let's teach our students what programming is like in reality: 90% of the work is reworking legacy code, fixing bugs and trying to understand why things are or are not working – by accident, this is where formal approaches could shine as well. Another aspect that could make a formal methods course interesting is to involve students in formal methods research rather than formal methods application. We used to teach formal methods by discussion software issues first and then having students try to find automatic ways to detect them, leading from simple static analysis ideas to model checking. The course has been thoroughly documented, also showing that the approach was highly motivating for students students [KKS19].

Notably, students (at least on the masters level) are able and willing to do 'actual research' in an inquiry-based course, eventually leading to publications [POKG19]. The inquiry- or research-based approach has taught students the internals of model checkers and how they can be efficiently implemented for prototypical languages.

PK. Shriram Krishnamurthi had a great Keynote at FM'19[8]. One of the main points to take away from that is that tools are a large issue. If you hit students with a full-blown industrial tool, they get frustrating error messages, because they have no idea what is going wrong (as the tool is able to understand a larger part of, e.g., a specification language than the student and raises errors related to other concepts). While it is nice to see that such tools are used in practice, they might be the wrong means to learn formal methods.

In Düsseldorf, our group has worked on an approach based on Jupyter notebooks [GL20]. It allows evaluation of smaller expressions or predicates without a state-based approach, so students can learn and experiment with the logical foundations of the language[9] where it is used to solve some logic puzzles). It can also be used to interact with B machines, so errors in a specification can be explained and documented in a nicer way (that you can replay). We think that might resolve some of the issues in teaching (but probably not all).

CD. Games are important, maybe even essential in teaching formal methods and making it fun. As a teacher of all ages from 8 years old to university level, I have found games to be one of the best tools to use when teaching. Students understand games and want to win them, naturally. When you explain to students that there is a method in which they are either guaranteed to win, or indeed a

[8] https://www.youtube.com/watch?v=UCwyOSHRBi0.
[9] https://gitlab.cs.uni-duesseldorf.de/general/stups/prob2-jupyter-kernel/-/blob/master/notebooks/tutorials/prob_solver_intro.ipynb.

method in which the second player cannot win, their interest levels peak! Students rush home to play the games against their parents and show off their new found ability.

Games can also be taught to most age groups. As said in our other paper in this volume "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 labelled transition systems" [MOPD20].

We have had success in asking 11 year olds to draw labelled transition systems. If we start teaching them sooner, this could act as a base from which we can build upon to further their understanding later on.

MF. In our experience [FW20] the students found it difficult to bridge the gap between the theory that was taught during the course (e.g. natural deduction proofs) and the associated tool support used during the lab sessions (e.g. Coq). As a result, I am inclined to agree with SK above in that the students need to see how these methods can work in reality rather than focus too much (although it is important and should be covered at some level) on the theory.

AC. The use of tools provides a great potential for introducing fun in teaching formal methods. This is particularly true for simulation and model-checking tools, whose emphasis is in giving "life" to formal specifications rather than getting involved in the complexity of a formal proof, as it happens, instead, for theorem-proving tools. Moreover, formal methods can be applied to a large range of problems, basically any problem, well beyond the domain of computer science. These give chances to teachers to propose fun problems, such as classical mathematical puzzles as well as popular games and even video games, and to learners to select problems that are close to their personal and professional interests [CL20]. One effective approach consists of providing learners with examples of formal methods descriptions of video games and inviting them to create formal models of their favourite video games. More in general, learners may be invited to define any problem they wish, formally specify/model it and carry out analysis with the support of tools. It is actually important to blur the distinction between learner and instructor by letting the learners drive the choice of exercises and use their creativity to identify and specify potential problems and invent new games. Blurring such a distinction will also contribute to instill in students a level of self-confidence that can lead students to carry out "actual research" [POKG19] and to actively contribute to curriculum development [Zhu20].

We can conclude our discussion on the teacher's view about fun by saying that if motivation is the dimension that allows learners to build up interest in formal methods, fun is actually the essential dimension to keep learners continuously engaged, thus assuring the retention and possibly increase of their interest over time [CL20, Cer16, RCS+21]. However, it is important that the fun occurs from the perspective of the student, not the teacher, and, if it is associated with some form of competition, this much effectively fosters motivation and does not cause frustration. In fact, nothing could be worse than "fun degenerating into frustration", which could be the case when a game that is fun for the teacher is actually too complex or uninteresting for the students or, especially in the case of school children, if the outcome of competition is interpreted by the student as a form of assessment [Cer20].

NS. Fun, puzzles, games and entertainment in teaching are not the unique ingredients needed to improve formal methods education (more general – computer science and software engineering education). All these (and something else) are just ways to engage (undergraduate) students with the learning, studying, comprehension and mastering of formal methods using curiosity and amusement. We believe that the experience of individual educators and expertise of research groups in the field of formal methods popularization deserves a positive attitude from the computer science, software engineering and (even) mathematics academic community and industry.

Another opportunity (just as an example) is a competitive spirit that is so appropriate for young people (in particular – for students of computer science and software engineering departments). International competitions between formal methods tools (e.g. automated theorem provers and satisfiability solvers) are popular, useful and valuable from the industrial and research perspectives, but not from the undergraduate education perspective. Unfortunately, competitions especially designed for (undergraduate) students (like Collegiate Programming Contest[10]) are still not involved in the education process in general and in formal methods education in particular. We hope that competitions of this kind may be used better for engaging students with theory of computer science and formal methods in software engineering [SY02].

PO. I also disagree with the 'puzzle'/'games'/'card tricks' approach. I do not think that they show the usefulness and relevance of formal methods. I also use small games (lots of them!) in my second-year course, up to blackjack, but only as small "toy examples" to get to know the modeling language and tool. On the other hand, real industrial applications, as others write here, are too large and complex to include in beginner's formal methods courses. A good compromise that I use (and describe in my FMfun'19 paper) are seminal systems/algorithms that are the cornerstones of different other domains and, equally important, of today's large software systems. For example, 2-phase-commit (while simple) and Paxos (less so) are still key building blocks in today's distributed systems. I include key designs from other courses and beyond, like cryptographic protocols (modeling and breaking NSKP), distributed transactions (2PC), distributed mutual exclusion, distributed leader election, transport protocols like TCP, ABP, sliding window, and so on. This shows the relevance of formal methods on many kinds of systems, and are small enough to easily model and analyze using formal methods, but might still give students (and other professors!) an idea of the usefulness of formal methods.

One final problem with games/tricks: even if you learn how to apply your formal method to model and analyze such games, can you then apply your formal method to a real distributed system such as Paxos or a cryptographic protocol?

[10] https://icpc.baylor.edu/.

I refer to my paper "Teaching Formal Methods for Fun Using Maude" [Ölv20] in this volume for a lengthier exposition of how I think formal methods should be taught at the undergraduate level.

3.1 Summarizing the Ideas

It is obviously impossible to establish general criteria to make formal methods teaching a fun activity. Fun cannot be characterised in an objective way and can only naturally emerge from the interaction between teachers and students. In fact, the emergence of fun is affected by the personalities of individual teachers and students as well as by the interaction context in which such different personalities meet in the classroom collaborative environment. Here, different criteria have been suggested and discussed, including:

– games and puzzles may represent a light-weight and fun introduction to formal methods;
– there should be an emphasis on building and verifying software for simple, but realistic, systems;
– teaching should focus on demonstrating that tools work rather than on delivering too much theory;
– students are likely to enjoy undertaking actual research activities;
– students should be involved in curricula development.

There is a general view among the co-authors that games and puzzles can be useful when it comes to teaching formal methods in the initial stages and represent a light-weight and fun introduction (MF, CD, AC). It is important to note that this view includes former formal methods students who became formal methods teachers [MOPD20]. Games may be also associated with some form of competition (AC, NS), which may be within-class (AC) or in terms of participation at an international context (NS). Games and puzzles are also a great tool to start formal methods education early, even by teaching to school level children, as young as 10–11 (CD, AC). Competition can also be beneficial in the context of school children, but should to carefully planned in order to avoid being interpreted by the student as a form of assessment, which therefore inhibits rather than motivates the students [Cer20].

In addition, there must also be some emphasis on building and verifying software (MF). However, such a connection with reality should be established in the right form to keep in line with the fun determined by the game-based approach. In fact, giving students the task of developing and verifying a simple, but realistic, model of a system would be beneficial while encouraging them to have fun with formal methods (MF). However, on the one hand, realistic, industrial systems are often far away from what students can experience and experiment with (SK) and most students will not go into the construction of safety-critical systems, important though they are (JD). On the other hand, the specialist safety-critical companies tend to do their own training (JD), which may provide a very different perspective from what students learn in formal methods courses. Moreover, focusing on examples from industry is very time consuming and often involves heavy technical details and, as a consequence, may be distractive rather than beneficial (SK). Instead, it might be more effective to motivate formal methods with more common, but still realistic examples, such as the Chromium Project (JD).

There is a general agreement among the co-authors that students need to see how formal methods work in reality using tools rather than focusing too much on the theory (SK, MF, PK, AC, PO). However, making students use industrial tools may result in heavy frustration. While it is nice to see that such tools are used in practice, they might be the wrong means to learn formal methods (PK).

An final aspect that could make a formal methods course interesting is to involve students in formal methods research rather than formal methods application (SK). In fact, students' publication are often highly appreciated [POKG19, Zhu20].

4 Increasing Visibility of Formal Methods Throughout the Curriculum

In common computer science and software engineering curricula, formal methods play a minor role. There are at most one or two specialized courses focusing on teaching formal methods. Often, these courses are only weakly linked to the rest of the curriculum.

Formal methods fail to link to the current hot topics in computer science and software engineering, both in teaching and research. In consequence, even students with considerable interest in software engineering are drawn towards courses such as data science, machine learning or artificial intelligence. However, now that artificial intelligence and machine learning techniques find their way into safety critical systems (such as autonomous cars), correctness considerations become more important every day.

The 'winner-takes-all' nature of today's software industry (where essentially *one* product/service in each category 'wins' and makes billions, and other solutions fade away, e.g., Facebook for social media; Google for search engines, eBay for online auctions, Zoom for online discussions/teaching/meetings) justifies an upfront investment in system quality. We note that major firms like Google [SvGJ+15], Facebook [DFLO19], and Amazon [NRZ+15] are all doing this, but this has yet to feed through to their hiring practices, or to students' perceptions of what they need to get a job at these favoured employers.

In consequence, an ideal integration of formal methods into a computer science or software engineering curriculum should first and foremost strive to present formal methods as a quality assurance tool to be used in other areas, be it embedded systems engineering or machine learning. This first contact to formal methods would aim at teaching usage scenarios as well as techniques and how they are to be deployed.

We believe that showing the benefit of formal methods by discussing applications to other areas will achieve two goals. First, it ensures code quality and system functionality are considered as critical. Furthermore, this initial contact to formal methods might spark an interest into their development and improvement. Both topics could then be a part of dedicated courses in formal methods.

While such a 'casual' approach would be ideal, it would require colleagues to be willing and to be able to teach small units on formal methods. This might be an unrealistic assumption. Organising 'guest sessions' from formal methods experts might be a way forward.

To gain an acceptance of having more formal methods visibility in a university curriculum, we need to persuade first and foremost our colleagues. Ultimately they decide whether/how/how much formal methods a university curriculum could/must contain. There is huge competition for places on a curriculum between the different specialties/fields. At least the older colleagues may remember times when formal methods were not too useful.

The 2013 "Curriculum Guidelines for Undergraduate Degree Programs in Computer Science" [ACM13] lists 18 "Knowledge Areas". In the following, we make a number of suggestions for formal method units in some of these areas:

AL-Algorithms and Complexity: formal verification of algorithms; model checking algorithms.

DS-Discrete Structures: logic, modelling, semantic foundations of formal methods.

HCI-Human-Computer Interaction: mode confusion problems; formal analysis of user dialogs; cognitive models.

IAS-Information Assurance and Security: formal analysis of security protocols.

IM-Information Management: specifying and analyzing both the correctness and the performance of cloud storage systems.

NC-Networking and Communication: protocol verification.

OS-Operating Systems: parallel modelling; scheduling.

PBD-Platform-based Development: formal model based development.

PD-Parallel and Distributed Computing: process calculi; Petri nets.

PL-Programming Languages: how to analyse software written in a specific programming paradigm; compiler correctness; semantics of programming languages; program correctness.

5 Syllabus of a Compulsory Formal Methods Course

Besides increasing the visibility of formal methods throughout all courses and also having specialised advanced courses on formal methods, we suggest that curricula for computer science and software engineering should include a compulsory formal methods course.

The target audience for such a compulsory formal methods course would be the complete cohort of computer science/software engineering students in year 2 or year 3 of a 3-year BSc degree programme.

Due to the wealth of available formal methods, we refrain from proposing a unified or 'standard' syllabus. Local expertise in specific formal methods and application domains should be taken into account. Therefore, we rather capture the essence of an ideal course in a generic way:

Introduction.

- The role of formal methods in the context of software engineering, see, e.g., Roggenbach et al. [RCS+21], Chapter 1, for a thorough discussion, and Barnes [Bar11] for a comparative case study.
- Success stories of formal methods, see, e.g., Roggenbach et al. [RCS+21] for a compilation of such stories, another good source is Section 1.3.4 of Garavel's report [GG13];
- Relating formal methods to current trends in computer science, such as machine learning, where one can use machine learning to improve formal methods [ALB18], or, a nascent field but one that is growing in importance and has already attracted the attention of ISO in the draft TR 24029-2, the application of formal methods to big data [vdA16, Cam14, MLM18] or to machine learning [HKW17, SKS19, WPW+18].

Main Part. The main part should offer one or two formal methods of different nature, e.g. a "model-oriented" and a "property-oriented" one, cf. [Win90] for further discussion of this classification; in order to demonstrate the 'universality' of formal methods, it would appear useful to draw examples from different domains.

The following topics (listed in no particular order) should be covered:

- Modelling: going from the informal to the formal; traceability; validation of models.
- Language design: explaining how the language of a formal method is designed for specific purposes (what are essentials necessary for expressivity, what is syntactic sugar easing the life of the specifier?).
- Semantics: presenting just the essentials – this needs to be one topic among many rather than the dominating one, as happens too often in current practice.
- Software engineering context: demonstrating that formal methods are applicable throughout the whole software lifecycle, e.g., in analysing designs, in software verification, testing from formal models.

– Method: systematically using tools to illustrate the 'method' aspect.
– Application domains: illustrate the reach of formal methods by selecting examples from different application domains. Safety, security, human-computer interaction, e-contracts, and non-computer areas (biological systems, ecology, chemistry) are some possible examples.
  Traditionally, formal methods teaching advocates the use of formal methods for safety-critical systems. Formal methods are of course super-important for those systems, but experience in class (and otherwise) suggests that this does not inspire and is almost counterproductive: most students do not foresee themselves designing the quite narrow range of safety-critical systems we tend to use as example (airplanes, cars, medical devices, etc.); focusing almost exclusively on safety-critical systems can actually be counterproductive as it (can be perceived to) send signals that formal methods are only usable for such systems.
  As cybersecurity failures are much in the news, we might look at these and see how formal methods might have found these (e.g. Heartbleed), or are being used (e.g. Chromium), as a way of emphasising the mainstream utility of formal methods.

Conclusion – Reflection on Formal Methods. We present below some items of reflective nature that ought to be addressed at the end of a formal methods course.

– General limitations: what formal methods can offer, what formal methods cannot deliver, e.g., based on Levenson's provocative article "Are You Sure Your Software Will Not Kill Anyone?" [Lev20].
– Scalability: why formal methods work on toy examples but their application might become impossible for technical reasons when it comes to real life challenges, see, e.g., [RMS+12] and [JMN+14]. [RMS+12] shows a formal methods in its early stages, where it can barely verify a toy example; [JMN+14] shows how, after two further years of research, with the help of abstractions it is possible to verify a real world example with the very same approach.
– Costs/benefits: what the cost and financial benefits of formal methods are [Bar11]. The key insight "Formal methods are surprisingly feasible for mainstream software development and give good return on investment." from Newcombe et al. [NRZ+15] and Amazon's "We can now use automated reasoning to provide inexpensive and provable assurance to customers" from J. Backes et al. [BBC+19] are probably a 'must have'!
– Acceptance: current uptake of formal methods in industry and reasons for the low acceptance.
– Current trends: where one expects the field of formal methods to be in, say, a decade.

Each lecturer will have her/his own subjective view concerning the above list of topics. Probably they offer a good point for discussion with students. The systematic element underlying them is that they ought to be addressed at the end of a formal methods course.

Learning Outcomes. Such a course would provide the learning outcomes that students

– understand the thinking behind formal methods and how it differs from ad-hoc programming;
– are fluent in the application of one or two formal methods to academic examples;
– are able to estimate the potential of formal methods with concrete challenges;
– are able to critically compare different formal approaches and choose the most appropriate for a given, specific application.

6 How to Assess Our Teaching Efforts?

Having introduced changes to teaching, it is important to assess if they have been successful. In this section, we collect a number of ideas as to how this could be done. In the order in which they were contributed, we present a number of personal statements.

MF. The obvious measurement is to compare exam results year after year, assuming that the same person teaches the course before and after any changes are made. We are working towards making some changes to our course that we could compare against the previous years' results. However, it is also important to survey the students before and after the course as well as during the lab sessions to really understand how they are progressing and how effective the notes, teaching and lab sessions are in improving their formal methods expertise.

CD. As a teacher on a Degree Apprenticeship programme, I think one such method of assessing our own teaching methods, is to actually assess the students' level of understanding by getting them to apply formal methods in their workplace: students on our programme are employed. Often, when we teach formal methods, our students have never seen them before. We tasked our students with producing a work-based portfolio where they have to apply discrete mathematics to their workplace. Whilst some students struggle with the task, for most of them it becomes apparent how beneficial it is in the workplace. Sometimes it even highlights issues with the existing systems logic. In my opinion this is the best outcome and therefore would demonstrate that we have been teaching successfully.

SK. The formal methods community ought to reflect on what it wants to achieve in teaching. Ultimately, there is no use in being able to enumerate different formal methods and just being able to use them if you don't see any reason to do so. Rather, I am in favour of indeed trying to change (and measure/evaluate) students' opinions and attitudes.

Employing a formal approach to software engineering is all about the resulting quality of the product. Thus, a formal methods course needs to change students' perceptions about software as a product that is used in different applications and situations – eventually, even in safety critical ones. Nobody would cross a bridge that seems like it might collapse. At the same time, delivering software that is known to cease working under certain conditions has become quite accepted. Once students gain an awareness and consciousness for quality aspects of software, formal methods (and the effort to use them) will appear more beneficial.

MR. In my teaching experience, students best learn those topics that they like to do, that they can try themselves, and that provide them with a feeling of achievement. For teaching practice in formal methods that means that we ought to run supporting lab classes. These would offer meaningful examples on which students can successfully apply a formal method, or explore why some specific formal method fails. In my view, lab tasks would be well-designed if, say, 80% of the students can solve them, i.e., offer them a sense of achievement.

The other objective would be to educate the majority of computer science students in such a way that they are capable of applying formal methods in their future careers in industry. This could be evaluated by looking at dissertations: do the majority of them report on the application of formal methods when the project concerns software development?

PK. One criterion could be the number of students that are interested in writing their dissertation in the field of formal methods. In particular, our experience is that while formal methods are not in high demand with students, the ones who finish our formal methods courses usually are willing to gain an expert level of knowledge. Many students stay interested, once they have developed an appetite for formal methods.

AC. Assessing the effect of teaching changes in standard formal methods courses is a tricky task for a number of reasons:

1. classes are normally small;
2. even within the small group there is often a large variety in background and interest of the students;
3. although students might be interested and even successful in using formal methods, in their future research or work goals they are driven by more trendy areas and topics, where there is little place for the use of formal methods.

Reason 1 prevents us from collecting enough data to allow us to produce statistically significant results. It is therefore more important to informally collect personal opinions from students through discussions, open-ended questionnaires and interviews, rather than analysing numerical data such as grades and percentage of successful students.

Reason 2 requires an initial assessment of the students to be compared with the final objectives that they achieve at the and of the course (see MF's statement earlier in this section). A possible form of initial assessment is a questionnaire to be administered during the very first course class. The questionnaire should aim at the assessment of

– mathematical background;
– logical and problem solving skills;
– experience with the logic and functional programming paradigms;
– knowledge of the software engineering concepts that are central in formal methods, such as specification, testing, verification, validation, assurance.
– knowledge of basic logical and set-theoretic concepts such as syntax, semantics, theorem, proof, function and more specific computability concepts such as decidability, enumerability, undecidability.
– perception of more "exotic" formal methods concepts such as system state and concurrent system.

Due to Reason 3, looking at dissertations or careers of former students does not really provide a measure of the achievement of learning objective. In fact, students' pragmatics in looking for a thesis topic or choosing their professional career may clash with their academic interests.

6.1 Summarizing the Ideas

Assessment is often an exercise of producing numbers that can be compared over several academic years. Here, different criteria have been suggested and discussed, including:

– exam results of a particular course;
– number of dissertations in which formal methods are applied; and
– number of dissertations in the area of formal methods.

AC provided arguments why one should look at such numbers with care.

For teaching a formal method it has been suggested to closely survey students during the course (MF), and to design lab classes with 'guaranteed success', i.e., which are barely contributing to a differentiation between students in form of marks (MR).

A slightly deeper looking approach would be to look at students' opinions and attitudes and see how they change over time (SK).

7 Conclusion and Outlook

In this white paper, we have analysed why formal methods are seldom prominently included in computer science and software engineering curricula. One often heard reason for this is that they fail to attract students. However, we believe that students often just have misconceptions about formal methods. Also, the 'coolness factor' of formal methods is low. Finally, formal methods are not visibly used by industry. It is a myth that formal methods teaching on a basic level would require a particularly strong mathematical background. We provided a number of ideas on how to make formal methods more attractive to students and gave examples of the uptake of formal methods in industry beyond the critical systems sector.

In the spirit of the workshop "Formal Methods – Fun for Everybody", this paper has collected a number of 'sparkling ideas' that aim at improving the situation summarised above. We grouped such ideas into four categories, namely individual teaching delivery, cf. Sect. 3, making formal methods visible throughout the syllabus, cf. Sect. 4, the proposal of a compulsory formal methods course, cf. Sect. 5, and ideas about how to measure the effect of teaching changes, cf. Sect. 6.

With this white paper a start has been made to make formal method teaching more popular. The ideas and arguments presented are ready to be picked up in order to improve existing courses, to design new courses, and to make formal methods more prominent in academic curricula. The participants of the 2019 workshop were enthusiastic about this topic, and we hope to have shared some of this enthusiasm with the reader. Let's turn this into a wider movement!

References

[ACM13] ACM. Computer science curricula 2013: Curriculum guidelines for undergraduate degree programs in computer science (2013). http://dx.doi.org/10.1145/2534860

[ACM15] ACM. Software engineering 2014: Curriculum guidelines for undergraduate degree programs in computer science (2015). https://doi.org/10.1145/2965631

[ALB18] Amrani, M., Lucio, L., Bibal, A.: ML + FV = ♡? A survey on the application of machine learning to formal verification. arXiv Software Engineering (2018)

[Bar11] Barnes, J.E.: Experiences in the industrial use of formal methods. In: Romanovsky, A., Jones, C., Bendiposto, J., Leuschel, M., (eds.) AVoCS 2011. Electronic Communications of the EASST (2011)

[BBC+19] Backes, J., Bolignano, P., Cook, B., Gacek, A., Luckow, K.S., Rungta, N., Schaef, M., Schlesinger, C., Tanash, R., Varming, C., Whalen, M.: One-click formal methods. IEEE Softw. **36**(6), 61–65 (2019)

[BDK+06] Brakman, H., Driessen, V., Kavuma, J., Bijvank, L.N., Vermolen, S.: Supporting formal method teaching with real-life protocols. In: Formal Methods in the Teaching Lab (2006). http://www4.di.uminho.pt/FME-SoE/FMEd06/Preprints.pdf

[BLA+09] Blanco, J., Losano, L., Aguirre, N., Novaira, M.M., Permigiani, S., Scilingo, G.: An introductory course on programming based on formal specification and program calculation. SIGCSE Bull. **41**(2), 31–37 (2009)

[Bou09] Boute, R.: Teaching and practicing computer science at the university level. SIGCSE Bull. **41**(2), 24–30 (2009)

[BS12] Brain, M., Schanda, F.: A lightweight technique for distributed and incremental program verification. In: Joshi, R., Müller, P., Podelski, A. (eds.) VSTTE 2012. LNCS, vol. 7152, pp. 114–129. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-27705-4_10

[Cam14] Camilli, M.: Formal verification problems in a big data world: towards a mighty synergy. In: Proceedings of ICSE 2014, pp. 638–641. ACM (2014)

[CC82] Cooper, D., Clancy, M.: Oh! Pascal. W.W. Norton & Company Inc., New York (1982)

[CCC+18] Chudnov, A., et al.: Continuous formal verification of Amazon s2n. In: Chockler, H., Weissenbacher, G. (eds.) CAV 2018. LNCS, vol. 10982, pp. 430–446. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-96142-2_26

[CDOY11] Calcagno, C., Distefano, D., O'Hearn, P.W., Yang, H.: Compositional shape analysis by means of bi-abduction. J. ACM **58**(6), 26:1–26:66 (2011)

[Cer16] Cerone, A.: Human-oriented formal modelling of human-computer interaction: practitioners' and students' perspectives. In: Milazzo, P., Varró, D., Wimmer, M. (eds.) STAF 2016. LNCS, vol. 9946, pp. 232–241. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-50230-4_17

[Cer20] Cerone, A.: From stories to concurrency: How children can play with formal methods. In: A. Cerone and M. Roggenbach (eds.) FMFun 2019, CCIS 1301, pp. 191–207. Springer, Cham (2017)

[CL20] Cerone, A., Lermer, K.R.: Adapting to different types of target audience in teaching formal methods. In: A. Cerone and M. Roggenbach (eds.) FMFun 2019, CCIS 1301, pp. 106–123. Springer, Cham (2017)

[CRS+15] Cerone, A., Roggenbach, M., Schlingloff, B.-H., Schneider, G., Shaikh, S.A.: Teaching formal methods for software engineering - ten principles (2015). https://www.informaticadidactica.de/uploads/Artikel/Schlinghoff2015/Schlinghoff2015.pdf

[DD07] Deitel, P.J., Deitel, H.M.: Java - How to Program, 7th edn. Pearson Education Inc., Upper Saddle River (2007)

[DFLO19] Distefano, D., Fähndrich, M., Logozzo, F., O'Hearn, P.W.: Scaling static analyses at Facebook. Commun. ACM **62**(8), 62–70 (2019)

[DS18] Dewar, R.B.K., Schonberg, E.: Computer science education: Where are the software engineers of tomorrow? CROSSTALK - The Journal of Defense Software Engineering (2018)

[Flo67] Floyd, R.W.: Assigning meaning to programs. Math. Aspects Comput. Sci. **19**, 19–32 (1967)

[FW20] Farrell, M., Wu, H.: When the student becomes the teacher. In: A. Cerone and M. Roggenbach (eds.) FMFun 2019, CCIS 1301, pp. 208–217. Springer, Cham (2017)

[GG13] Garavel, H., Graf, S.: Formal Methods for Safe and Secure Computers Systems. Federal Office for Information Security (2013). https://www.bsi.bund.de/DE/Publikationen/Studien/Formal_Methods_Study_875/study_875.html

[Gib08] Paul Gibson, J.: Formal methods: never too young to start. In: Proceedings of FORMED 2008, pp. 151–160 (2008)

[GJS+20] Gosling, J., et al.: The Java language specification - Java SE 14 Edition. Technical Report JSR-389 Java SE 2014, Oracle America, February 2020

[GL20] Geleßus, D., Leuschel, M.: ProB and Jupyter for logic, set theory, theoretical computer science and formal methods. In: Raschke, A., Méry, D., Houdek, F. (eds.) ABZ 2020. LNCS, vol. 12071, pp. 248–254. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-48077-6_19

[Gla00] Glass, R.L.: A new answer to "how important is mathematics to the software practitioner?". IEEE Softw. **17**(6), 136 (2000)

[GM13] Gnesi, S., Margaria, T.: Some Trends in Formal Methods Applications to Railway Signaling, pp. 61–84 (2013)

[HK17] Heule, M.J.H., Kullmann, O.: The science of brute force. Commun. ACM **60**(8), 70–79 (2017)

[HKW17] Huang, X., Kwiatkowska, M., Wang, S., Wu, M.: Safety verification of deep neural networks. In: Majumdar, R., Kunčak, V. (eds.) CAV 2017. LNCS, vol. 10426, pp. 3–29. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63387-9_1

[HW73] Hoare, C.A.R., Wirth, N.: An axiomatic definition of the programming language PASCAL. Acta Inf. **2**, 335–355 (1973)

[ISO90] ISO 7185:1990 Information technology - Programming languages - Pascal (1990)

[JMN+14] James, P., Moller, F., Nga, N.H., Roggenbach, M., Schneider, S.A., Treharne, H.: Techniques for modelling and verifying railway interlockings. Int. J. Softw. Tools Technol. Transf. **16**(6), 685–711 (2014)

[KKS19] Krings, S., Körner, P., Schmidt, J.: Experience report on an inquiry-based course on model checking. In: Tagungsband des 16. Workshops zu Software Engineering im Unterricht der Hochschulen, CEUR, vol. 2358 (2019)

[Lev20] Leveson, N.: Are you sure your software will not kill anyone? Commun. ACM **63**(2), 25–28 (2020)

[MC15] McCormick, J.W., Chapin, P.C.: Building High Integrity Applications with SPARK. Cambridge University Press, Cambridge (2015)

[MLM18] Mandrioli, C., Leva, A., Maggio, M.: Dynamic models for the formal verification of big data applications via stochastic model checking. In: Proceedings of CCTA 2018, pp. 1466–1471. IEEE Computer Society (2018)

[MOPD20] Moller, F., O'Reilly, L., Powell, S., Denner, C.: Teaching them early: formal methods in school. In: A. Cerone and M. Roggenbach (eds.) FMFun 2019, CCIS 1301, pp. 173–190. Springer, Cham (2017)

[NRZ+15] Newcombe, C., Rath, T., Zhang, F., Munteanu, B., Brooker, M., Deardeuff, M.: How Amazon web services uses formal methods. Commun. ACM **58**(4), 66–73 (2015)

[Ölv20] Ölveczky, P.: Teaching formal methods for fun using Maude. In: A. Cerone and M. Roggenbach (eds.) FMFun 2019, CCIS 1301, pp. 58–91. Springer, Cham (2017)

[Pat94] Pattis, R.E.: Teaching EBNF first in CS 1. In: Proceedings of the Twenty-Fifth SIGCSE Symposium on Computer Science Education, SIGCSE 1994, New York, NY, USA, pp. 300–303. Association for Computing Machinery (1994)

[POKG19] Petrasch, J., Oepen, J.-H., Krings, S., Gericke, M.: Writing a model checker in 80 days: reusable libraries and custom implementation. In: Proceedings of AVoCS 2018, vol. 76, Electronic Communications of the EASST (2019)

[RCS+21] Roggenbach, M., Cerone, A., Schlingloff, B.-H., Schneider, G., Shaikh, S.A.: Formal Methods for Software Engineering. Springer, Switzerland (2021)

[RMS+12] Roggenbach, M., Moller, F., Schneider, S., Treharne, H., Nguyen, H.N.: Railway modelling in CSP||B: the double junction case study. ECEASST, 53 (2012)

[Sek06] Sekerinski, E.: Teaching the mathematics of software design. In: Formal Methods in the Teaching Lab (2006). http://www4.di.uminho.pt/FME-SoE/FMEd06/Preprints.pdf

[SKS19] Sun, X., Khedr, H., Shoukry, Y.: Formal verification of neural network controlled autonomous systems. In: Proceedings of HSCC 2019, pp. 147–156. ACM (2019)

[SvGJ+15] Sadowski, C., van Gogh, J., Jaspan, C., Söderberg, E., Winter, C.: Tricorder: building a program analysis ecosystem. In: 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering, vol. 1, pp. 598–608 (2015)

[SY02] Shilov, N.V., Yi, K.: Engaging students with theory through ACM collegiate programming contests. Commun. ACM **45**(9), 98–101 (2002)

[vdA16] van der Aalst, W.: Process Mining - Data Science in Action, 2nd edn. Springer, Heidelberg (2016)

[vRtPdt20] van Rossum, G., the Python development team: the Python Language Reference - Release 3.8.3. Python Software Foundation, June 2020. Retrieved 2020–06-15

[Win90] Wing, J.: A specifier's introduction to formal methods. IEEE Comput. **23**(9), 8–22 (1990)

[WPW+18] Wang, S., Pei, K., Whitehouse, J., Yang, J., Jana, S.: Formal security analysis of neural networks using symbolic intervals. In: Proceedings of Sec 2018, pp. 1599–1614. ACM (2018)

[Zhu20] Zhumagambetov, R.: Teaching formal methods in academia: a systematic literature review. In: A. Cerone and M. Roggenbach (eds.) FMFun 2019, CCIS 1301, pp. 218–226. Springer, Cham (2017)

Axiom Based Testing for Fun and Pedagogy

Magne Haveraaen(✉)

Bergen Language Design Laboratory, Department of Computer Science, University of Bergen, Bergen, Norway
Magne.Haveraaen@ii.uib.no
https://bldl.ii.uib.no/

Abstract. The intricacies of programming demands a precise understanding of a programming language, its libraries and its conventions. Programming education normally does not emphasise this well enough, often leaving technicalities vague and omitting corner cases. In addition, students starting computing studies at university have about 12 years of basic education behind them. The intuition the students bring with them to university is in many ways at odds with the technicalities of programming. This mismatch is later the source of many computer related vulnerabilities and in some cases causes disasters.

Here we advocate *axiom based testing* as a technique to master these demands. Axiom based testing is based on programming language semantics, not mathematical semantics. There is *no need* for quantifiers in the specifications, and *no need* for formal reasoning to achieve the benefits. Using axiom based testing does not presuppose more than the ability to write methods and assertions, a part of any beginning programming course. Thus axiom based testing is a *very lightweight* formal method. It can be used both to understand other people's code, e.g., libraries and APIs, and to validate own code. Axiom based testing integrates naturally with unit testing and can be an aid for both students and practitioners in getting the technical details right. Our experience is that axiom based testing can easily be taught at the undergraduate level. It is as fun to use as unit testing, giving the same direct feedback.

This paper contains many hands on examples, mostly in Java. It can be the basis for self studies in axiom based testing for Java or for teaching axiom based testing for any programming language.

Keywords: Axiom based testing · Parameterised unit tests · Lightweight formal methods · Java collection classes · Behavioural subtyping · Liskov substitution principle · Software vulnerabilities · Programming caveats · Teaching object-orientation

1 Introduction

Software is a complex system of intricate technical details interacting in order to achieve some overall behaviour. A software developer must master this in the formal system defined by the programming language, its standard libraries and conventions. Unfortunately this formal system can be at odds with the programmer's training, making the development task hard and error prone.

Most programming languages lack a formal definition and resort to stylised English (*standardese*) in overly large documents, e.g., [14,21,22]. This makes the task of gaining a precise understanding of the programming language overwhelming. Contemporary programming languages also come with a standard library, sometimes defined outside of the language standard document [23]. In order to master programming one needs to master a changing selection of the types and operations of the language and its libraries, whether the libraries are standard, third party, or programmer developed. Hoare and Wirth's formalisation of Pascal used axioms to capture the semantics of the intrinsic types [18]. Axioms have a sound theoretical basis. They can be applied on the types and methods we are interested in, e.g., on the intrinsics and libraries we actually use in a software system. Sets of axioms can be composed while keeping the insights already gained from the component specifications.

Here we advocate using axioms as test oracles for understanding data types—and for placing requirements on our own code. *Axiom based testing* is using axioms as test oracles for checking code against expected properties [12]. This is useful for newly developed code, as a generalisation of test driven development [2], but also for gaining a precise understanding of existing types, e.g., in libraries or programming language intrinsics. Programming is a setting where the semantics of computers, as opposed to the mathematical ideas we learnt in school, is significant. Mastering the corner cases is important to avoid vulnerabilities in software.

Axiom based tests can be written as methods with assertions. The parameters to the methods are the variables of the axioms. This blends naturally with unit testing which uses assertions in method bodies to express the tests. The only difference is that axioms are parameterised tests [28] and need a selection of test data to be used for unit testing.

In objected oriented programming axiom based tests capture the behavioural aspect of classes. Class hierarchies organised according to the Liskov substitution principle [26] will inherit these axioms. A prominent example is the Java standard library [23] which is designed around the substitution principle. In addition, many of the Java standard library specifications are expressed as algebraic properties. These can readily be written as axiom based tests using JUnit assertions [24]. A nice aspect is that these tests are written as Java (JUnit) code. The student thus needs no additional mathematical background in order to write and explore these axioms. A basic programming background is sufficient. And checking axiom based tests gives direct feedback through the unit testing system in the same way as good unit tests.

This is lightweight formal methods at its best:

the *fun* of programming, the *feedback* of unit testing, and
the *precision* of axioms, all in one.

In the rest of this paper we next present the ideas of axiom based testing and behavioural subtyping (the Liskov substitution principle). Then we look at how these are engrained into the Java standard library. In Sect. 4 we turn the tool of axiom based testing to understanding the computer integers and how they deviate from mathematical integers. We then have a short interlude with floating point numbers before the final summary and conclusion.

2 Axiom Based Testing and Liskov Substitution Principle

Here we first present the idea of axiom based testing and place it in a conceptual and pragmatic context. Then we see how it supports behavioural subtyping—known as the Liskov substitution principle in object-oriented programming.

2.1 Axiom Based Testing

The principle of *axiom based testing* is writing parameterised test oracles, i.e., test oracles that can be applied to arbitrarily many values of the appropriate types. Thus writing axiom based tests is as simple as writing a parameterised test oracle, which is as simple as writing a method with an assertion. Since assertions are part of most modern programming languages, and are well supported by all unit testing frameworks, axiom based testing can be integrated in most modern software development practices.

From a formal methods perspective, axiom based testing is a natural stepping stone. It sensitises a student or practitioner to the idea of writing precise definitions for their code. It also prepares them for understanding algebraic specifications, central to understanding properties of both mathematical and computer structures. This includes the pragmatics of correctly using, e.g., collection classes from a programming language library.

Historically formal specifications has been a mathematical discipline, while testing has originated from pragmatical considerations in software development. This creates a divide in the semantics of the two worlds, which can lead to severe integration problems [29].

The DAISTS system countered this by showing that axioms can be used as test oracles embedded in the programming language [12]. This integrates the fun and immediate feedback of testing with the precision of axioms. The DAISTS tool merges the classes being tested, the properties expected of the class (axioms), a set of data points, and a monitoring library, into a test program.

From a programming viewpoint the principles used in DAISTS was to write an axiom as an assertion in a procedure (the axiom), placing the free variables of the axiom as parameters to the procedure. In modern parlance we can consider the monitoring library as a unit testing framework. This gives a parameterised unit test [28] with the axiom as the test oracle, e.g., as embodied in JUnit 5 [24].

To reap the benefits of axiom based testing, we can make the axiom generic. Then the types and operators of the axiom's generic arguments are instantiated by the classes under test.

The JUnit parameterised test sets are normally limited to a finite list of test cases. Our JAxT (Java Axiom Testing) Eclipse [11] plugin [16,25] supports randomised testing integrated with JUnit. JAxT also comes with a collection of test oracles for the Java standard library, and most importantly, axioms checking user code against requirements for the Java Collection classes (see Sect. 3).

The principles used for axiom testing are not limited to the mentioned languages. We have developed Catsfoot for C++ testing [1,9] which allows a mix of crafted and randomised tests. The most well known system for axiom based testing is QuickCheck [6]. QuickCheck embodies a very clever mechanism for searching for a minimal counter example for axioms that fail. It also has good approaches to finding non-trivial randomised data for conditional axioms. The QuickCheck tool was originally developed for Haskell, but now has been extended to many languages, most notably Erlang.

In axiom based testing the notation, semantics and logical expressions of the programming language are used when formulating the axioms. This goes very well for equational properties, conditional equational axioms, and in general for any kind of boolean expression. Boolean expressions are typically the languages' mechanism for writing conditionals and are thus very familiar for any software developer or student of programming.

The more powerful logics, such as first order predicate logic are not that easy to embed into axiom based tests. Predicate logic is, for instance, heavily used when verifying algorithms as in pre/post specifications [10,17,27]. The observation that algebraic specifications rarely need first order, and often can do with equational specifications, is beneficial for axiom based testing as a practical approach to formal methods. Additionally, there is theoretical backing for the observation that enlarging an API will allow a simpler specification logic [4]. Pragmatically using a simpler, more easily testable specification logic, has the cost of implementing a larger API. Often this additional cost gives a more versatile class, more often a benefit rather than a drawback.

2.2 Behavioural Subtyping and Axioms

Subtyping is a relationship between types that allow data of a subtype to be used in place of a supertype. There are two levels to this. The first level is ensuing type safety, such that runnable code can be generated for the subtype when it has been checked for the supertype. Thus the subtype must, e.g., support the same methods as the supertype. The second level is to what extent the subtype is semantically related to the supertype. That is, how is the subtype constrained by the supertype such that any program expecting supertype data should compute essentially the same result given subtype data.

There are various approaches to *behavioural subtyping* that attempts to formalise "constrained by the supertype". Here we will understand behavioural subtyping as when a subtype inherits a specified set of properties of a supertype. Defining such a property by an axiom, this implies that the axioms for a supertype are valid for the subtype. For axiom based testing it follows that selected axioms valid for a type can be used as test oracles for all its subtypes.

In the object oriented setting behavioural subtyping is known as the Liskov substitution principle [26]. According to the principle, object oriented hierarchies are designed to propagate properties from superclasses to subclasses. This is in contrast to the actual inheritance mechanism which is a code reuse mechanism [8].

The Java standard library [23] is designed according to the Liskov substitution principle. Its documentation also has a clear algebraic flavour, making it easy to make properties and requirements into axiom based test oracles in Java/JUnit. A significant benefit of this is that the code that defines the axioms has the same semantic interpretation as the programming language itself.

The JAxT tool [25] mentioned above traces the Java inheritance hierarchies when deciding which axioms to apply to a given class for testing purposes.

3 Behavioural Subtyping – Java Style

We will use behavioural subtyping as a perspective on the class hierarchy of Java and its standard library. Studying the Java collection classes specifically, it is clear that they have been designed with axiom inheritance in mind.

All classes in Java belong to a single inheritance hierarchy rooted in the class `Object` while Java's primitive types are stand-alone.

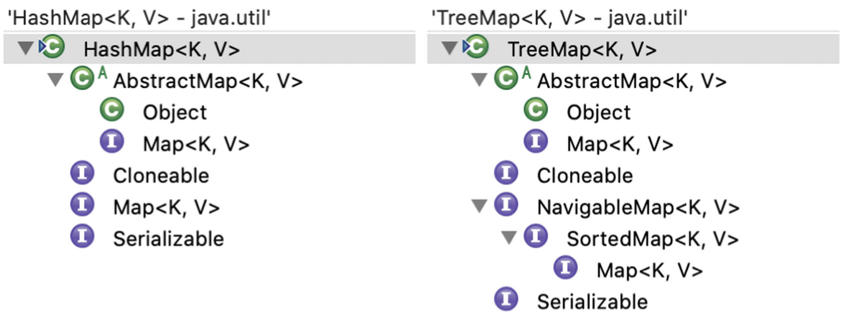
Axiom based testing will help us see how we can gain a precise understanding of a class and its requirements. These requirements are essential to get right in order to use the standard library appropriately. A limited, informal understanding can miss out on important corner cases, leaving software with reliability and security vulnerabilities. Nothing of this is specifically tied to Java, though it is a nice candidate to study due to the documentation style used in the standard libraries.

In Sects. 4 and 5 we will apply the same axiom based testing methods to gain insight in the primitive data types (integral and floating point, respectively) of Java. Java has a strict separation between primitive types and classes. The primitive types include booleans, integers, floats, and characters.

A Java class inheriting another class gets access to all declarations and methods of the superclass (modulo visibility issues). New declarations of field variables may overshadow the superclass's field variables, while new declarations of methods override the superclass's methods, i.e., replaces the algorithms.

The documentation of the Java standard library emphasises what properties should be expected, and relates in several distinct ways to the class hierarchy [23].

- Properties that should hold for this class and all subclasses.
  For instance that the `equals` method of `Object` (and all subclasses) should be an equivalence relation. This is behavioural subtyping in practice.
- Properties that hold for this class only, and is not expected to hold for subclasses.
  For instance that `Object`'s `equals` method is comparing object references and not values. This is OK since the algorithms used for a class's methods have specific properties and these can be made clear.



```
'HashMap<K, V> - java.util'          'TreeMap<K, V> - java.util'
HashMap<K, V>                         TreeMap<K, V>
  AbstractMap<K, V>                     AbstractMap<K, V>
    Object                                Object
    Map<K, V>                             Map<K, V>
  Cloneable                             Cloneable
  Map<K, V>                             NavigableMap<K, V>
  Serializable                            SortedMap<K, V>
                                            Map<K, V>
                                        Serializable
```

Fig. 1. Examples of Java standard library hierarchies: `HashMap` hierarchy to the left, `TreeMap` hierarchy to the right. A green circled C represents a class, a blue circled I an interface. Note that the `SortedMap` properties take precedence over the `Map` properties for `TreeMap`. (The illustration is made using Eclipse's *Open Type Hierarchy* feature [11].)

- Properties that do not hold for this class, but should hold for all subclasses. This is OK since these are properties that will be introduced one level down from where they are described.
- Properties that are recommendations only.
  For instance that a class's *natural order* should be compatible with its `equals` method. This is OK since these are properties that may be introduced one level down from where they are described.
- Properties that take precedence over other properties.
  For instance, `SortedMap` properties take precedence over `Map` properties, see `TreeMap` in Fig. 1. Properties that take precedence over other properties break the Liskov substitution principle.
- Properties that appear multiple places in the hierarchy, but have no formal relation.
  This is the case with, e.g., properties for `Stack` and its recommended replacements, the `Deque` implementations, and the properties for primitive types. The lack of a relevant common superclass forces us to copy-paste the specification in order to adapt the axiom.

The textual description, in English, of the properties follow an algebraic style based on conditional equations. This blends well with axiom based testing.

In addition to the inheritance hierarchy, Java has interface declarations. A class may implement one or more interfaces, and interfaces may extend one or more interfaces. Interfaces form multiple inheritance hierarchies of declarations. The textual description of the standard library's interfaces follow the same algebraic style as for classes. When a class explicitly implements one or more interfaces it is bound by the interfaces' behavioural description. All subclasses of such a class implicitly implement the interfaces and their axioms. This quickly becomes convoluted, see Fig. 1 to get an impression.

For Java generic classes interfaces and inheritance play an important role. A generic parameter may require all instantiations to implement an interface or be a subclass of a specific class. This ensures that the generic code can expect all declared methods from the generic parameter to be available. The Java type checker enforces that only declared methods can be used in the generic code. Further, the generic code may assume the described properties of a generic parameter class. Thus a method sorting arrays of generic `Comparable` elements can rely on the properties of the generic element's `compareTo` method defining its *natural order*. Any generic parameter, whether explicitly stated or not, is assumed to be a subclass of `Object`.

3.1 Java Class Requirements – HashMap

The `HashMap` class has two generic parameters, `K` for *key* and `V` for *value*. The class represents dictionaries of key-value pairs, mapping stored keys to values. The `HashMap` is a `Map`, so it satisfies the following property, here written as an axiom in Java.

```
public static <K, V>
void retrieveEqualsAxiom(Map<K, V> map, K key, V value, K k2) {
  V v1 = map.get(k2);
  map.put(key, value);
  V v2 = map.get(k2);
  if (key.equals(k2))
    assertEquals(value, v2);
  else
    assertEquals(v1, v2);
}
```

According to the Java type system, this axiom can be used for any class implementing (directly or indirectly) the `Map` interface. The behavioural properties of the class hierarchy decides whether the axiom is relevant for any specific such class. The `assertEquals` method is a JUnit method for checking that two values are equal, using the appropriate `equals` method (or regular `==` for primitive types). Apparently neither generic parameter `K` nor `V` has any constraints, and it should be fine to use any class for these.

There is a subtle point in Java, which is often omitted from beginning courses in programming:

Java's `Object` class description places strict constraints on every Java class.

Specifically every, class, whether developer defined or available in an external library, must provide both an `equals` method and a `hashCode` method prone to specific requirements. These are utilised by the Java standard library classes, e.g., `HashMap<K,V>`. Any mistake in the implementation of these two methods may leave data unretrievable in a collection. Not overriding these methods may also induce errors.

To expose this problem, consider a simple `Person` class with two data fields, a name and a numerical post code (post codes in Norway are 4 digit numbers and fit into Java's `short` integer data type).

```java
public class Person {
  String name;
  short postcode;
  public Person(String name, int postcode) {
    this.name = name;
    this.postcode = (short) postcode;
  }
}
```

Assume we are building a small hospital support system, and we will use the `Person` class above as key for storing and retrieving electronic patient records (EPRs) in a hash map. But according to Java semantics, two `Person` objects with identical values will not be equal.

```java
  Person a = new Person("Ole",5000);
  Person b = new Person("Ole",5000);
  assert a != b; // Distinct objects have distinct  pointers
  assert ! a.equals(b); // Class Object's equality method is inherited
```

Though technically not breaking the `retrieveEqualsAxiom` property above, it seems wrong not to be able to retrieve person `b` when person `a` is stored in the map, as in the following Java snippet.

```java
  HashMap<Person,EPR> patients = new HashMap<Person, EPR>();
  Person a = new Person("Ole",5000);
  Person b = new Person("Ole",5000);
  patients.put(a, epr);
  EPR epra = patients.get(a); // Retrieves stored EPR.
  assert epra.equals(epr);
  EPR eprb = patients.get(b); // Does not retrieve stored EPR.
  assert eprb == null;
```

It would be necessary to retrieve patients by `Person` data, not just by the pointer to the original `Person` object. To fix this, we need to implement an equals method for `Person`.

Most integrated development environments (IDEs) will have options for generating matching equals and hash code methods. This normally is a very good sketch for such implementations. Note that not overriding these two functions causes the error above.

In the following two subsections we investigate the requirements imposed by the `Object` class specification on any user implemented class.

The Equals Method. Java's requirements for the `equals` method are listed in the documentation of `Object` [23], the root class for all classes in a Java software system.

> `boolean java.lang.Object.equals(Object obj)`
>
> Indicates whether some other object is "equal to" this one.
>
> The `equals` method implements an equivalence relation on non-null object references:
>
> – It is *reflexive*: for any non-null reference value `x`, `x.equals(x)` should return `true`.
> – It is *symmetric*: for any non-null reference values `x` and `y`, `x.equals(y)` should return `true` if and only if `y.equals(x)` returns `true`.
> – It is *transitive*: for any non-null reference values `x`, `y`, and `z`, if `x.equals(y)` returns `true` and `y.equals(z)` returns `true`, then `x.equals(z)` should return `true`.
> – It is *consistent*: for any non-null reference values `x` and `y`, multiple invocations of `x.equals(y)` consistently return `true` or consistently return `false`, provided no information used in equals comparisons on the objects is modified.
> – For any non-null reference value `x`, `x.equals(null)` should return `false`.
>
> The equals method for class Object implements the most discriminating possible equivalence relation on objects; that is, for any non-null reference values `x` and `y`, this method returns `true` if and only if `x` and `y` refer to the same object (`x == y` has the value `true`).
>
> Note that it is generally necessary to override the `hashCode` method whenever this method is overridden, so as to maintain the general contract for the `hashCode` method, which states that equal objects must have equal hash codes.
>
> **Parameters:** `obj` the reference object with which to compare.
> **Returns:** `true` if `this` object is the same as the `obj` argument; `false` otherwise.
> **See Also:** `hashCode()`, `java.util.HashMap`

These properties of the equals method are expected to hold for all classes. The first three properties are the standard properties defining equivalence relations, though there is a conditional here: the variables are not `null` pointers. The properties can be captured as parameterised JUnit tests. The following code is taken from JAxT [25].

```java
public static void equalsProperty1reflexive(Object x) {
  if (x != null)
    assertEquals(x, x);
}
public static void equalsProperty2symmetric
(Object x, Object y) {
  if (x != null && y != null)
    assertEquals(x.equals(y), y.equals(x));
}
public static void equalsProperty3transitive
  (Object x, Object y, Object z) {
  if (x != null && y != null && z != null)
    if (x.equals(y) && y.equals(z))
      assertEquals(x, z);
}
```

These axioms are close to the mathematical formulation, but utilise Java semantics as suggested in the Java standard library documentation. The fourth property is a bit more awkward. It is tightly related to the Java semantics and asks that no volatile or temporal or out-of-object (etc.) data is used in the `equals` algorithm. Though an axiom for the property can be written, e.g., a loop repeatedly testing two objects for "equality", running it will have minute chances of discovering any problems. The last property in the list is also related to Java semantics. It completes the right hand cases by providing comparisons to `null`. Note that a call `null.equals(x)` per Java semantics is meaningless and will throw `java.lang.NullPointerException` (if not already discovered and complained about by the compiler).

```java
public static void equalsProperty5null(Object x) {
  if (x != null)
    assertFalse(x.equals(null));
}
```

The JUnit `assertFalse` assertion checks that the argument is the boolean value `false`.

The HashCode Method. The other part of the motivating problem is to understand the `hashCode` method. The Java documentation for the `Object` class states the following [23].

> `int java.lang.Object.hashCode()`
>
> Returns a hash code value for the object. This method is supported for the benefit of hash tables such as those provided by `java.util.HashMap`.
>
> The general contract of hashCode is:
> – Whenever it is invoked on the same object more than once during an execution of a Java application, the `hashCode` method must

consistently return the same integer, provided no information used in `equals` comparisons on the object is modified. This integer need not remain consistent from one execution of an application to another execution of the same application.
- If two objects are equal according to the `equals(Object)` method, then calling the `hashCode` method on each of the two objects must produce the same integer result.
- It is *not* required that if two objects are unequal according to the `equals(Object)` method, then calling the `hashCode` method on each of the two objects must produce distinct integer results. However, the programmer should be aware that producing distinct integer results for unequal objects may improve the performance of hash tables.

Returns: a hash code value for this object.
See Also: `java.lang.Object.equals(java.lang.Object)`
`java.lang.System.identityHashCode`
Impl Spec: As far as is reasonably practical, the `hashCode` method defined by class `Object` returns distinct integers for distinct objects.

Only the second of these properties is meaningful to test for. This is also the crucial property that ensures equal objects will end up in the same hash bucket.

```
public static void hashCodeProperty2congruenceEquals
  (Object a, Object b) {
  if (a.equals(b))
    assertEquals(a.hashCode(), b.hashCode());
}
```

This axiom states that the `equals` method behaves as a congruence relation with respect to the `hashCode` method.

By capturing these axiomatic properties from the Java documentation as parameterised unit tests, we now have test oracles that will allow us to check the properties for an arbitrary amount of data.

On Writing Equals/hashCode Methods. To make our implementation of `Person` compatible with the Java language's standard library specification and allow us to use it with the Java collection classes, we need to provide an equivalence relation `equals` and a compatible `hashCode` method. Often case we can use an IDE like Eclipse [11] to generate these methods from the fields of the class.

There are two options for these methods. They can be restricted to compare objects belonging to the same class, or they can allow comparing objects within the subclass hierarchy being instances of this class.

Option 1: comparing only objects from the same class. Here an explicit check `getClass() != obj.getClass()` has been used to ensure that the two objects belong to the same class.

```
@Override
public boolean equals(Object obj) {
   if (this == obj)
     return true;
   if (obj == null)
     return false;
   if (getClass() != obj.getClass())
     return false;
  Person other = (Person) obj;
  return Objects.equals(name, other.name)
    && postcode == other.postcode;
}
@Override
public int hashCode() {
  return Objects.hash(name, postcode);
}
```

Note all the caveats in testing for special cases, partly due to the requirements imposed on the `equals` and `hashCode` methods, partly for optimisation purposes. Above we are using support methods `Objects.equals` and `Objects.hash`. These take care of the mentioned caveats for the fields, but equality comparison and computing hash codes is with case sensitive operations on `String`. Using case insensitive operations needs explicitly taking care of the technicalities with possibly `null` pointers in the fields.

Here two objects can compare equal only if they belong to the `Person` class. Thus if we also have subclasses of `Person`, e.g., a person with a birth place or with an age, an object of such classes will not compare equal to a `Person` object. However, unless explicitly overridden, the equals method above will be inherited by these subclasses. It will still be an equivalence relation for each such class, but it may be coarser than intended since the extra fields are not used in the equality algorithm.

Option 2: comparing objects within the subclass hierarchy. We might want people with the same name and address to compare equal, even though one may have a designated birth place, and the other may have a known age. One way of doing this is to compare objects belonging to classes rooted in `Person` by using `instanceof` rather than the technique above for demanding a specific class.

```
@Override
 final public boolean equals(Object obj) {
   if (this == obj)
     return true;
```

```
    if (obj == null)
      return false;
    if (!(obj instanceof Person))
      return false;
   Person other = (Person) obj;
   return Objects.equals(name, other.name)
     && postcode == other.postcode;
 }
 @Override
 final public int hashCode() {
   return Objects.hash(name, postcode);
 }
```

This equality comparison is fixed to the `Person` class (line 7 in the code snippet above) for all subclasses. Note that we have declared these methods as final. This is to prevent one of the subclasses reimplementing them, e.g., by taking into account additional fields. Any such modification of the `equals` method in a subclass would break symmetry or transitivity:

Proof. Assume we want to have a more fine grained equality between subclass objects. Symmetry breaks unless the subclass `equals` method takes great care in letting a subclass object be equal to a superclass object based only on the superclass `equals` algorithm. But then transitivity breaks, since now we can have two unequal (using fine grained equals) subclass objects `s1` and `s2` and a superclass object `x` such that `s1.equals(x) && x.equals(s2)`.

Similar arguments can be given to show that any modification a subclass could do to the `equals` method will break the method's requirements.

As an addendum to this observation, any modification of the `hashCode` method in a subclass would break the `hashCodeProperty2congruenceEquals` property.

On Creating Test Data for Object Axioms. The JUnit 5 approach to parameterised testing is to annotate the axiom class with a set of data for testing. This gives a fixed set of test data for an axiom, contrary to what we want to achieve with axiom based testing. We actually want to reuse the `Object` class test oracles (parameterised tests) for all classes that we develop, but for each class, or group of classes, we need a specific test data generator.

First observe that we actually want to test a specific set of `equals` algorithms: a new class and possibly all its subclasses (and maybe also some of its superclasses) that are supposed to interact with respect to equality. To activate the algorithms we want to test, we need to make certain the first argument to the test oracles belong to this set of classes.

As the second argument to `equalsProperty2symmetric` we want this set plus data for some irrelevant classes to make certain symmetry holds or fails symmetrically.

For the last two arguments of `equalsProperty3transitive` we want a large proportion of "equal data" from distinct objects. This indicates we want to create such data using different constructors, different expressions and different sequences of data. In the `Person` example it is sufficient to call the constructors with "equal" values, but for classes with interesting ways of building data creating good data is demanding.

In general, random test selection is as good as any, at least when providing a 10–15% increase in the test size [15]. Random testing also has the benefit it can span the data space and avoid a bias towards data sets that happen to pass the tests. Specifically random testing makes it impossible to adapt the algorithms under test to the test data. Unfortunately, completely random data will seldom generate data that is interesting for the equality axioms, since most of the random data will compare unequal. Some care can be used to skew random data generators towards a high proportion of "equal" data.

Also note that having some large number N of data points and reusing them for all the test arguments can have a profound effect on testing time. For two-argument oracles the testing time is squared $N * N$, for three-argument oracles it is cubed $N * N * N$, and for n-argument oracles the testing time becomes N^n. Thus, e.g., doubling N to increase test coverage using a random testing approach will increase the cost of testing the `equalsProperty3transitive` by a factor of 8.

3.2 Orderable Data – TreeMap

The `TreeMap` is a Java collection class that provides data sorted according to some chosen criterion. Such data can be traversed in order, as well as being stored and retrieved, similar to a `Map`. To achieve this the keys must be orderable, either directly by implementing the `Compareable` specification, or by having an explicit `Comparator` object. The `retrieveComparableAxiom` axiom below, expressed in Java, takes precedence over the `retrieveEqualsAxiom` from `Map` for every implementation of `SortedMap`.

```
public static <K extends Comparable<K>, V>
void retrieveComparableAxiom
  (SortedMap<K, V> map, K key, V value, K k2) {
  V v1 = map.get(k2);
  map.put(key, value);
  V v2 = map.get(k2);
  if (key.compareTo(k2) == 0)
    assertEquals(value, v2);
  else
    assertEquals(v1, v2);
}
```

This axiom is slightly simplified as a `SortedMap` may be using a `Comparator` instead of `Comparable`. See Fig. 1 to get an impression how difficult it can be to keep track of precedence properties.

The `java.lang.Compareable` specification [23] defines `k1.compareTo(k2)`, a method call which returns an integer. The object `k1` is considered smaller than `k2` if the returned integer is negative, they are equivalent if the returned integer is 0, and `k1` is larger than `k2` if the integer is positive. In Java terminology the `compareTo` method is called the class's *natural order*.

A `java.util.Comparator` object `x` defines `x.compare(k1,k2)`, a method call which returns an integer and orders objects `k1` and `k2` as above.

Both comparables and comparators are defined as total orders, which may cause conceptual problems when encoding a partial order. For instance, it may need to be more fine grained or coarse grained than the class's `equals` method.

Below is the axiom based test oracles for the comparable interface [23]. The axioms for `compareTo` from the interface `Comparable` are captured as follows.

```
public static <T extends Comparable<T>>
void compareToProperty1aDuality(T x, T y) {
  try {
    assertEquals
      (signum(x.compareTo(y)), -signum(y.compareTo(x)));
  } catch (RuntimeException re) {
    succeed("compareTo is allowed to throw.");
  }
}

public static <T extends Comparable<T>>
void compareToProperty1bNull(T e) {
  try {
    e.compareTo(null);
    fail(e + ".compareTo(null) should throw a NullPointerException");
  } catch (NullPointerException npe) {
    succeed("Throws NullPointerException as required");
  }
}

public static <T extends Comparable<T>>
void compareToProperty1cStrongSymmetry(T x, T y) {
  try {
    x.compareTo(y);
    y.compareTo(x);
    succeed("neither call  fails ");
    } catch (RuntimeException e) {
      // at least one of the calls throws an exception
      try {
        x.compareTo(y);
        fail("y.compareTo(x) throws while x.compareTo(y) does not");
      } catch (RuntimeException e1) {
        try {
          y.compareTo(x);
          fail("x.compareTo(y) throws while y.compareTo(x) does not");
        } catch (RuntimeException e2) {
          succeed("OK! Both calls fail symmetrically");
        }
      }
    }
  }

  public static <T extends Comparable<T>>
  void compareToProperty2primeTransitive(T x, T y, T z) {
    try {
      if (x.compareTo(y) >= 0 && y.compareTo(z) >= 0)
        assertTrue(x.compareTo(z) >= 0);
    } catch (RuntimeException re) {
      succeed("compareTo is allowed to throw.");
    }
  }
```

These parameterised unit tests also show a technique for capturing allowed exceptions so as to avoid them being reported as problems by the unit testing system. The assertion `success` is not part of JUnit, but is a NOOP introduced for documentation purposes.

The first axiom, duality, captures that swapping the order of comparison should swap the sign of the comparison. The second axiom, comparison with `null`, demands that a specific exception should be thrown when comparing with `null`. The third axiom, strong symmetry, demands that the comparator should consistently throw or not throw an exception when given the same pair of arguments in any order. The last axiom, transitivity, captures the transitivity requirement for the ordering operations. Note that while the first three axioms closely follow the Java standard library documentation, the last axiom has been formulated differently and, in consort with the other axioms, replaces two explicit requirements in the documentation.

The comparator defines an equivalence relation for `x.compareTo(y)==0`. The Java standard library strongly recommends that this equivalence is compatible with the equivalence defined by the `equals` method.

```
  public static <T extends Comparable<T>>
  void compareToProperty3anaturalOrderingEquals(T x, T y) {
    try {
      if (x.equals(y))
        assertEquals
          ("The natural ordering is inconsistent with equals.",
            0, x.compareTo(y));
    } catch (RuntimeException e) {
      succeed("some run-time exception occurred");
    }
  }

  public static <T extends Comparable<T>>
  void compareToProperty3bnaturalOrderingEquals(T x, T y) {
    try {
      if (x.compareTo(y) == 0)
        assertEquals
          ("The natural ordering is inconsistent with equals.", x, y);
    } catch (RuntimeException e) {
      succeed("some run-time exception occurred");
    }
  }
```

The first axiom requires that the `equals` equivalence relation is more fine grained than the `compareTo` operation. The second axiom requires that the `equals` equivalence relation is more coarse grained than the `compareTo` operation. Together they check the consistency between the two equivalence relations. By splitting the requirement into two axioms, any failed check will be clear on which granularity is failing the requirement. These axioms are optional, since they are not demanded by the library, only recommended.

The reason for a recommendation only, is that when partial orders are shoehorned into total orders, they are often made either more coarse grained or more fine grained to accomplish this.

The reason that consistency is strongly recommended, is an anomaly in Java's behavioural inheritance hierarchy. The definition of the `java.util.Map` interface is formulated using the `equals` method, but any orderable map is defined using the comparator for ordering. Thus the equivalence relation defined by the comparator is used, as written in the `java.util.SortedMap` interface documentation [23]:

> Note that the ordering maintained by a sorted map (whether or not an explicit comparator is provided) must be *consistent with equals* if the sorted map is to correctly implement the `Map` interface. (See the `Comparable` interface or `Comparator` interface for a precise definition of *consistent with equals*.) This is so because the `Map` interface is defined in terms of the `equals` operation, but a sorted map performs all key com-

> parisons using its `compareTo` (or `compare`) method, so two keys that are deemed equal by this method are, from the standpoint of the sorted map, equal. The behavior of a tree map *is* well-defined even if its ordering is inconsistent with equals; it just fails to obey the general contract of the `Map` interface.

All maps on orderable data, i.e., any class implementing the `SortedMap` interface, uses the equivalence relation `a.compareTo(b)==0` and not the equivalence relation `a.equals(b)`. This is in direct violation of the properties defined by `Map` which uses the equivalence relation `a.equals(b)`. Figure 1 illustrates how `HashMap` and `TreeMap` relate to the behavioural specifications `Map` and `SortedMap`. Following the path from `Map` to `TreeMap` a programmer might think that they both will store and retrieve the same data sets. This is the case if the two equivalence relations are consistent, but if they are not, subtle differences may occur when storing and retrieving data.

On Writing Comparison Operations. There are several caveats to consider when writing a comparator.

One issue is the symmetry of throwing specific exceptions. This turns out to be less of a special case, and fearlessly accessing expected components and calling comparators on the components generally takes care of the intended exceptions.

When comparing fields containing strings there are options to choose comparators that ignore case. If you want to keep the natural ordering consistent with `equals`, be certain to choose the same style of string comparison for both comparators and equality.

Since comparators return integers indicating the ordering, it may be tempting to use subtraction when comparing `int` fields. This is generally a bad idea, since if the subtraction yields `Integer.MIN_VALUE` the duality rule breaks. The reason is that `Integer.MIN_VALUE == -Integer.MIN_VALUE` in Java's semantics for `int`. On the other hand, in our `Person` example where the post code is encoded as `short`, we can safely use subtraction to find the ordering. The reason is that the range of `short` is too small for the subtraction to reach that limit, and the subtraction returns the result as `int` which avoids any problems with representability of the difference. For more details on this see Sect. 4.

In the same way as for the `equals` method, there are two options regarding the interaction with the subclass hierarchy.

Option 1: comparing only objects from the same class. This requires the same trick as for the `equals` method: a check for `getClass() != obj.getClass()` needs to be included in the code.

Option 2: comparing objects within the subclass hierarchy. Since the `Comparable` is generically typed, the default is that subclasses are allowed in the comparison. Be certain to mark the `compareTo` method as **final** to avoid problems with overriding implementations.

Note. The same option should be chosen both for the comparator and for the `equals` method for a class. Otherwise the software design will induce a lack of consistency between the two equivalence relations.

3.3 Discussion

In this section we have shown how axiom based testing can be used to understand the Java standard libraries. Much of the documentation is easy to translate from its algebraic style of English to parameterised test oracles. The library is mostly organised according to behavioural subtyping, which allows axioms to follow the inheritance hierarchy, both for classes and for interfaces. There is one notable exception from this, namely that the `SortedMap` specification takes precedence over the `Map` specification. The two specifications use two different equivalence relations so being aware of this anomaly is very important.

The axiom based tests also make the requirements on `equals` / `hashCode` and `Comparable` / `Comparator` methods testable. These are methods most programmers need to implement for the classes they are developing. Getting these methods right is central to the use of the collection classes. With axiom based testing we have a set of tests which can be reused for every implementation.

Further analysis reveals that there are two approaches to implementing these methods. Option 1 is making certain they can be used only between objects of the same class. Option 2 is making them into **final** methods that compare objects for the entire subclass hierarchy.

As illustrated by Fig. 1, traversing the inheritance hierarchy to find all appropriate axioms by hand can be challenging. A tool like JAxT [25] can trace the inheritance hierarchies and identify all relevant inherited axioms.

Hopefully seeing and studying these axioms empowers the student in writing their own axiom based tests when developing code. Axiom based software development is similar to test based development [2], but has the benefit that the test oracles capture the intention across the entire range of data, not just the few data points embedded in a typical unit test.

4 Computer Integers

When students arrive at university to study computer science/software engineering, they have about twelve years of training (primary school through high school) in *natural numbers* $\mathbb{N}$ (non-negative integers) and *mathematical integers* $\mathbb{Z}$. Some students may even have some programming experience using a dynamic programming language with arbitrary precision integers. This background intuition is not compatible with *computer integers* as represented in hardware or in major languages like Java and C++. In addition, the way we teach programming often strengthens the students' misconception of computer integers. Our examples use small numbers and thus never hit the boundary where the difference between mathematical and computer integers are manifest. And when problems could occur, we recommend increasing the precision of the integers to circumvent trouble rather than discussing the issue.

In this section we try to unravel the students' knowledge and intuition of integers, and confront this with the actual behaviour of the computer integers. We will do so using our tool of axiom based testing: gently formalising the students' math training as axioms, then hit those axioms with data that breaks the perceived communality between mathematical integers and computer integers.

4.1 Ring Properties

Discovering the *commutative ring* properties should be easy for students, as most of them have been trained from early school years in the rules for computing with numbers. A few examples should be sufficient to identify the axioms.

Example	Rule	Property name
$5+7=7+5$	$x+y=y+x$	commutativity
$5*7=7*5$	$x*y=y*x$	
$(5+7)+9=5+(7+9)$	$(x+y)+z=x+(y+z)$	associativity
$(5*7)*9=5*(7*9)$	$(x*y)*z=x*(y*z)$	
$38+0=38$	$x+0=x$	neutral element
$1*38=38=38*1$	$1*x=x,\quad x*1=x$	
$5*(7+9)=5*7+5*9$	$x*(y+z)=x*y+x*z$	distributivity
$(7+9)*5=7*5+9*5$	$(y+z)*x=y*x+z*x$	
$(7+5)-7=5$	$(x+y)-x=y$	subtraction
$57*0=0=0*57$	$x*0=0,\quad 0*x=0$	annihilation

All these rules are equational, and easy to formulate as parameterised test oracles. Without the commutativity rule for $*$ we define a *ring*. With the commutativity rule for $*$ the second forms of the neutral element for $*$, the distributivity and the annihilation rules are redundant and can be omitted from testing. The subtraction rule above is closer to what we learn in primary school math than the unary minus operation (*additive inverse*) normally used when specifying rings. The subtraction rule, with certain limitations, also work for natural numbers (non-negative integers) $\mathbb{N}$. Since natural numbers lack proper subtraction they are not proper (commutative) rings. The annihilation rule is redundant as a ring property. However, if we remove subtraction from the operations we consider, but keep the other operations and properties, including the annihilation rule (which is no longer redundant), we get a *commutative semiring.* Further dropping commutativity for $*$ we get a *semiring.* Obviously all (commutative) rings are (commutative) semirings. The set of natural numbers forms a commutative semiring.

No Zero Divisors. For the mathematical integers (and natural numbers) there is also the *no zero divisors* rule: the product of two non-zero numbers is non-zero. This can be formalised as $x * y = 0 \Rightarrow (x = 0 \mid\mid y = 0)$. Though closely related to the ring operations $(+, *, -)$ and constants $(0, 1)$, this property does not hold for all rings.

Sign Change. Another expectation is that mathematical integers change sign with the unary minus operator: $x == -x \Rightarrow x == 0$. This does not hold for rings in general. For ordered rings, see Sect. 4.3 below, this property is redundant.

4.2 Total Order Properties

It is also relatively easy to guide students to discover the total order properties of the integers.

Example	Rule	Property name
$12 \leq 12$	$x \leq x$	reflexivity
$8 \leq 2+6$ && $2+6 \leq 8 \Rightarrow$ $2+6=8$	$x \leq y$ && $y \leq x \Rightarrow x = y$	antisymmetry
$23 \leq 33$ && $33 \leq 54 => 23 \leq 54$	$x \leq y$ && $y \leq z \Rightarrow x \leq z$	transitivity
$24 \leq 14 \mid\mid 14 \leq 24$	$x \leq y \mid\mid y \leq x$	connexity

Note that connexity implies reflexivity, so reflexivity can be omitted from testing.

4.3 Ordered Ring

An *ordered ring* has ring properties, order properties, and two axioms explaining how ring operations $(+, *, -)$ are to interact with the order operation $(\leq)$ and constants $(0, 1)$.

Example	Rule	Property name
$10 \leq 13 \Rightarrow$ $10+43 \leq 13+43$	$x \leq y \Rightarrow x + z \leq y + z$	ordered plus
$10 \leq 13$ && $0 \leq 43 \Rightarrow$ $10 * 43 \leq 13 * 43$	$x \leq y$ && $0 \leq z \Rightarrow x * z \leq y * z$	ordered multiply

These two properties are more dim for students, though they can easily recognise and accept them as part of the mathematical numbers. The tacitness of these properties should be a major concern when switching from mathematical to computer integers. Worryingly this seems mostly ignored by the programming literature though it is a significant concern for software vulnerabilities. Even more troublesome it also seems mostly ignored by the formal methods community.

Using examples like $0 \leq 67 \Leftrightarrow -67 \leq 0$ it is possible to present a third more familiar property, $0 \leq x \Leftrightarrow -x \leq 0$. The rule is a simple consequence of the ordered plus property. It implies the *sign change* rule above.

4.4 Integer Types and Test Data

The computer integers are several distinct types. Java has four ranges of integers (subsets of $\mathbb{Z}$) [14] in two's complement representation.

- `byte` 8-bit, range $-128 = -2^7$ through $127 = 2^7 - 1$.
- `short` 16-bit, range $-32\,768 = -2^{15}$ through $32\,767 = 2^{15} - 1$.
- `int` 32-bit, range $-2\,147\,483\,648 = -2^{31}$ through $2\,147\,483\,647 = 2^{31} - 1$.
- `long` 64-bit, range $-9\,223\,372\,036\,854\,775\,808 = -2^{63}$ through $9\,223\,372\,036\,854\,775\,807 = 2^{63} - 1$.

This is the standard representation of computer integers on current commodity computing devices. Older programming languages, like C and C++, used to allow more flexibility in ranges and representation of integer types, but C++ currently is also intending to standardise on two's complement representation. In addition C and C++ support unsigned integers, corresponding to ranges of natural numbers (subsets of $\mathbb{N}$):

- `unsigned char` 8-bit, range 0 through $255 = 2^8 - 1$.
- `unsigned short` 16-bit, range 0 through $65\,535 = 2^{16} - 1$.
- `unsigned int` 32-bit, range 0 through $4\,294\,967\,295 = 2^{32} - 1$.
- `unsigned long` 64-bit, range 0 through $18\,446\,744\,073\,709\,551\,615 = 2^{64} - 1$.

From a mathematical perspective, the unsigned integers form a commutative ring with modulus computations (2^k for $k = 8, 16, 32, 64$). In hardware all operations $+, *, -$ and $0, 1$ are the same for signed and unsigned integers. The difference shows in the interpretation of the bit patterns when comparing (total orders) and I/O. In Java, unsigned integers are treated as signed integers, but the program calls special functions for comparing and conversion to and from strings. Below we illustrate this in Java for the ordered plus axiom.

```
public static void orderedPlus(int x, int y, int z) {
 if (x <= y)
    assertTrue(x + z <= y + z);
}
public static void orderedPlusUnsigned(int x, int y, int z) {
 if (Integer.compareUnsigned(x, y) <= 0)
    assertTrue(Integer.compareUnsigned(x + z, y + z) <= 0);
}
```

The computer integer properties are well suited for random testing. Spelling out the axioms for type `int` and using the `Random` class's `nextInt()` method is a good setup for testing 32-bit integers. A few hundred test cases will pass the commutative ring and total order axioms,[1] but both ring order axioms will fail.

[1] There is of course a problem with the conditional axioms like antisymmetry, where few, if any, of the test values actually will reach the target of the conditional. Quickcheck [6] has special treatment of conditionals in order to generate random data that actually reaches the target.

The benefit of using an appropriate random number generator is that it will span the whole domain of the computer integer under test. Values selected manually tend to be from a small range since small numbers are more easy to comprehend by the tester. Such a bias will prevent discovering that the ring order axioms fail to hold.

When testing the other computer integers in Java (and C or C++), there is need for caution. For instance, due to widening (see Sect. 4.5), when writing axioms for 16-bit integers, care has to be taken to make certain computed values are treated as `short`. Thus explicit casting needs to be introduced as in the Java example test oracles below.

```java
public static void orderedPlus(short x, short y, short z) {
  if (x <= y)
    assertTrue((short) (x + z) <= (short) (y + z));
}
public static void orderedPlusUnsigned
  (short x, short y, short z) {
  if (Short.compareUnsigned(x, y) <= 0)
    assertTrue(
      Short.compareUnsigned((short) (x + z),
                            (short) (y + z)) <= 0);
}
```

Without the casts the code will compute with 32-bit integers. Using 16-bit values embedded in a 32-bit range will not invalidate the ring order axioms.

When creating test values for 64-bit integers designating the types as `long` does not incur problems as all computations then will take place using 64-bit. However, it is important to use a random generator for `long`, e.g., `Random`'s `nextLong()`.

```java
@Test
 final void testOrderedAdd() {
   for (int i = 0; i < n; ++i) {
     long x = r.nextLong();
     long y = r.nextLong();
     long z = r.nextLong();
     OrderLong.orderedPlus(x, y, z);
   }
}
```

If using `nextInt()` the test will be limited to 32-bit integers, which when silently embedded in 64-bit integers will not invalidate the ring order axioms.

A drawback with testing `int` and `long` is that the violating values will be so large that they are difficult to read and comprehend. The shear number of digits involved means that they defy manual validation. This hinders a deeper understanding of why the axioms are violated. Testing the axioms on a smaller range, like `short`, keeps numbers manageable. Using 16-bit integers also permits complete testing for all axioms with 1 or 2 arguments. Complete testing of a 3 argument axiom (such as associativity, distributivity and transitivity) using 16-bit integers will take several days on a modern computer. A hybrid solution is an alternative: complete coverage of the first 2 arguments combined with random selection of the third. The test code below shows this approach for the ordered plus axiom on `short` in Java.

```
@Test
final void testCompletishOrderedAdd() {
  for (int x = Short.MIN_VALUE; x <= Short.MAX_VALUE; ++x)
    for (int y = Short.MIN_VALUE; y <= Short.MAX_VALUE; ++y) {
      short z = (short) r.nextInt();
      OrderShort.orderedPlus((short) x, (short) y, z);
    }
}
```

Note the use of `int` in the loops to be certain we cover the entire range for `short`. This will take a few minutes per successful 3 argument axiom. Typically much shorter time for the failing axioms, as the one tested above.

Using complete testing of the *no zero divisors* axiom will reveal that it does not hold for computer integers. For this axiom, random testing will most probably not uncover the problem. The ratio of failing number pairs is about 10^{-2} for 8-bit integers, about 10^{-4} for 16-bit integers, about $3 * 10^{-9}$ for 32-bit integers (the most used computer integer type), and about $2 * 10^{-18}$ for 64-bit integers. For the larger ranges it is thus practically impossible to randomly discover any failing pair of values. Complete testing of 2 argument axioms is out of hand for any integer type larger than 16-bit. The number of complete test cases grows by a factor of about $4 * 10^9$ going from 16-bit to 32-bit, and by about $2 * 10^{19}$ going from 32-bit to 64-bit. Only crafting failing values will reveal that computer integers have zero divisors, and being able to craft such values requires a proper understanding of the problem.

The *sign change* rule is also interesting from this perspective. Since computer integers are not ordered rings, this property is not redundant. It is violated by exactly one value per computer integer in two's complement representation. Since this is a 1 argument axiom, it can be tested completely for up to 32-bit integers in at most a few minutes. The range for `long` is just too large to stumble upon the result in reasonable time, unless the "culprit" is identified and manually injected early into the test set. Though, for this axiom, and in signed two's complement representation, the culprit will always be the minimum number represented in its range. For unsigned integers it will be the unsigned interpretation of that bit pattern. A naïve complete test that starts at the minimum signed number will actually find the culprit effortlessly (sic). However, as mentioned in Sect. 3.2, understanding this property properly is important for the correct implementation of comparators.

4.5 Narrowing and Widening Computer Integers

In the programming literature, especially related to Java or C++, integer *widening* is transforming an integer from a type with fewer bits to a type with more bits. Widening is also called *promotion* since the value is "promoted" to a larger bit range, ensuring it can be represented. In the common two's complement representation this amounts to replicating the sign bit (when relevant) into the newly assigned bits. Widening often occurs automatically, e.g., these languages typically carry out all integer computations in 32-bit or 64-bit, and widen any type with fewer bits to the nearest enclosing size. A danger is that students will think widening is always good since it automatically removes some problems from dealing with the smaller integer sizes.

The opposite, going from many bits to fewer bits, is called *narrowing*, and requires explicit casting in Java and C++, i.e., the code must explicitly request conversion to the intended type. Narrowing just truncates the higher end bits in the two's complement representation. This may change the sign of the truncated value, depending on which bit of the value that happens to be in position of the sign bit. For unsigned numbers narrowing corresponds to a modulus operation, i.e., going from n bits to k bits ($n > k$) is computing modulus 2^k.

Promotion of expressions is the reason the first two loops below are infinite since `++i` will wrap around (as `byte` or `short`, respectively) while `i+1` is promoted to `int` and does not wrap around. The latter two loops are finite due to the wraparound that occurs when `i+1` exceeds the bounds for its type.

```
for (byte i=0; i < i+1; ++i) {}
for (short i=0; i < i+1; ++i) {}
for (int i=0; i < i+1; ++i) {}
for (long i=0; i < i+1; ++i) {}
```

The following loops are all finite, since the `++i` operation returns the wrapped value (for comparison) before `i` is updated. However, the loop body sees the updated value, hence it will not see the iteration for `i==0`.

```
for (byte i=0; i < ++i; ) {}
for (short i=0; i < ++i; ) {}
for (int i=0; i < ++i; ) {}
for (long i=0; i < ++i; ) {}
```

In both groups the loop iterating over `long` will take too long to compute to notice that the loops will terminate.

If we look at the computer integers as algebraic structures, we can relate widening and narrowing to the formal concept of homomorphisms. Informally, a homomorphism maps between two models for a specification (same selection of operations, constants and axioms), such that the operations of the algebra behave consistently across the mapping. We call a homomorphism an embedding if the mapping is injective. Homomorphisms compose.

The computer integers satisfy the commutative ring specification (operations $+, *, -$ and constants $0, 1$) and the total order specification (operation $\leq$ including the constants $0, 1$), but they have zero divisors and are not ordered rings.

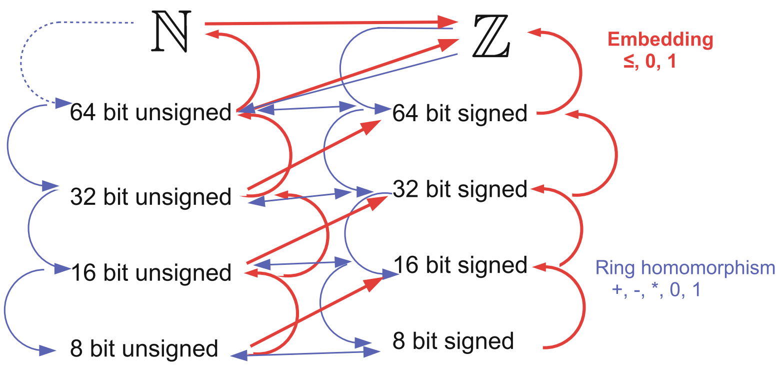


Fig. 2. Illustrating the homomorphic (ring related operations) and embedding (total order related operations) relationships between natural numbers $\mathbb{N}$ and mathematical integers $\mathbb{Z}$ and various common computer integer representations.

Widening can be seen as embedding the smaller ranges into the larger ranges. Embedding preserves the total order properties, but not the ring properties. The order embeddings are illustrated with the thick red arrows in Fig. 2. This includes the embedding into the natural numbers $\mathbb{N}$ for the unsigned integers, and into the mathematical integers $\mathbb{Z}$ for all computer integers.

The ring homomorphisms give a rather interesting picture, the thin blue arrows in Fig. 2. They are bijections between the signed/unsigned integers with the same number of bits. There are also ring homomorphisms from the wider ranges to the narrower ranges that preserve the ring computations: computing a value then narrowing it has the same results as first narrowing the values, then computing in the narrower range. These homomorphisms do not preserve ordering. For instance, the 32-bit integers $65535 < 65536$, but truncating to 16-bits we get the numbers 65535 and 0, respectively, thus the ordering is not preserved. In Fig. 2 there is also a mapping from the mathematical integers $\mathbb{Z}$ to the computer integers. The mapping from the natural numbers $\mathbb{N}$ to the unsigned numbers is "dotted" since the subtraction operation does not really exist for $\mathbb{N}$, which is thus only a semiring, not a ring as the unsigned computer integers.

The seemingly unproblematic widening and narrowing between various modulus integers and computer integers can leave the students with the intuition that modulus and computations using computer integers are interchangeable. This is an artefact of the two's complement integers all being signed and unsigned "modulus 2^k" numbers. It only holds when the narrow "modulus" is a factor of the wide "modulus". A useful challenge is to consider what happens when trying to compute modulus 7 operations inside the wider range of modulus 13 operations.

The wraparound effect taking place in the modulus 13 ring computations causes problems for the modulus 7 operation we attempted to implement.

e	(e mod 7)	(e mod 13) mod 7	(e mod 7644) mod 7
$6 + 6$	5	5	5
$2 * 5$	3	3	3
$4 * 5$	6	0	6
$3 - 1$	2	2	2
$1 - 3$	5	4	5

The problems with "(e mod 13) mod 7" relates to a lack of ring homomorphisms from modulus 13 integers to modulus 7 integers. Modulus 7 within modulus 7644 works fine since $7644 = 2 * 2 * 3 * 7 * 7 * 13$, i.e., there is a ring homomorphism from modulus 7644 integers to modulus 7 integers, the homomorphism being $x \mapsto (x \bmod 7)$.

4.6 Discussion

Here we have expanded how a student's perception of the mathematical integers gives a misconception of the computer integers. This misconception is reinforced when we only use unproblematically small integers in our examples.

We have argued that making the student's expectations of integer behaviour explicit as axiom based tests, allows the student to generate test cases and see for themselves that some expectations are broken for the computer integers. Some of the test cases must be hand crafted since they are rare among the humongous sizes of the wider computer integers (32-bit and 64-bit). To better understand the issues at hand, computing with 16-bit integers (`short`) is illustrative. Due to the promotion rules between integer computations in many programming languages, great care has to be taken when writing axioms for the `short` data type.

These misconceptions are important for both the correctness and the vulnerability of code. Even the rare "culprit" data values are often used in attacks against computer systems. Triggering a strange behaviour in an innocent game may be sufficient to cause denial of service for other applications.

Some common examples of problems related to overflow and modulus problems.

- Overflow in binary search algorithms can cause programs to crash in Java or access of arbitrary memory in C [5]. This was first noticed in the mid 2000s when computer memory started to exceed 2 GB, allowing array indices to span the entire positive 32-bit integer range.
- Buffer overwrites (and arbitrary changes to code) may occur due to miscalculation of buffer sizes. For instance, a request for 24642 buffers each of size 194651 bytes requires 4796589942 bytes (4.8 GB) of buffer space, but due to 32-bit wrapping an `int` calculation will end up requesting 501622646 bytes (0.5 GB). The incoming data stream can then overwrite other parts of computer memory, parts that are later interpreted as instructions. It is difficult to

spot that the request should have 10 digits but the allocated size is 9 digits, unless one is on the outlook for such problems.

These days a few minutes of digital video easily exceeds 2 GB, so arrays beyond 2 GB are not exceptional.

Another issue to consider is that many formal tools support mathematical integers and computer integers, but fail to alert the user of what is lost in the transition from mathematical integers to computer integers [3]. This is especially a problem for the ring order axioms. The tool may be happy verifying at the mathematical integer level that the computation of buffer size it correct, and it may verify at the computer integer side that the computation of buffer size is correct according to the wraparound semantics. But without the tool detecting that the transition from mathematics to computers breaks the allocation size assumption, the program is still wrong and harbours a security threat.

Wraparound semantics is one alternative when going from mathematical integers to computer integers. Hoare in [17] mentions two other approaches, the *strict interpretation* and the *firm boundary* (commonly known as *saturation integers*). The alternatives trade off other properties than the wraparound integers do. For instance, the strict interpretation approach makes certain computations, like adding two large numbers, illegal, possibly terminating a program. The saturation integers may trade off associativity and others in order to keep the ordered plus and ordered multiply properties.

This leaves us with some final thoughts.

– There are multiple choices in what integer properties to sacrifice when deciding on a computer integer system.
– It will be fun for students to write down the relevant axiom based tests and figure out which tradeoffs each such representation has made.
– Different choices will leave programmers with different blind spots unless they are specifically trained to be aware of them.
– Even though an axiom only fails for a minute fraction of the possible values, it is as serious as any vulnerability.

And what stories about vulnerabilities can be conjunctured for alternative integer representations?

5 Floating Point Numbers

From school we know that we have learnt the inclusions from the naturals to the mathematical integers to the rationals to the reals $\mathbb{N} \subseteq \mathbb{Z} \subseteq \mathbb{Q} \subseteq \mathbb{R}$. This is also the typical sequence these number systems were introduced during our school years up through high school. With our formal concept of homomorphism, we can see that in this mathematical world these inclusions are commutative (semi)ring, total order, ordered ring, no zero divisor, and sign change embeddings. As such a beautiful picture.

In the computer setting this breaks down. We need to compromise on which properties to retain for the unsigned/signed computer integers. Computer rationals do not make practical sense. They will be represented as fractions, by a pair of computer integers, and after a few additions of fractions we often hit the range limits of the underlying computer integers. Reals are not representable on computers, and are approximated with floating point numbers.

Computer scientists, especially the theoretically oriented ones, shy away from floating point as the data types are unruly and difficult to analyse.

In some ways floating point is a saturation number system with $-\infty$ and ∞ as the boundaries. From a formalistic viewpoint it is problematic that $0 = -0$ but $\frac{1}{0} = \infty \neq -\infty = \frac{1}{-0}$, but this is well motivated from a pragmatic viewpoint. Floating point systems also include a large range of *not a number* (NaN) which is used to encode some errors, e.g., $\infty + (-\infty)$ or $\frac{0}{0}$.

One idea could be to familiarise students with floating point properties. For instance, they could write axiom based tests based on the properties of the real numbers. The task should not be to figure out which laws fail, as we normally do. Instead we could try to make statistics on how well each law holds up (its ratio of success). There are many parameters to vary: the precision of the numbers, the distribution of test data, etc.

Hopefully such ideas can make numerical computations less of an anomaly in computer science and software engineering. The engineers and scientists working with numerical programs are in dire need of more software expertise. They are specifically looking for knowhow in formal methods, see the report [13] and the followup workshop series CORRECTNESS [7]. And exciting things are happening in the floating point domain as well. New number representations are being investigated in aim of reproducible results for numerical computing. This domain harbours many interesting challenges for software science students.

6 Summary and Conclusion

Throughout this paper we have presented Java axiom based tests (parameterised JUnit 5 tests [24]) showing how to capture precisely

- specifications of types & classes (computer integers in detail, and sketches for `HashMap`, `TreeMap` and floating point),
- requirements for the use of classes (`equals`, `hashCode` and `Comparable` in full detail with two specific implementation options that need to be chosen consistently, and implications for `Comparator`),
- behavioural subtyping in Java's class hierarchy (generally inheritance of axioms, and the `Map` — `SortedMap` anomaly)

Guidelines on how to create data for these parameterised JUnit tests have been presented, both to discover general issues and to be aware of rare special cases. The examples expose the many technicalities involved in programming and are coupled to discussions of some related vulnerabilities. They provide reusable tests that can be used both when developing own classes and also for exploring libraries for conformance.[2] More examples are available in [25].

The discussion of computer integers can alert students and practitioners alike to problems that occur by tacitly transferring assumptions about mathematical numbers to programming. This mismatch also seems insufficiently handled by the formal methods community [3].

The examples in this paper demonstrate that axiom based tests are simple to code. Writing parameterised unit tests is on par with writing regular unit tests. This is undoubtedly accessible for students with beginning knowledge of programming. Experience with teaching Quickcheck indicates axiom based testing is better taught after the first semester [20]. Clearly axiom based testing is a toolset that belongs in an undergraduate degree. The tests are straight forward to program using boolean expressions, as there in practice is no need for explicit quantifiers or other constructs that require a separate specification language. Developing and using axiom based tests require no training in formal methods. Teaching axiom based testing early can be a good stepping stone for a follow up formal methods course.

References

1. Bagge, A.H., David, V., Haveraaen, M.: Testing with axioms in C++ 2011. J. Object Technol. **10**, 10:1–10:32 (2011). https://doi.org/10.5381/jot.2011.10.1.a10
2. Beck, K.: Extreme programming: a humanistic discipline of software development. In: Astesiano, E. (ed.) FASE 1998. LNCS, vol. 1382, pp. 1–6. Springer, Heidelberg (1998). https://doi.org/10.1007/BFb0053579
3. Beckert, B., Schlager, S.: Refinement and retrenchment for programming language data types. Formal Asp. Comput. **17**(4), 423–442 (2005). https://doi.org/10.1007/s00165-005-0073-x
4. Bergstra, J.A., Tucker, J.: Algebraic specifications of computable and semi-computable data types. Theor. Comput. Sci. **50**, 137–181 (1987). https://doi.org/10.1016/0304-3975(87)90123-X
5. Bloch, J.: Extra, extra - read all about it: Nearly all binary searches and merge sorts are broken. https://ai.googleblog.com/2006/06/extra-extra-read-all-about-it-nearly.html. Accessed 23 July 2020
6. Claessen, K., Hughes, J.: QuickCheck: a lightweight tool for random testing of Haskell programs. In: ICFP 2000: Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming, pp. 268–279. ACM Press, New York, NY, USA (2000). https://doi.org/10.1145/351240.351266
7. Correctness 2020: Fourth International Workshop on Software Correctness for HPC Applications. https://correctness-workshop.github.io/2020/. Accessed 08 Apr 2020
8. Dahl, O.J., Myhrhaug, B., Nygaard, K.: SIMULA 67 Common Base Language, Vol. S-2. Norwegian Computing Center, Oslo (1968)
9. David, V.: Catsfoot (2011). https://catsfoot.sourceforge.net/
10. Dijkstra, E.W.: Guarded commands, non determinacy and formal derivation of programs. Commun. ACM **18**(8), 453–457 (1975). https://doi.org/10.1145/360933.360975

[2] The end of [19] shows additional benefits.

11. Eclipse. https://www.eclipse.org. Accessed 22 June 2020
12. Gannon, J.D., McMullin, P.R., Hamlet, R.G.: Data-abstraction implementation, specification, and testing. ACM Trans. Program. Lang. Syst. **3**(3), 211–223 (1981). https://doi.org/10.1145/357139.357140
13. Gopalakrishnan, G., et al.: Report of the HPC correctness summit, Jan 25–26, 2017, Washington, DC. CoRR abs/1705.07478 (2017). https://arxiv.org/abs/1705.07478
14. Gosling, J., Joy, B., Steele, G., Bracha, G., Buckley, A., Smith, D., Bierman, G.: The Java language specification - Java SE 14 Edition. Technical report. JSR-389 Java SE 14, Oracle America (Feb 2020). https://docs.oracle.com/javase/specs/
15. Hamlet, R.: Random testing. In: Marciniak, J. (ed.) Encyclopedia of Software Engineering, pp. 970–978. Wiley, Hoboken (1994). https://doi.org/10.1002/0471028959.sof268
16. Haveraaen, M., Kalleberg, K.T.: JAxT and JDI: the simplicity of JUnit applied to axioms and data invariants. In: OOPSLA Companion 2008: Companion to the 23rd ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications, pp. 731–732. ACM, New York, NY, USA (2008). https://doi.org/10.1145/1449814.1449834
17. Hoare, C.A.R.: An axiomatic basis for computer programming. Commun. ACM **12**(10), 576–583 (1969). https://doi.org/10.1145/363235.363259
18. Hoare, C.A.R., Wirth, N.: An axiomatic definition of the programming language PASCAL. Acta Inf. **2**, 335–355 (1973). https://doi.org/10.1007/BF00289504
19. Hughes, J.: Quickcheck testing for fun and profit. In: Hanus, M. (ed.) PADL 2007. LNCS, vol. 4354, pp. 1–32. Springer, Heidelberg (2006). https://doi.org/10.1007/978-3-540-69611-7_1
20. Hughes, J.: Experiences from teaching functional programming at Chalmers. ACM SIGPLAN Notices **43**(11), 77–80 (2008). https://doi.org/10.1145/1480828.1480845
21. ISO/IEC 14882:2017 - Programming languages - C++ (2017). https://www.iso.org/standard/68564.html
22. ISO/IEC 9899:2018 - Information technology - Programming languages - C (2018). https://www.iso.org/standard/74528.html
23. Java platform, standard edition & Java development kit - version 14 API specification. https://docs.oracle.com/en/java/javase/14/docs/api/java.base/module-summary.html. Accessed 29 June 2020
24. Junit 5. https://junit.org/junit5/. Accessed 24 June 2020
25. Kalleberg, K.T., Haveraaen, M.: JAxT - Java Axiomatic Testing. https://www.ii.uib.no/mouldable/testing/. Accessed 24 June 2020
26. Liskov, B.: Keynote address - data abstraction and hierarchy. In: Addendum to the Proceedings on Object-Oriented Programming Systems, Languages and Applications (Addendum), pp. 17–34, OOPSLA 1987. Association for Computing Machinery, New York, NY, USA (1987). https://doi.org/10.1145/62138.62141
27. Meyer, B.: Applying "Design by contract". Computer **25**(10), 40–51 (1992). https://doi.org/10.1109/2.161279
28. Saff, D.: Theory-infected: or how I learned to stop worrying and love universal quantification. In: OOPSLA 2007: Companion to the 22nd ACM SIGPLAN Conference on Object Oriented Programming Systems and Applications Companion, pp. 846–847. ACM, New York, NY, USA (2007). https://doi.org/10.1145/1297846.1297919
29. Sannella, D., Tarlecki, A.: Mind the gap! Abstract versus concrete models of specifications. In: Penczek, W., Szałas, A. (eds.) MFCS 1996. LNCS, vol. 1113, pp. 114–134. Springer, Heidelberg (1996). https://doi.org/10.1007/3-540-61550-4_143

Teaching Formal Methods for Fun Using Maude

Peter Csaba Ölveczky(✉)

University of Oslo, Oslo, Norway
peterol@ifi.uio.no

Abstract. In this paper I try to identify some general criteria for teaching an undergraduate formal methods course in a "fun" way. Based on those criteria, I have developed an introductory formal methods course using rewriting logic and Maude. I explain why Maude is a suitable formal method for such a course, give an overview of the course and its textbook, and summarize student feedback to the course.

1 Introduction

These days present a great opportunity for integrating formal methods into mainstream software development, beyond their traditional role of verifying safety-critical systems, for a number of reasons:

– The software industry is realizing that standard industrial validation techniques are insufficient and do not scale up to today's systems.
– The "winner-takes-all" nature of the software industry justifies an up-front investment into making the systems as reliable and efficient as possible.
– Society is increasingly reliant also on "non-safety-critical" systems.
– Success stories are emerging on the use of formal methods in standard software development, including from Amazon Web Services, the most profitable part of one the world's most valuable brands.

To take advantage of this opportunity and achieve the goal of making formal methods an integral part of mainstream software development, we need to educate students who have some knowledge of formal methods, and appreciation that they can add value to industrial software development.

However, there are many challenges to make students study and appreciate formal methods that I discuss in Sect. 2: students may not have heard of formal methods, and if they have heard of them they may not consider them relevant for the job market; students may have limited mathematical background; and our colleagues may manage to keep formal methods far away from mainstream course programs. This easily leads to a vicious circle, where few formal methods people in industry leads to limited use and appreciation of them, so that prospective students do not see the point of studying formal methods, and so on.

To break out of this vicious circle, the organizers of the FMfun workshop argue that formal methods teaching should be fun. But how should we teach

A. Cerone and M. Roggenbach (Eds.): FMFun 2019, CCIS 1301, pp. 58–91, 2021.
https://doi.org/10.1007/978-3-030-71374-4_3

formal methods in a fun way? I try to identify what a computer science student thinks is fun in Sect. 3. I use this knowledge in Sect. 4 to identify some general criteria for what an introductory undergraduate course in formal methods should look like.

Based on those criteria, and that as an undergraduate student eons ago I thought that functional programming was the most "fun," I have developed, taught, and written a textbook [42] for a second-year introductory course on formal methods using rewriting logic [28] and its simulation and model checking tool Maude [13]. I give an overview of the course and its textbook in Sect. 5. The course emphasizes the formal modeling and model checking analysis of cornerstone distributed algorithms in today's systems, including transport protocols, distributed algorithms, and cryptographic protocols. Maude should be very well-suited for this, since it combines a powerful object-based and functional-programming (style of) modeling with automatic model checking. In particular, Maude's simple yet expressive and general formalism makes it easy to formalize textbook distributed systems in different domains, as I illustrate in Sect. 5.5.

Is the course fun and seen as relevant? Section 6 summarizes student feedback from the last 10+ years. What is promising is that twice as many students finished the course this year compared to last year.

2 Making Students Study Formal Methods: Challenges

This section discusses challenges involved in making undergraduate students take introductory formal methods courses, and how these challenges can be addressed.

The first challenge is a perception problem, summarized by Amazon Web Services (AWS) engineers in their paper "How Amazon Web Services Uses Formal Methods" [35]:

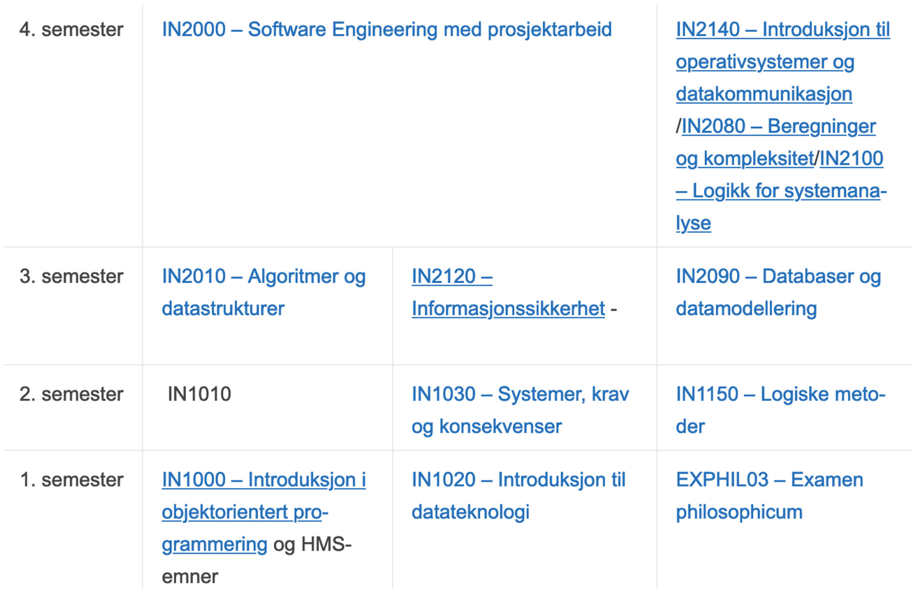
> In industry, formal methods have a reputation for requiring a huge amount of training and effort to verify a tiny piece of relatively straightforward code, so the return on investment is justified only in safety-critical domains (such as medical systems and avionics).

This perception is not restricted to "industry"—and therefore, by word-of-mouth, to prospective students—but may also be shared by our non-formal-methods professor colleagues, which easily leads to formal methods being marginalized in the various course plans, as exemplified by the courses required for the "Programming" bachelor degree at my department[1] shown in Fig. 1.

The introductory formal methods course (IN 2100) competes for a single 10-credit slot with two other courses, one that introduces operating systems and computer networking, which probably appeals quite a lot and seems work-relevant to most students, and an (important) course on computational complexity.[2] I am not convinced that the situation is better at other universities.

[1] Since the bachelor degree is only offered in Norwegian, this course plan is unfortunately only available in Norwegian.

[2] Oddly enough, the formal methods course is placed last in its slot, which is sorted neither by course code nor alphabetically.



4. semester	IN2000 – Software Engineering med prosjektarbeid		IN2140 – Introduksjon til operativsystemer og datakommunikasjon /IN2080 – Beregninger og kompleksitet/IN2100 – Logikk for systemanalyse
3. semester	IN2010 – Algoritmer og datastrukturer	IN2120 – Informasjonssikkerhet -	IN2090 – Databaser og datamodellering
2. semester	IN1010	IN1030 – Systemer, krav og konsekvenser	IN1150 – Logiske metoder
1. semester	IN1000 – Introduksjon i objektorientert programmering og HMS-emner	IN1020 – Introduksjon til datateknologi	EXPHIL03 – Examen philosophicum

Fig. 1. The course plan for the "Programming and Networks" bachelor degree at my university. The third year is devoted to freely selected courses and is not shown.

I would not fault a student for taking a course on operating systems and networking instead of formal methods. I would probably have done so myself as a young student with an eye on the non-academic job market.

The second part of the quote above deals with the perception—often heavily promoted by ourselves—that formal methods are important for safety-critical systems like aircrafts and nuclear power plants. Since Norway, where I currently work, does not produce aircrafts (or, as far as I know, larger medical devices) and does not have commercial nuclear power plants, justifying formal methods with such safety-critical applications may not sway the average 19-year-old student.

I think that some solutions to the above challenges is to emphasize that formal methods provide useful and cost-efficient methods to achieve high-quality non-safety-critical systems. Whereas previously, applying formal methods to an in-house system intended for an in-house user base would probably not be worth it, today we live in a globalized "winner-takes–all" world: Only the "best" program/system/application in each domain/problem (online auctions, social media, search, cloud provisioning, photo storage, online meetings, and so on) will be widely adopted, and these "winners" will rake in billions of dollars, while the runners-up disappear. Amazon, Google, Facebook, Alibaba, and Tencent are among the top 10 companies in the BrandZ Top 100 Most Valuable Global Brand ranking 2019.[3]

While none of these companies produce what we would call safety-critical systems, their products are complex distributed systems where any flaw (e.g., Gmail losing your emails from time to time, Facebook losing your photos or leaking your confidential data, or Amazon Web Services losing some of the data stored for you) could (should?) lead to loss of consumer confidence, with users taking up competing systems, costing billions of dollars and potentially killing the company. Today's systems rely heavily on complex algorithms—just think of the many variants of Paxos that feature prominently in large cloud-based applications—and on large libraries. The application of formal methods on such complex algorithms and libraries should therefore be very worthwhile.

The above-mentioned AWS paper [35] makes a strong case for using formal methods in industry. The sentences after the above quote are:

> Our experience with TLA+ shows this perception to be wrong. [...] Amazon engineers have used TLA+ on 10 large complex real-world systems. In each, TLA+ has added significant value, either finding subtle bugs we are sure we would not have found by other means, or giving us enough understanding and confidence to make aggressive performance optimizations without sacrificing correctness.

I can add that Facebook, Google, Amazon, and others are hiring, and have recently hired, many formal methods researchers.

While I am skeptical to focus on safety-criticality, there *are* a number of fashionable safety-critical systems these days that might motivate students: self-driving cars, embedded devices, drones, and maybe even power distribution. Blockchains and their electronic contracts are also "sexy" topics that could motivate the use of formal methods.

Nevertheless, I think that we must emphasize the usefulness of formal methods in mainstream software development, and provide examples that seem more work-relevant to the 19-year-old student than airplanes and nuclear power plants.

Changing the (mis-)perceptions of our esteemed professor colleagues is probably difficult. Maybe the best (or only) option to gain their appreciation is by showing how formal methods can perform interesting analysis of systems in their fields of expertise. Neither do I have brilliant ideas on how to make students choose formal methods (which they probably have not even heard about when they select courses) instead of seemingly more work-relevant courses. The most realistic approach is to make excellent and fun formal methods courses that seem relevant to students who will soon look for jobs, and hope that the courses grow year by year through word-of-mouth. To achieve this, an introductory formal methods course should demonstrate its usefulness on non-trivial applications in different domains/problems that seem work-related to the student.

[3] https://www.cnbc.com/2019/06/11/amazon-beats-apple-and-google-to-become-the-worlds-most-valuable-brand.html.

Another challenge to the uptake of formal methods is that students tend to have worse mathematical background than ever [34,53] and (maybe therefore) are skeptical to mathematics. After all, if they were into mathematics they probably would study mathematics or physics instead of computer science. The straight-forward way of addressing this problem is to base your teaching on intuitive formal methods that do not require too much mathematical background.

This discussion therefore also leads to another conclusion: *Automatic* model checking methods should be emphasized, even ahead of (or together with) theorem proving. Model checking allows us to analyze even fairly complex and interesting systems with modest effort (only modeling). It is also worth emphasizing that the use of formal methods at AWS reported in [35] solely used model checking, which nevertheless increased their confidence so much that they released sophisticated products without formally verifying them.

A problem often mentioned is that formal methods teaching is not integrated with other courses, and should be parts of other courses (see, e.g., [55]). I am not sure how realistic such an approach is, because:

1. Teachers of other courses may not be formal methods experts and would therefore be unwilling and/or unable to use such methods in their courses. Furthermore, adding formal methods to their courses inevitably means that they have to remove some of their own stuff from the course, which most professors are reluctant to do.
2. Introducing a formal method and an associated tool to the degree that it can be useful on applications in other courses may itself require a few lectures.

Instead, I believe that a realistic, and even quite good, solution is to apply formal methods on systems/algorithms encountered in other courses that students are taking, for example on security protocols, transport and other network protocols, databases/distributed transactions, and operating systems algorithms. I also think that applying formal methods on systems that the students study in other courses is crucial to illustrate that formal methods cannot only be used on avionics, but on the kinds of systems that the students will face when they start working. As explained in Sect. 5.5, this is the approach I have followed, and it involved asking professors teaching databases and distributed systems about algorithms that would be interesting to formalize and formally analyze.

Finally, the last "problem," mentioned in [22] is that:

> Courses on formal methods are often based on examples. [...] However, examples often fall into one of two categories: First, many are constructed and thus do not relate to practice. Second, examples are based on projects of industry partners and are, thus, way too involved for students to understand them.

The solution to this problem is to study systems which look relevant, for example for social media (e.g., distributed transactions), online shopping, and cloud applications (Gmail, ebay, etc.). Even simple examples such as distributed atomic commit protocols and distributed leader election and consensus algorithms can be motivated by such applications, as explained in Sect. 5.5.

To summarize, in this Sect. 1 have argued for teaching formal methods using a fairly *expressive, intuitive, and general formalism* that allows the students to easily model and analyze a range of relevant-looking systems/algorithms, for example those they study in other courses.

3 Making Formal Methods Teaching Fun

The stated goal of the FMfun 2019 workshop is to investigate how formal methods can be taught in such a way that every student can have fun with them.

To make teaching formal methods fun, let us try to figure out what a computer science student thinks is "fun." First of all, why does someone choose to (continue to) study computer science? I think that there are two main reasons:

1. (S)he thinks that programming is fun.
2. The job market for computer science graduates has always been (perceived to be) excellent.

Therefore, to make formal methods fun, we should base it on "programming." But what kind of programming? As an undergraduate student way too many years ago, I got a taste of: standard imperative programming in a Pascal-like language; C programming; assembly programming; and functional programming in LISP/Scheme and a local functional language. While I enjoyed all of these programming paradigms, I was most fascinated by the power and elegance of functional programming. Therefore, at least for me, a formal method involving "programming" in a functional-programming style would be the most "fun."

As for applications, some of the very few relevant hits I got when I searched for "teaching formal methods" and "fun" suggested using formal methods on card tricks [15] and on games and puzzles such as Pac-Man, chess, Sudoku, and the wolf-goat-cabbage problem [22, 50]. However, without having any evidence, I think that applying formal methods to relevant computer systems, such as distributed algorithms and security protocols, should be more "fun" and certainly much more motivating for the students, in particular since many of them study computer science because of the job prospects. Furthermore, even if a student becomes proficient in applying a formal method on small games, that may not teach them how to apply the method on computer systems. Finally, this approach would also perpetuate the misconception that formal methods can only be applied to artificial toy examples.

Let me end this section by mentioning two things that are *not* fun:

1. Struggling with an immature and buggy tool.
2. A number of students once complained that they were taking a formal methods course using some kind of automata to model and analyze distributed systems, and that the hacks and tricky encodings needed to model anything of interest made it very "un-fun." Again: a nice powerful modeling language that allows you to easily and elegantly model non-trivial systems without awkward encodings are needed to make formal methods "fun."

4 How to Teach Formal Methods?

This section first discusses some seminal papers on teaching formal methods. It then presents my thoughts on what to teach, and finally summarizes all the requirements for a "fun" formal methods course that I have derived in this paper.

4.1 Related Work

This section summarizes a few key papers on teaching formal methods. A common thread in these papers is that their recommendations do not seem to have been properly scientifically validated: the authors just get the impression and some anecdotal evidence that their suggestions work well in their teaching. I follow their lead in Sect. 6.

In "Teaching Formal Methods in the Context of Software Engineering," Shaoying Liu and researchers at The Nippon Signal Co. propose using a combination of VDM, refinement calculus, and Hoare logic to teach formal methods in a software engineering context (which is also the context of the present paper) [24]. In contrast to almost all other papers on the subject I have read, Liu at al. think that using tools when teaching formal methods is "perhaps less effective" than *not* using a tool, since "most effective for students [...] is to write formal specifications by hand, [just] as they learn English as a foreign language."

However, Liu et al. admit that their suggested formal methods are not easy to use by practitioners on real software projects and that "there is little hope to apply the refinement calculus in practice." In a recurring theme among papers on the topic, Liu et al. also say that each course should not be too ambitious, and should instead be focused: It takes time to digest and master mathematical concepts, and we should teach them slowly with many examples. Then there is just not enough time to introduce too many formal methods concepts.

That the field of formal methods is too large to gain encyclopedic knowledge, and that one should therefore choose a non-representative selection of formal methods to teach is also the first of ten "principles for teaching Fun With Formal Methods" given by Antonio Cerone and others in their paper "Teaching Formal Methods for Software Engineering – Ten Principles" [10]. In contrast to Liu et al., Cerone et al. advocate strongly for teaching using available and stable "tools for simulation of behaviour and visualization of state space or traces" that are powerful, even industrial-strength, and come with many "big" examples (Principles 3, 5, and 6). The modeling language should make it easy to model systems at a suitable level of abstraction (Principle 4). Principle 7 says that formal methods are best taught by (computing) examples that are familiar to students, which is in contrast to studying formal methods using card tricks and games and puzzles. Cerone et al. end their list of principles by asking us to shout out loud that formal methods are fun, and to motivate the students to participate in competitions such as the SAT competition. I am not convinced that shouting out loud that formal methods are fun is a good idea, or that a student will be attracted to a formal methods course for the opportunity to participate in the SAT competition, but I share their opinion that human learning capacity is highest when we enjoy what we are doing, so formal methods must be fun!

Luca Aceto and others [1] also argue that *less is more* in formal methods education, emphasizing the need to repeatedly convey a few key concepts instead of giving a broad overview. Their "main messages" are that formal models should be developed using very expressive and flexible, but mathematically simple, executable formalisms, that modal and temporal logics are fundamental to specify system requirements, and that automatic verification tools should be used.

4.2 What to Teach?

What should be taught in an introductory formal methods course aimed at second-year university students?

The main point of any university course is to teach *concepts*, and not single logics, tools, and formalisms for their own sake. In my view, the key concepts in formal methods are:

1. Mathematical *modeling/formalization* of both *systems/designs* and of the *properties/requirements* that the systems should satisfy.
2. *Reasoning* about such systems models and whether they satisfy their requirements. There are two main ways to do this: automatic *model checking* and interactive *theorem proving* verification. In today's world, where performance often is as important as correctness, model-based reasoning about system *performance* would also be useful.
3. Mathematical analysis of *programs/code*. (A related, less central, concept is how to obtain correct code from a verified formal specification.)

It would also be good to give the student a flavor of logical reasoning in general. If possible, the student should be introduced to some logic and the concepts of logical deduction, model theory, satisfaction, maybe even soundness and completeness, and so on. The student should also be exposed to key folklore results, such as basic undecidability results (for example of termination and reachability of certain states, which they can relate to their imperative programs).

The main applications of formal methods are modeling and analyzing *designs/algorithms* and analyzing *program code*. In an introductory course at a non-selective university you may not be able to cover both ("less is more"). In that case, focusing on modeling and formally analyzing high-level designs seems to be the best choice for a number of reasons:

1. Programmers code pretty well, and there are also good programming environments and tools for generating or developing correct code from correct specifications. In an early example illustrating the importance of developing correct system models, it turned out that only *three* of the 197 critical defects identified during integration and testing of the Voyager and Galileo spacecrafts were due to coding errors [27,48]. Most faults arose in requirements and difficult design problems related to distribution [48]. Furthermore, John Rushby wrote in 2011 that "no [plane] crash has ever been caused by software

error" [49], and what we know of aircraft problems (such as Boeing 737 Max) since then confirms this: the problems lie in understanding the problems and developing a correct design. I discuss the successful use of formal methods at Amazon Web Services in Sect. 5.5; this also concerned modeling and model checking designs of distributed algorithm—not code.
2. Not only are defects more likely to be introduced in the early stages of system development; it is also much cheaper to catch errors as early as possible.
3. It is easier to achieve something interesting in a short time with modeling and analysis of high-level designs than with program verification, which typically requires defining the formal semantics of a (possibly toy) programming language combined with theorem proving.

The courses discussed in Sect. 4.1 deal with modeling and analyzing high-level designs. Program language semantics and verification are usually part of courses on programming languages.

4.3 Summarizing the Requirements

To summarize, in my view a fun introductory course in formal methods should satisfy the following criteria:

1. Be based on functional (or another fun) style of programming for executable systems modeling.
2. Be based on a fair amount of examples/applications, which should be relevant for other computer science courses that the students are taking, and which should be seen as industrially relevant.
3. Should use a few mature and available tools that are seen as relevant in industry.
4. Should motivate formal methods with industrial successes, preferably not only on safety-critical systems.
5. Should introduce the key concepts in formal methods:
   - modeling system designs;
   - formalizing system requirements;
   - formal correctness analysis, by model checking and by theorem proving, and possibly also support formal model-based performance analysis;
   - program verification; and
   - provide some basics of logics and key folklore results.

On the other hand:

6. The course should focus on a few concepts.

It follows from these requirements that we need an *expressive* executable formalism, that allows us to easily model a range of not-entirely-trivial systems, preferably in different domains (e.g., taken from different other courses). The formal method should also be simple and intuitive and should not require much mathematical background. If we want students to achieve meaningful results on interesting problems, this may lead us to prefer automatic model checking analysis over interactive theorem proving verification, which scales less well to non-trivial systems in the short time available in an introductory formal methods course. Finally, we must show that the formal method has industrial relevance.

5 Teaching Introductory Formal Methods Using Rewriting Logic

This section gives an overview of an introductory formal methods course—and its accompanying textbook—aimed at second-year undergraduate students at the University of Oslo that tries to teach formal methods according to the criteria in Sect. 4.3. I first present the setting of the course. Sections 5.2 and 5.3 briefly introduce rewriting logic and its associated modeling language and formal analysis tool Maude. Section 5.4 explains why I think that Maude is a promising formal modeling language and analysis tool addressing the requirements for teaching formal methods for fun. Finally, Sect. 5.5 gives an overview of the course and its textbook, with some samples thrown in to give a flavor of modeling and analysis in Maude.

5.1 Course Setting

As already mentioned, the course is an elective course taught to second-year "programming and networks" students at the University of Oslo. The course was taught to third- and fourth-year students until 2018, when the textbook was published. The course has one 90-minute lecture and one 90-minute seminar, discussing solutions to weekly exercises, per week for 15 weeks. As shown in Fig. 1, the students have taken some general imperative programming courses and a number of software engineering courses, as well as introductions to databases and computer security, before taking this class. They have also taken a basic first-year introduction to standard mathematical logic, although my course does not assume any significant knowledge of mathematical logic.

5.2 Formalism Used: Rewriting Logic

I base the course on *rewriting logic* [9,28,31], which is a simple but powerful logic of *change* developed by José Meseguer in the late 1980s and early 1990s. Rewriting logic has been shown to be very suitable to model a wide range of distributed systems in a natural way. In particular, rewriting logic has a simple model of concurrent objects, which are ideal to model distributed systems.

In rewriting logic, data types (domains and functions on such domains) are specified using (first-order) algebraic *equational specifications*, which could be many-sorted, order-sorted, or being based on membership equational logic [29].

Dynamic behaviors are then specified by labeled conditional rewrite rules $l : t \longrightarrow u \;\; \textbf{if} \; cond$, where t and u are two terms[4] representing *state patterns.*

5.3 Language and Tool Used: Maude

Maude [13,14,16] (http://maude.cs.illinois.edu) is a specification language and high-performance analysis tool for rewriting logic developed at SRI International

[4] More precisely, they are equivalence classes of terms modulo the equations in the equational specification.

and the University of Illinois since the beginning of the 1990s. Maude supports convenient mix-fix operator syntax; deduction modulo equational axioms such as associativity, commutativity, and identity, and their combinations; and order-sorted and membership equational logic theories. Maude assumes that the equations (when oriented from left to right) are ground confluent and terminating (modulo the equational axioms), and executes an equational specification by computing the normal form of a term in a standard term rewriting sense.

Since rewrite rules modeling atomic transition patterns may not be terminating and/or confluent, there are different ways of formally analyzing rewriting logic specifications (called *rewrite theories*). *Rewriting* applies rewrite rules to a ground term representing the initial system state to simulate one possible behavior of the system from the initial state, and explicit-state *reachability analysis* uses a breadth-first search strategy to search for states reachable from the initial state that match a given *state pattern*. More sophisticated system requirements can be formalized as *linear temporal logic* (LTL) formulas [12], where atomic propositions are terms of sort `Prop` and LTL formulas terms of sort `Formula`. If the state space reachable from the initial state is finite, Maude's explicit-state LTL model checker can check whether all behaviors from the initial state satisfies an LTL property. Maude has recently been equipped with *symbolic* analysis methods (where reasoning is performed on state patterns, i.e., terms with variables, that represent infinite sets of concrete states), such as narrowing and rewriting combined with SMT solving for symbolic reachability analysis [14,16].

Thanks to Maude's meta-programming features, where any Maude module can be represented as a term of a sort `Module` at the Maude meta-level, and where we hence can define Maude functions on such (meta-represented) Maude modules, the user can define specific analysis commands herself. She can also do so in Maude 3 using Maude's *strategy language*.

Maude specifications can also be subjected to interactive theorem proving verification of invariants [45] and reachability logic properties [52].

It should also be mentioned that rewriting logic has natural extensions to model probabilistic [2] and real-time systems [37]. Such systems can be analyzed by, respectively, statistical model checkers such as PVeStA [3] and MultiVesta [51], and by the Real-Time Maude tool [36,38].

5.4 Why Maude?

How does Maude address the requirements in Sect. 4.3 for teaching formal methods in a "fun" way?

- Maude provides a fun functional-programming-style specification of data types and a functional-programming and object-oriented style of modeling distributed systems, which are the systems we want to target these days.
- Maude provides a very simple and intuitive formalism that does not require much (if any) mathematical background. The students should be familiar with equations, having used equations such as $(x + y)^2 = x^2 + 2xy + y^2$ as simplification rules in school.

– The Maude formalism is very general and expressive, so that a wide range of distributed systems and forms of communication can be easily modeled at the desired level of abstraction, without tricky encodings. This makes it possible to specify different kinds of non-trivial systems in the limited time frame of such an introductory course.
– As argued above, to illustrate the use of formal methods on interesting problems, one may have to prefer automatic model checking methods over interactive theorem proving methods in such an introductory course, and Maude provides automatic reachability analysis and LTL model checking.
– The tool is quite mature and efficient, is freely available, and is very easy to install on Linux platforms. Furthermore, I have never had a student with a Windows machine who could not run Maude.
– Although neither my course nor my textbook covers it, rewriting logic can also be used to *verify programs* in a wide range of languages, such as C, Java, and so on, using Grigore Rosu's rewriting-logic-based K framework [47] and matching logic [46].
– Students tend to be more motivated to use a new tool when it seems relevant to industry and is used on interesting real applications. Maude and related tools have been applied to a wide range of complex systems. For example, in security, Maude was applied at Microsoft to discover previously unknown address bar and status bar spoof attacks in Internet Explorer [33], and one of the leading formal crypt-analysis tools, the latest version of Cathy Meadows' NRL Protocol Analyzer, called Maude-NPA, is written in Maude. We have already mentioned Grigore Rosu's work on rewriting logic semantics of programming languages [7,17,30,32]. This framework is used in a commercial setting to formalize the Ethereum Virtual Machine and to formally analyze electronic contracts on the blockchain [20,43]. Maude and PVeStA have been used to formally model and analyze both the correctness and performance of large transport protocols [23,39], state-of-the-art wireless sensor network algorithms [21,40], and large cloud-based transaction systems such as Google's Megastore [18,19], Apache Cassandra [25], and others (see [6,41] for an overview). Researchers at NASA have used Maude to verify programs written in NASA's PLEXIL language for commanding and monitoring autonomous systems [44]. In the biological and medical domains, rewriting logic and Maude have been used to formalize and analyze cell biology [54] and simple models of biochemical processes in the brain [4,5]. Maude has also been used to reason about human cognition [11], in particular human multitasking [8]. Th survey paper [31] gives a more comprehensive overview of some applications of Maude as of 2012. My students may also be inspired by the fact that two of my former TAs started a company with a product written in Maude that is still thriving, more than 15 years later.
– Since Maude provides support for sockets, a Maude instance can communicate with other Maude instances and with other external objects. In [26] this is used to automatically generate correct-by-construction *distributed implementations* with decent performance from verified Maude models of distributed

transaction systems. These implementations can then run on real workloads, such as those generated by YCSB.

5.5 Overview of the Course and Its Textbook

This section summarizes the content of the course and its textbook, *Designing Reliable Distributed Systems: A Formal Methods Approach Based on Executable Modeling in Maude*, which was published in 2018 as a volume in Springer's *Undergraduate Topics in Computer Science* series.

The course (and the textbook) are divided into two parts: Part I shows how to define data types in Maude, and gives a quite standard introduction to algebraic equational specifications and term rewrite systems. Part II explains how the dynamic behaviors of distributed systems can be modeled and analyzed in Maude.

To give a flavor of the course, I also give a few small examples of specification and analysis in Maude. The section headers show in parenthesis the number of 90-minute lectures I devote to each topic.

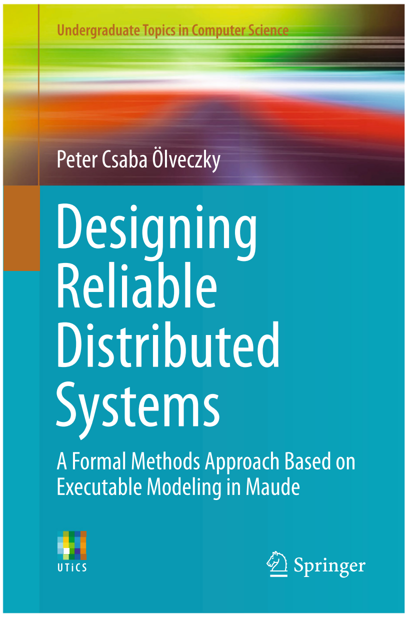

Undergraduate Topics in Computer Science
Peter Csaba Ölveczky
Designing Reliable Distributed Systems
A Formal Methods Approach Based on Executable Modeling in Maude
UTiCS
Springer

Fig. 2. Course textbook

Equational Specification in Maude (3.5 Lectures). This chapter introduces equational specification of data types in Maude, starting with a "Hello Word" example, a specification of the natural numbers with addition in a Peano style:

Example 1. The following Maude functional module (`fmod`) defines a sort `Nat` whose constructor ground terms `0`, `s(0)`, `s(s(0))`, ... represent the natural numbers 0, 1, 2, ..., and defines the addition function on such (representations of) natural numbers, where '`_`' denotes the argument positions in "mix-fix" function symbols:

```
fmod NAT-ADD is
  sort Nat .
  op 0 : -> Nat [ctor] .                vars M N : Nat .
  op s : Nat -> Nat [ctor] .            eq 0 + M = M .
  op _+_ : Nat Nat -> Nat .             eq s(M) + N = s(M + N) .
endfm
```

Maude's *reduce* (`red`) command can then be used to compute the value of $3+2$:

```
Maude> red s(s(s(0))) + s(s(0)) .
...
result Nat: s(s(s(s(s(0)))))
```
□

In this way, we define data types such as lists, multisets, binary trees, graphs, and so on, in rewriting logic/Maude. "Syntactic subtypes" can be defined using *subsorts*, and "semantic subtypes" can be defined by *membership axioms*. A (binary) function/operator can be declared to be associative (`assoc`), commutative (`comm`), and/or to have an identity element t (`id:` t), so that matching is performed *modulo* these properties.

Example 2. Combining subsorts and operator attributes, we can define lists and non-empty lists (of natural numbers) as follows:

```
fmod LIST is
  protecting NAT .
  sorts List NeList .  subsorts Nat < NeList < List .
  op nil : -> List .
  op _:_ : List List -> List [ctor assoc id: nil] .
  op _:_ : NeList NeList -> NeList [ctor assoc id: nil] .
endfm
```

The list $\langle 2, 8, 5, 3\rangle$ is then represented as the term `2 : 8 : 5 : 3` of sort `NeList`; since `NeList` is a subsort of the sort `List`, this term is also a term of sort `List`.

We can then define the *insertion sort* algorithm, which sorts a list by inserting the elements, one by one, in the right place in the sorted list of the elements that have already been treated. In the auxiliary function, the first argument is the elements that have not yet been inserted into the sorted (sub)list, and the second argument is the sorted list of elements that have already been treated:

```
fmod INSERTION-SORT is protecting LIST .
  op insertionSort : List -> List .
  op insertionSort : List List -> List .

  vars L L2 L3 : List .     vars M N K : Nat .

  eq insertionSort(L) = insertionSort(L, nil) .
  eq insertionSort(M : L, nil) = insertionSort(L, M) .
 ceq insertionSort(M : L, N : L2) = insertionSort(L, M : N : L2) if M <= N .
 ceq insertionSort(M : L, L2 : N) = insertionSort(L, L2 : N : M) if M > N .
 ceq insertionSort(M : L, L2 : K : N : L3)
   = insertionSort(L, L2 : K : M : N : L3) if M > K and M <= N .
  eq insertionSort(nil, L) = L .
endfm
```

```
Maude> red insertionSort(8 : 5 : 12 : 2 : 45 : 3 : 45 : 46 : 47) .
...
result NeList: 2 : 3 : 5 : 8 : 12 : 45 : 45 : 46 : 47
```
□

Multisets (of, say, natural numbers) can be defined equally easily using an associative and commutative multiset union operator (which we denote by empty syntax: _ _):

```
fmod MULTISET-NAT is protecting NAT .
  sort Mset .          subsort Nat < Mset .
  op none : -> Mset [ctor] .
  op __ : Mset Mset -> Mset [ctor assoc comm id: none] .
endfm
```

As examples, and to "sneak-introduce" classic NP-complete problems, the book introduces and defines functions solving problems such as *subset sum*, *Hamiltonian circuit*, *(integer) knapsack*, and the *traveling salesman* problem.

Example 3. The *subset sum* problem, where the question is to decide whether it is possible to pick a subset of numbers with sum K from a given multiset M of natural numbers, can be solved as follows (where `sd` denotes symmetric difference ("minus") on natural numbers):

```
  op subsetSum : Mset NzNat -> Bool .
  vars N : Nat .   var NZ : NzNat .  var REST : Mset .
  eq subsetSum(none, NZ) = false .
  eq subsetSum(N REST, NZ)
   = if N > NZ then subsetSum(REST, NZ)
     else (if N < NZ then subsetSum(REST, sd(NZ,N)) or subsetSum(REST,NZ)
           else true fi) fi .   --- N == NZ
```
□

Finally, the book discusses parametrized modules in Maude, which are not taught in class, and the Bergstra-Tucker meta-theorem that any computable data type can be defined by a terminating and confluent equational specification.

Operational Semantics (Half a Lecture), Termination (1–2 Lectures), and Confluence (1 Lecture). This part defines the operational semantics of equational specifications (by rewriting). Since this is an introductory textbook, all treatment of theoretical issues is restricted to one-sorted unconditional theories without operator attributes such associativity and commutativity.

Since Maude assumes the equations to be terminating and (ground) confluent, we must be able to reason about termination and confluence. The book gives a proof for the undecidability of termination using Turing machines. It then shows how "weight" functions, where each ground term is assigned a weight in a well-founded strict partial order, can be used to prove termination. The book then explains the elegant theory of *simplification orders*, which leads to the lexicographic and multiset path orders (*lpo* and *mpo*, respectively). I used to teach the theory of simplification orders, but now omit it for the second-year students (who must learn temporal logic instead). The book contains lots of examples and exercises, including indicating how the techniques also can be applied to imperative programs. One exercise is to implement *lpo*, which can be done very elegantly in Maude, and which also implicitly introduces *meta-programming.*

A chapter on checking confluence leads to the critical-pair algorithm for checking confluence in terminating specifications.

Equational Logic (1 Lecture). To introduce students to fundamentals such as proof systems, proof theory, and logics, the course introduces equational logic (again, in its basic unsorted version), with its deduction rules, and basic results such as undecidability of equality in the general case, and decidability when the specification is terminating and confluent. The second part of that chapter deals with inductive theorems, and includes an explanation of how it follows from the negative solution to Hilbert's Tenth Problem that there is no finitary sound and complete proof system for inductive theorems. This part also presents the general "constructor induction" scheme for proving inductive theorems, applied to simple equalities for lists and binary trees, and shows how Maude in some cases can prove inductive theorems automatically.

Models of Equational Specifications. The chapter on the model theory for algebraic specifications gives the basics: σ-algebras, term algebras, (Σ, E)-algebras, quotient algebras, the algebra $\mathbb{T}_{\Sigma,E}$, proof of the soundness and completeness of equational logic, and explains how initial algebras are the intended models that satisfy expected properties. This chapter is not taught in the course.

Rewriting Logic and Executing Rewrite Theories in Maude (1 Lecture). In rewriting logic, data types are defined as equational specifications, and dynamic behavior is modeled by *labeled rewrite rules* $l\ :\ t \longrightarrow t'$ **if** *cond*, where l is a label, and t and t' are terms that should be seen as *state fragments*, parametrized by the variables that appear in the rule. The key point is that the rewrite rules, modeling dynamic behaviors, need not be terminating and/or confluent. This chapter introduces rewriting logic and its deduction rules, as well as how to reason logically about which steps can be performed concurrently.

In Maude, a rewrite rule is executed by first reducing the state to its equational normal form, and then applying the rewrite rule to simulate one step of the system. Maude's *rewrite* (`rew`) command simulates *one* of the behaviors from a given initial state. Maude's `search` command performs breadth-first search to check whether a given state pattern is reachable from a given initial state. We apply Maude to model and analyze small games and populations of humans, simulating Turing machines, and exhibiting solutions to NP-complete problems such as *knapsack* and *traveling salesman.*

Example 4. In the blackboard game, a bunch of natural numbers are written on a blackboard. In each step of the game, any two numbers on the blackboard can be replaced with their arithmetic mean. This exciting game can be modeled as follows in Maude, where the blackboard is represented as a multiset of numbers:

```
mod BLACKBOARD-GAME is including MULTISET-NAT .
  vars M N : Nat .
  rl [replace] : M N => (M + N) quo 2 .
endm
```

We can simulate one behavior of this game from the blackboard `98 2 4 56 7`:

```
Maude> rew 98 2 4 56 7 .
result NzNat: 64
```

and can check whether it is possible to reach a state where the blackboard only has a single number, which, in addition, is less than 15:

```
Maude> search [1] 98 2 4 56 7 =>* N such that N < 15 .
Solution 1 (state 156)
N --> 14
```

□

Object-Oriented Specification in Maude (1 Lecture). A convenient way to represent the state of a distributed system is as a multiset of *objects* and *messages* traveling between the objects. Objects and messages can be any terms; a convenient notation we use is that the term

`<` o `:` C `|` att_1 `:` val_1`,` ... `,` att_n `:` val_n `>`

denotes an object o of class C, with attributes att_1 to att_n, whose current values are val_1 to val_n, resp. A message is a term of sort `Msg` which in this course has the form `msg` *content* `from` o_1 `to` o_2.

Full Maude is an extension of Maude, specified in Maude, that provides convenient syntax for object-based specification, as well as support for subclasses. In Full Maude, a class is declared `class` C `|` att_1 `:` s_1`,` ... `,` att_n `:` s_n `.` This chapter illustrates object-oriented specification not only with populations of humans, but also with the *dining philosophers* problem and with *blackjack*, where we use Maude's `random` function to draw cards pseudo-randomly and to simulate the outcome of playing blackjack with different strategies.

Modeling Communication and Transport Protocols (1 Lecture). The book then explains how different forms of communication, including synchronous communication, (unordered) unicast, multicast, and broadcast, message loss and duplication, ordered unicast, wireless broadcast, and communication using "shared variables" can be abstractly modeled in Maude.

This enables us to start modeling and analyzing some of the most well-known and key distributed algorithms/protocols, and we start by modeling and analyzing classic transport protocols such as TCP, the alternating bit protocol, and different versions of the sliding window protocol.

Distributed Algorithms (1 Lecture). The chapter which shows how Maude can be used to formalize and analyze central algorithms in distributed systems is an important chapter in the book. The algorithms were selected as follows:

- A professor colleague in Oslo from the database community challenged me to model and analyze the two-phase commit (2PC) protocol.
- A professor teaching distributed systems at the University of Illinois suggested some key algorithms in distributed systems.
- When I was part of the University of Illinois Center for Assured Cloud Computing, I noticed that 2PC and distributed consensus algorithms (in particular various flavors of Paxos) show up as key components in many cloud-based systems, such as Google's Megastore and UC Berkeley's RAMP transactions.
- It is easy to motivate the selected algorithms with simple use cases.

The chapter first treats the *two-phase commit* (2PC) protocol, admittedly a simple protocol, which is nevertheless much used. It is also easy to motivate: a transaction today is typically a *multi-site* transaction. For example, a travel agent may sell a trip with both hotel room and plane ticket included. Such a transaction involves at least three different sites: the flight reservation system, the hotel reservation system, and the payment processing system. If one of the operations fails (there are no flights or no hotel rooms, or the payment is unsuccessful), the whole transaction must be aborted. Modern systems *replicate* data for availability and disaster tolerance; therefore, two different replicating sites/servers may sell the *same* seat on a flight (or the same unique ebay item) to two different persons at the same time. 2PC solves the problems by aborting the transaction unless all servers agree to commit the transaction (which they will not do if there are double bookings, or if the payment (or the hotel reservation or the flight reservation) fails).

The part on 2PC also discusses techniques for injecting faults into the system.

Distributed mutual exclusion algorithms are also easy to motivate (e.g., to avoid lost updates in a distributed setting, or to disallow that the same flight seat can be accessed (and hence sold) by different servers at the same time). We model and analyze the *central server*, the *token ring*, and the *Maekawa* distributed mutual exclusion algorithms. Exam problems have asked students to model and analyze Lamport's *bakery* algorithm and the *Suzuki-Kasami* algorithm.

Instead of canceling *both* transactions when the same seat is sold to two persons, it would be much better if the sites can agree (i.e., reach consensus) on *one* person to sell the ticket to. This leads to distributed consensus algorithms, which typically include distributed leader election algorithms as key components. We study a distributed token ring leader election algorithm, as well as a spanning-tree-based leader election algorithm that is the basis of many *wireless* algorithms. The book also discusses distributed consensus and gives a *very* abstract description of Paxos, but does not provide details.

Example 5. In the *token ring* distributed mutual exclusion algorithm, the nodes form a ring. Each node executes forever, alternating between executing outside its critical section and executing inside its critical section. There is one *token* that the nodes send along the ring; a node can only execute inside its critical section when it holds the token.

This algorithm can be specified in (Full) Maude as follows:

```
load full-maude

(omod TOKEN-RING-MUTEX is
  sort Status MsgContent .
  ops outsideCS waitForCS insideCS : -> Status [ctor] .
  op msg_from_to_ : MsgContent Oid Oid -> Msg [ctor] .
  op token : -> MsgContent [ctor] .

  class Node | next : Oid, status : Status .

  vars O O1 O2 : Oid .

  rl [wantToEnterCS] :
     < O : Node | status : outsideCS >
    =>
     < O : Node | status : waitForCS > .

  rl [rcvToken1] :
     (msg token from O1 to O)
     < O : Node | status : waitForCS >
    =>
     < O : Node | status : insideCS > .

  rl [rcvToken2] :
     (msg token from O1 to O)
     < O : Node | status : outsideCS, next : O2 >
    =>
     < O : Node | >
     (msg token from O to O2) .

  rl [exitCS] :
     < O : Node | status : insideCS, next : O2 >
    =>
     < O : Node | status : outsideCS >
     (msg token from O to O2) .
endom)
```

The first line starts Full Maude. The class declaration declares a class `Node` with two attributes. The attribute `status` shows the "execution status" of the node, i.e., whether the node is executing outside its critical section (`outsideCS`), is waiting to access its critical section (`waitForCS`), or is executing inside its critical section (`insideCS`). The attribute `next` points to the *object identifier* of the next node in the ring.

In rule `wantoToEnterCS`, a node that is executing outside its critical section needs to enter its critical section, and starts waiting for the token. In rule rcvToken1, such a waiting node receives the token (message), and starts executing inside its critical section (i.e., changes its status to insideCS). In rule rcvToken2, a node that is executing outside its critical section receives the token, and just sends the token (message) to the next node in the ring. Finally, in rule exitCS, a node ends its execution inside its critical section and sends the token to the next node in the ring.

The following module defines a suitable initial state init consisting of four nodes, named a, b, c, and d, and where the token is "on the way" to node a:

```
(omod INITIAL is including TOKEN-RING-MUTEX .
  ops a b c d : -> Oid [ctor] .     --- object names
  op init : -> Configuration .      --- initial state
  eq init
   = (msg token from d to a)
     < a : Node | status : outsideCS, next : b >
     < b : Node | status : outsideCS, next : c >
     < c : Node | status : outsideCS, next : d >
     < d : Node | status : outsideCS, next : a > .
endom)
```

We can then simulate 100 steps of this (nonterminating) algorithm:

```
Maude> (frew [100] init .)
...
result Configuration :
  < a : Node | next : b, status : insideCS >
  < b : Node | next : c, status : waitForCS >
  < c : Node | next : d, status : waitForCS >
  < d : Node | next : a, status : outsideCS >
```

The main invariant that the algorithm should satisfy is that two nodes never execute inside the critical section at the same time. We check this invariant by searching for a reachable state where two objects both have status insideCS (variables in search patterns are given as *var* : *sort*):

```
Maude> (search [1] init =>*  REST:Configuration
                             < O1:Oid : Node | status : insideCS >
                             < O2:Oid : Node | status : insideCS > .)

No solution.
```

Finally, we check whether it is possible to reach a deadlock (=>!) from init:

```
Maude> (search [1] init =>! SYSTEM:Configuration .)

No solution.
```

□

Modeling and Breaking Cryptographic Protocols (1 Lecture). One chapter of the textbook gives a basic introduction to cryptography (public/private-key cryptography, shared-key cryptography, digital signatures, and so on), and shows how the well-known *Needham-Schroeder public-key* (NSPK) mutual authentication protocol can be modeled and broken using the Maude techniques that the students have learnt.

Yes, NSPK is a standard example, but it should be inspiring for the students. I use it in the beginning of the course to motivate formal methods:

– NSPK is an excellent example for the need for formal methods. It was a well-known and well-studied protocol from 1978. The *Handbook of Applied Cryptography* from 1996 discusses it without mentioning any flaws. The key (pardon the pun) thing is that it was broken by Gavin Lowe in 1995 using exhaustive analysis of a formal model, which is exactly what we are doing. That is, the flaw in NSPK went undiscovered for 17 years until formal analysis found a successful attack on NSPK.
– NSPK is a prime example of the complexity of distributed systems: the whole protocol is described in *three lines*, yet it is so hard to really understand that the flaw was not found for 17 years.
– More or less all our use of computers (email, social media, online shopping and banking, etc.) is based on our ability to *authenticate* ourselves to a service, so this is an absolutely crucial problem.
– I guess that security is a popular topic with students, and using NSPK allows me to both show the use of formal methods on a sexy topic, as well as to give the students the briefest of crash courses on cryptography.

The NSPK protocol is usually described in standard crypto-protocol notation as follows, where A (the initiator) and B (the responder) are two agents who want to authenticate themselves to each other.

$$\begin{array}{lll} \text{Message 1.} & A \rightarrow B: & A\,.\,B\,.\,\{N_a\,.\,A\}_{PK_B} \\ \text{Message 2.} & B \rightarrow A: & B\,.\,A\,.\,\{N_a\,.\,N_b\}_{PK_A} \\ \text{Message 3.} & A \rightarrow B: & A\,.\,B\,.\,\{N_b\}_{PK_B} \end{array}$$

In the first step, A generates the *nonce* ("fresh random number") N_a, adds her identity A, encrypts this concatenation $N_a\,.\,A$ with the public key of B, and sends this encrypted message, together with her own and B's name (unencrypted) to B. When B receives this first message, he decrypts the encrypted part using his private key to obtain the nonce N_a. The responder B then generates his own nonce N_b, and returns the nonce N_a along with the new nonce N_b, encrypted with the public key of A. When A receives this Message 2 she decrypts it with her private key to read both N_a and N_b, and sends the nonce N_b, encrypted with B's public key, back to B. It should be (and for many years was) obvious that after the three messages have been successfully read and decrypted, that A and B really wanted to participate in a protocol run with each other.

I do not show the declaration of the messages in Maude, but refer to the book [42] for details. For example, a Message 1 can be modeled as the term

```
msg (encrypt (nonce(A,3) ; A) with pubKey(B)) from A to B.
```

where `nonce(A,3)` is the third nonce generated by `A`. Our model allows multiple runs of the protocol with multiple participants. An initiator is an object of class

```
class Initiator | initSessions : InitSessions, nonceCtr : Nat .
```

where `nonceCtr` is a counter for generating nonces, and `initSessions` is a multiset of elements of the following kinds:

- `notInitiated`(B) indicates that A wants to initiate contact with B but has not yet done so;
- `initiated`(B, N) indicates that A has sent Message 1 to B with nonce N and is waiting for Message 2 from B; and
- `trustedConnection`(B) indicates that A has established (what she thinks is) an authenticated connection with B.

The following two rewrite rules model the behavior of initiator nodes. The rule `send-1` models sending Message 1. The agent `A` has `notInitiated(B)` in its `initSessions` attribute, which means that it wants to establish a connection with `B`. The agent `A` generates a fresh nonce `nonce(A,N)` and sends the corresponding Message 1 to `B`. Agent `A` must also remember that it has initiated contact with `B` using nonce `nonce(A,N)` and must increase its nonce counter:

```
rl [send-1] :
   < A : Initiator | initSessions : notInitiated(B) IS,
                     nonceCtr : N >
 =>
   < A : Initiator | initSessions : initiated(B,nonce(A,N)) IS,
                     nonceCtr : N + 1 >
   msg (encrypt (nonce(A,N) ; A) with pubKey(B)) from A to B .
```

In rule `read-2-send-3` an agent `A` receives a Message 2 from `B`. If the first nonce (`NONCE`) in the message received (and decrypted) by `A` is the same as the nonce stored in `A`'s `initSessions` attribute for `B`, then agent `A` figures out that it has established an authenticated connection with `B`, and sends Message 3 (`B`'s nonce (`NONCE'`) encrypted with `B`'s public key) to `B`:

```
rl [read-2-send-3] :
   (msg (encrypt (NONCE ; NONCE') with pubKey(A)) from B to A)
   < A : Initiator | initSessions : initiated(B,NONCE) IS >
 =>
   < A : Initiator | initSessions : trustedConnection(B) IS >
   msg (encrypt NONCE' with pubKey(B)) from A to B .
```

Responders are modeled by two similar rewrite rules (modeling receiving Message 1 and sending Message 2, and receiving Message 3). Some nodes may be both initiators and responders (in different runs of the protocol). They are modeled as subclasses of both `Initiator` and `Responder`, and therefore inherit the attributes and rewrite rules of both superclasses:

```
  class InitAndResp .
  subclass  InitAndResp < Initiator Responder .
```

“Dolev-Yao” intruders are modeled by specifying their capabilities, and by in each step also storing any new agent names, nonces, or encrypted messages whose content it cannot understand:

```
  class Intruder | initSessions : InitSessions,
                   respSessions : RespSessions,   nonceCtr : Nat,
                   agentsSeen : OidSet,
                   noncesSeen : NonceSet,
                   encrMsgsSeen : EncrMsgContentSet .
```

Rules then model the intruder participating in normal protocol runs (and storing the obtained information), intercepting and stealing messages, and sending any kind of fake messages, using information it has gathered. For example, in the following rule, an intruder sends out a completely random Message 2:

```
  crl [send-2-fake] :
      < I : Intruder | agentsSeen : A ; B ; OS,
                       noncesSeen : NONCE NONCE' NSET >
    =>
      < I : Intruder | >
      (msg (encrypt (NONCE ; NONCE') with pubKey(A)) from B to A)
     if A =/= B /\ A =/= I .
```

We then define the following initial state `intruderInit`:

```
 op intruderInit : -> Configuration .
 eq intruderInit
  = <"Scrooge" : Initiator |
        initSessions : notInitiated("BeagleBoys"), nonceCtr : 1 >
    < "Bank" : Responder |
        respSessions : emptySession,  nonceCtr : 1 >
    < "BeagleBoys" : Intruder |
        initSessions : emptySession,  respSessions : emptySession,
        nonceCtr : 1,   agentsSeen : "Bank" ; "BeagleBoys",
        noncesSeen : emptyNonceSet,  encrMsgsSeen : emptyEncrMsg > .
```

The Beagle Boys do not know any other agent, except the bank, but hope to be contacted by some rich guys after creating an enticing web site promising ... Indeed, Scrooge wants to contact the Beagle Boys but *not* the bank. Therefore, if it is possible to reach a state where the bank thinks that it has established an authenticated connection with Scrooge, then the protocol is broken, and Scrooge's wealth can be transferred to the Beagle Boys. The following search command checks whether such an undesired state is reachable from `intruderInit`:

```
Maude> (search [1] intruderInit =>*
                   C:Configuration
                   < "Bank" : Responder | respSessions :
                       trustedConnection("Scrooge") RS:RespSessions > .)
```

This Maude search actually finds such a bad state where the Bank thinks that is has a connection with Scrooge:

```
Solution 1
...
```

Maude can then output the path leading from the initial state to this bad state, and this behavior indeed corresponds to a real attack on NSPK.

System Requirements (1 Lecture). Whereas up to this point, the course has dealt with formalizing the *behaviors* of the system, this and the following chapter deals with the *requirements* that the system should satisfy. I first introduce state-based and action-based properties, and then classes of properties, such as invariants, reachability, "guarantee" ("something good must eventually happen"), response properties, stability, and so on. I also discuss fairness assumptions, which are often needed to have liveness/guarantee properties, and how invariants can be proved inductively.

Formalizing and Model Checking Requirements Using Temporal Logic (1 Lecture). Maude is equipped with a linear temporal logic (LTL) model checker. Atomic propositions are terms of sort `Prop`, and LTL formulas are constructed (as terms of sort `Formula`) in the usual way. One chapter of the book introduces LTL and Maude's LTL model checker, and explains how various requirements, including fairness assumptions, can be formalized in LTL, and how crucial requirements of the distributed algorithms in the book can be analyzed.

Example 6. Consider the token-ring mutual exclusion algorithm in Example 5. The key liveness property we want to prove is that each node executes in its critical section infinitely often. This cannot be proved using search, but can easily be done using LTL model checking. We define a parametric atomic proposition `inCS(`o`)` to hold if node o is currently executing inside its critical section:

```
(omod MODEL-CHECK-MUTEX is protecting INITIAL . including MODEL-CHECKER .
  subsort Configuration < State .
  op inCS : Oid -> Prop [ctor] .
  var REST : Configuration .  var S : Status .  var O : Oid .
  eq REST < O : Node | status : S > |= inCS(O) = (S == insideCS) .
endom)
```

We check if each node in `init` executes infinitely often in its critical section:[5]

```
Maude> (red modelCheck(init,  ([] <> inCS(a))  /\  ([] <> inCS(b))  /\
                              ([] <> inCS(c))  /\  ([] <> inCS(d))) .)

result ModelCheckResult : counterexample(...)
```

[5] '`[]`' and '`<>`' denote the temporal operators $\Box$ and $\Diamond$, respectively, and '`/\` and '`->`' denote logical conjunction and implication.

The property does not hold: the model checker returns a counterexample where node `d` never starts waiting to enter its critical section. We therefore add the following *justice fairness* assumption for the first rule: *for each node o*, if, from some point on, the first rule is continuously enabled for *o* (that is, *o*'s `status` is `outsideCS`), then the first rule must also be taken infinitely often for *o* (i.e., *o*'s `status` must be `waitForCS`). We add the following declarations to the above module to define the formula `justAll` that encodes this justice assumption:

```
  ops waiting outside : Oid -> Prop [ctor] .
  eq REST < O : Node | status : S > |= waiting(O) = (S == waitForCS) .
  eq REST < O : Node | status : S > |= outside(O) = (S == outsideCS) .
  op just : Oid -> Formula .
  op justAll : -> Formula .
  eq just(O) = (<> [] outside(O)) -> ([] <> waiting(O)) .
  eq justAll = just(a) /\ just(b) /\ just(c) /\ just(d) .
```

We can check whether the justice fairness assumption `justAll` implies the desired property:

```
Maude> (red modelCheck(init, justAll ->
                                   (([] <> inCS(a)) /\ ([] <> inCS(b)) /\
                                    ([] <> inCS(c)) /\ ([] <> inCS(d)))) .)

result Bool :  true
```

□

Real-Time and Probabilistic Systems (Not Taught). Up to this point, the models have been *untimed*. However, these days the *performance* of a system is also an important metric, whose analysis requires modeling *time*. Furthermore, fault-tolerant systems must detect message losses and node crashes, which is impossible in untimed asynchronous distributed systems. Therefore, most larger system these days are *real-time* systems, whose modeling and analysis in Maude is supported by the Real-Time Maude tool [36,38]. The course textbook briefly introduces how real-time systems can be modeled and analyzed in Maude, and also mentions timed extensions of temporal logics.

Randomized simulations, such that those performed simulating playing *blackjack* with each card drawn pseudo-randomly, do not provide performance estimates with mathematical guarantees. I need more solid guarantees to quit my day job and move to Las Vegas. My textbook therefore indicates how probabilistic systems can be modeled in rewriting logic as *probabilistic rewrite theories* [2]. Such probabilistic models can then be subjected to *statistical model checking* (SMC) using Maude-connected tools such as PVeStA [3] and MultiVesta [51], which estimate the expected value of a path expression up to certain confidence intervals. Although, in contrast to precise *probabilistic model checking*, SMC does not give absolute guarantees, it is considered to be a *scalable* formal method, which, since it is based on simulating single paths until the desired confidence level has been reached, can be easily parallelized.

PVeStA analysis showed that if I start with $1000 and play 20 $100-rounds of blackjack, then with 99% statistical confidence, I am expected to walk home with between \$875 and \$877, and that the expected probability that I can walk out of the casino with \$1200 or more is a promising 31%.

In contrast to the other chapters in the book, the book only gives a flavor of these subjects, and does not give details about how to run Real-Time Maude or PVeStA. I have sometimes taught this part to fourth-year students, but do not currently teach it to second-year students.

Using Maude on Cloud Systems and the Use of Formal Methods at Amazon (1 Lecture). To give students the impression that Maude can be applied to analyze industrial designs, in the last lecture I give an overview of the use of Maude (and PVeStA) to model and analyze both the correctness and performance of cloud transaction systems such as Google's Megastore (which runs, e.g., Gmail and Google AppEngine), Apache Cassandra (developed at Facebook and used by, e.g., Amadeus, CERN, Netflix, Twitter), and the academic P-Store design, as well as our own extensions of these designs (see [6] for an overview).

The last lecture should summarize the course: What have you learnt? What is it useful for? Instead of singing the praises of formal methods myself, I summarize the course by quoting the experiences of engineers at Amazon Web Services, who used formal methods while developing their *Simple Storage System* and *DynamoDB* data store, which are key components of Amazon's profitable cloud computing business (which is much more profitable than Amazon's retail business). The engineers at Amazon used Lamport's TLA+ formalism with its model checker TLC. They report that formal methods have been a big success at Amazon, and describe their experiences in the previously mentioned paper "How Amazon Web Services Uses Formal Methods" [35] as follows:

- Formal methods found serious "corner case" bugs in the systems that were not found with any other method used in industry.
- A formal specification is a valuable precise *description* of an algorithm, which, furthermore, can be directly tested.
- Formal methods can be learnt by engineers in short time and give good return on investment.
- Formal methods makes it easy to quickly explore design alternatives and optimizations.

It is worth remarking that both the TLA+ efforts at Amazon and Maude as taught in this course use *model checking*. There is no evidence that Amazon formally verified their algorithms: model checking gave them enough confidence.

My textbook does not contain a chapter on the topics covered in this lecture.

6 Evaluation

The fact that I think that the course described above *should* be fun is irrelevant. What do the students think? Unfortunately, I have not solicited their feedback. Ideally, I should have asked: all students in the "Programming" program who did not take the course why they did not take it; all students who signed up for the course but did not finish it why they did not finish it; and all students who did finish it what they thought about the course.

Instead, at the end of each semester, the department sends an email to all students, making them aware of the possibility of providing feedback to courses signed up for. Most students typically do not bother to do this. Therefore, although I am trying to summarize the students' experiences the best I can, this evaluation is unscientific and anecdotal.

In addition to the random collection of students who answer the call to provide course feedback in the middle of the summer, there are many other variables as well, such as the quality of the lecturer and the TA, time of lectures (avoid Friday afternoons!), pandemics, and so on. *Bonus tip:* Giving good grades to many students seems to improve student satisfaction.

The course has changed *a lot* since its embryonic first version was given in 2002, but has stabilized since the textbook was published in 2018. Until 2018, it was taught to third-year and fourth-year students. In 2019 and 2020 it was taken by second-year students.

6.1 Summary of Student Feedback

I have gathered anonymous student feedback, administered by the department, from 2007. In general, only 10%–15% of the invited students submit responses, and those include students who quit the course during the semester.

The following tables show the cumulated response to the all-important questions "How do you rate this course in general?"[6] and "How do you rate the level (difficulty) of the course?" Since 2019 was the first time the course was given at the second-year level, I also show the results from 2019 in separate columns.

How do you rate this course in general?		
	2007–2019	2019
Exceptionally good	15	4
Very good	23	3
Good	8	0
OK (neither good nor bad)	6	1
Not that good	1	0
Not good	0	0

Difficulty/level of the course		
	2007–2019	2019
Too difficult	1	0
Somewhat difficult	38	4
OK/Average	38	4
Easy	0	0
Too easy	0	0

[6] This general question did not appear in the evaluation form the first couple of years.

An overwhelming majority (75–80%) of the student report that the workload is "OK" (or average) for the number of credits (10) given.

Oddly enough, none of the 30 questions in the 2018 and 2019 evaluation forms concerned the quality of the course textbook, so I cannot report on the students' impressions of my book.

6.2 Selected Student Comments

The evaluation form allows students to comment on the course in free-text. Below I quote some student opinions about the course content from 2015 to 2019. What students liked about the course:

- "Very interesting course where we learnt a lot. A unique course at the bachelor level in informatics in Norway."
- "Different and powerful method for system analysis. Creative textbook."
- "Learn a different kind of programming language. Learn about algorithms, and how to model them to check security vulnerabilities. After finishing the course you have relevant knowledge that some of the world's leading companies are looking for."
- "Programming was fun."
- "Introduction to a different programming paradigm."
- "Interesting, but not too extensive, curriculum."
- "Fun curriculum."
- "Course content."
- "IN2100 is the best course I have taken at the University of Oslo. [...] The funniest lecturer in Norway."
- "Showed the importance of the topic."
- "Interesting topic."
- "It allows to develop complex systems, and test safety and security of critical systems as well."
- "Strong foundations, applicable to real systems, useful for developing robust systems."
- "All in all I think this was a very fun course, clearly one of those I remember the most from my bachelor. Maude essentially worked well, and even though I don't think that I will ever use it after the course, I have learnt a lot by using it."
- "I did not choose this course [...] but I loved every week and content."
- "The assignments are really well balanced between theory and the entertaining Maude programming parts."

What the students liked less:

- "Language that is not used much or at all."
- "Course might be difficult for many of us."
- "Need more real world critical systems for analysis. [...] Lack of applicability in industry."

Other complaints concern Full Maude and its “peculiarities” (lack of robustness and good error messages) and that there are not too many resources about Maude. From earlier years, I also remember complaints about Full Maude, and, as always, a number of students do not understand why they need to learn a programming language that is not widely used.

6.3 Other Issues

Temporal Logic. I was afraid that introducing temporal logic to second-year students is recipe for a disaster, especially since only one lecture is devoted to the topic (and one lecture is devoted to classes of requirements). I am very surprised to observe that students seem to master temporal logic pretty well: Their exam solutions show that they understand temporal logic formulas and can judge whether such a formula holds in a system.

Industrial Impact. I have no idea whether the students who have taken the course will ever use Maude or formal methods again. What I know is that two former students and TAs in my course started a company based on a product programmed in Maude. That company is still doing well after 15 years, and sometimes hires my better master’s students.[7] Another alumnus of the course started a company on security analysis using Maude a few years ago; I believe that the company still exists.

Popularity of the Course. I have discussed in Sect. 2 the difficulties of attracting students to formal methods courses. In 2019, when the course for the first time became a fourth-semester course, and one of three elective courses that semester, around 20 students took the exam. This year, 48 students finished all three mandatory assignments, and 42 students submitted solutions to the exam.

Level. As mentioned, until 2018, the course was a third/fourth-year course. The move to a second-year course in 2019 was risky, also since my textbook had then been published and I could therefore not simplify it much (if replacing simplification orders and Turing machines with temporal logic counts as simplification). My experiences so far are positive. The grades in 2019 were significantly better than most years, although I think that the exam might have been slightly easier. The students follow the course very well, and, if anything, seem more enthusiastic than their older precursors. I am so far very happy with my decision to move the course down to the fourth semester.

6.4 Weaknesses

The course has a number of weaknesses. First and foremost, although I still teach Full Maude for its interface that supports elegant modeling of object-oriented

[7] Coincidentally, my son’s teacher recommended me to use their Maude product to teach my son mathematics during the home schooling caused by the corona virus.

systems, Full Maude *is* frustrating, with its lack of (informative) error messages and its lack of robustness. This makes even small modeling tasks a frustrating experience for the students. Most people working with objects in Maude therefore do it all at the (core) Maude level, which requires cluttering the rewrite rules with variables capturing the "remaining attributes" of the objects in the rewrite rules, and which makes it much harder to use subclasses.

Another issue is that Maude, at least as taught in the course, relies on explicit-state model checking. Even though the state space is significantly reduced by the fact that the states are E-equivalence classes of terms modulo the equational theory E (or, equivalently, the states are E-normal-forms), such explicit-state model checking nevertheless encounters the state space explosion problem pretty early. In this course, with its small- and medium-sized models and modest initial states, this is not a significant problem. I actually *want* the students to experience having to wait a few minutes for a (model checking) execution to end, which I do not think they have experienced before.

Every formal methods researcher who reads this paper will miss a lot of her favorite things in the course. Notable omissions include: SMT solving and symbolic methods, higher-order logics, and tool-assisted theorem proving.

The course focuses on modeling and analyzing *designs*, and does not discuss software/code analysis. However, as mentioned above, Maude and Grigore Rosu's rewriting-logic-based K framework have been used to provide the most complete formal semantics of languages like C and Java, and have successfully been applied to verify source and virtual machine code.

7 Concluding Remarks

Although the value of formal methods for mainstream software development is increasingly realized in industry, trying to introduce formal methods to undergraduate students is challenging. The main challenges, I believe, is that students consider computer science education as job training instead of as a science, and therefore prefer more "practical" courses (computer networks, security, machine learning, databases, software engineering, ...), and that our colleagues do not see the need for something they think "requires huge effort to verify a tiny piece of straight-forward code" and therefore relegate formal methods to the hidden corners of course plans—far away from the mandatory courses—where they have to compete with sexy-sounding topics for the few spare slots available. In the face of these challenges, the best approach to make students take formal methods courses is to make them "fun," motivating, and industry-relevant.

In this paper, I have distilled some requirements for an undergraduate course introducing formal methods in a fun and motivating way. Some of these are: use few, but simple yet expressive and executable, formalisms; study relevant and motivating problems, for example from other CS courses; focus on automatic analysis; and demonstrate industrial relevance.

When I was an undergraduate student, I thought that functional programming was the most "fun" style of programming. I therefore suggest rewriting logic, with its fairly mature simulation and model checking tool Maude, as a suitable formal method for an introductory formal methods course. What is unique about Maude compared to other formalisms used for formal methods education (such as different kinds of transition systems, (timed) automata, functional programming, HOL/Coq/Isabelle, Z, B, Event-B, Hoare logic or other logics on imperative programs, and so on[8]) is the combination of:

- (modeling in a) functional programming (style),
- object-based executable modeling,
- focus on distributed systems, and
- model checking.

Thanks to the intuitive and expressive formalism, even in my fourth-semester undergraduate course, students can model and analyze a wide range of key distributed algorithms in computer science. I give an overview of that course and its accompanying textbook [42] in this paper.

Exam results show that second-year students indeed can formally model and analyze textbook cryptographic protocols, transport protocols, and distributed mutual exclusion and leader election algorithms. What surprises me more is that they also understand temporal logic formulas quite well. I have summarized students' feedback to the course, since I believe that the only way to attract students to study formal methods, unless it is made mandatory, is the hard way: by word-of-mouth from student to student. Preliminary results are promising: 42 students took the exam in 2020, which is almost twice as many as in 2019.

Acknowledgments. I am grateful to Antonio Cerone and Markus Roggenbach for inviting me to give a talk at FMfun 2019, and for patiently waiting for this paper to be finished.

References

1. Aceto, L., Ingólfsdóttir, A., Larsen, K.G., Srba, J.: Teaching concurrency: theory in practice. In: Gibbons, J., Oliveira, J.N. (eds.) TFM 2009. LNCS, vol. 5846, pp. 158–175. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-04912-5_11
2. Agha, G.A., Meseguer, J., Sen, K.: PMaude: rewrite-based specification language for probabilistic object systems. Electr. Notes Theor. Comput. Sci. **153**(2), 213–239 (2006)
3. AlTurki, M., Meseguer, J.: PVeStA: a parallel statistical model checking and quantitative analysis tool. In: Corradini, A., Klin, B., Cîrstea, C. (eds.) CALCO 2011. LNCS, vol. 6859, pp. 386–392. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22944-2_28
4. Anastasio, T.J.: Computer modeling in neuroscience: from imperative to declarative programming: Maude modeling in neuroscience. In: Martí-Oliet, N., Ölveczky, P.C., Talcott, C. (eds.) Logic, Rewriting, and Concurrency. LNCS, vol. 9200, pp. 97–113. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-23165-5_4

[8] See https://fme-teaching.github.io/courses/ for a list of formal methods courses.

5. Bentea, L., Ölveczky, P.C., Bentea, E.: Using probabilistic strategies to formalize and compare α-synuclein aggregation and propagation under different scenarios. In: Gupta, A., Henzinger, T.A. (eds.) CMSB 2013. LNCS, vol. 8130, pp. 92–105. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-40708-6_8
6. Bobba, R., et al.: Survivability: design, formal modeling, and validation of cloud storage systems using Maude. In: Assured Cloud Computing, chap. 2, pp. 10–48. Wiley-IEEE Computer Society Press (2018)
7. Bogdănaş, D., Roşu, G.: K-Java: a complete semantics of Java. In: Proceedings of POPL 2015. ACM (2015)
8. Broccia, G., Milazzo, P., Ölveczky, P.C.: Formal modeling and analysis of safety-critical human multitasking. Innovations Syst. Softw. Eng. **15**(3–4), 169–190 (2019)
9. Bruni, R., Meseguer, J.: Semantic foundations for generalized rewrite theories. Theoret. Comput. Sci. **360**(1–3), 386–414 (2006)
10. Cerone, A., Roggenbach, M., Schlingloff, H., Schneider, G., Shaikh, S.: Teaching formal methods for software engineering - ten principles. In: Proceedings of Fun With Formal Methods (a CAV 2013 Workshop) (2013)
11. Cerone, A.: A cognitive framework based on rewriting logic for the analysis of interactive systems. In: De Nicola, R., Kühn, E. (eds.) SEFM 2016. LNCS, vol. 9763, pp. 287–303. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-41591-8_20
12. Clarke, E., Grumberg, O., Peled, D.A.: Model Checking. MIT Press, Cambridge (1999)
13. Clavel, M., et al.: All About Maude - A High-Performance Logical Framework. LNCS, vol. 4350. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-71999-1
14. Clavel, M., et al.: Maude Manual (Version 3.0) (2020). http://maude.cs.illinois.edu
15. Curzon, P., McOwan, P.W.: Teaching formal methods using magic tricks (2013). Paper presented at the Workshop "Fun with formal methods" at CAV 2013
16. Durán, F., et al.: Programming and symbolic computation in Maude. J. Log. Algebr. Meth. Program. **110**, 100497 (2020)
17. Ellison, C., Rosu, G.: An executable formal semantics of C with applications. In: Proceedings of POPL 2012. ACM (2012)
18. Grov, J., Ölveczky, P.C.: Formal modeling and analysis of Google's Megastore in Real-Time Maude. In: Iida, S., Meseguer, J., Ogata, K. (eds.) Specification, Algebra, and Software. LNCS, vol. 8373, pp. 494–519. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-642-54624-2_25
19. Grov, J., Ölveczky, P.C.: Increasing consistency in multi-site data stores: Megastore-CGC and its formal analysis. In: Giannakopoulou, D., Salaün, G. (eds.) SEFM 2014. LNCS, vol. 8702, pp. 159–174. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-10431-7_12
20. Kasampalis, T., et al.: IELE: a rigorously designed language and tool ecosystem for the blockchain. In: ter Beek, M.H., McIver, A., Oliveira, J.N. (eds.) FM 2019. LNCS, vol. 11800, pp. 593–610. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-30942-8_35
21. Katelman, M., Meseguer, J., Hou, J.: Redesign of the LMST wireless sensor protocol through formal modeling and statistical model checking. In: Barthe, G., de Boer, F.S. (eds.) FMOODS 2008. LNCS, vol. 5051, pp. 150–169. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-68863-1_10
22. Krings, S., Körner, P.: Prototyping games using formal methods. In: Proceedings of FMfun 2019. CCIS, Springer, pp. 124–142 (2020)
23. Lien, E., Ölveczky, P.C.: Formal modeling and analysis of an IETF multicast protocol. In: Proceedings of SEFM 2009. IEEE Computer Society (2009)

24. Liu, S., Takahashi, K., Hayashi, T., Nakayama, T.: Teaching formal methods in the context of software engineering. ACM SIGCSE Bull. **41**(2), 17–23 (2009)
25. Liu, S., Ganhotra, J., Rahman, M.R., Nguyen, S., Gupta, I., Meseguer, J.: Quantitative analysis of consistency in NoSQL key-value stores. LITES **4**(1), 03:1–03:26 (2017)
26. Liu, S., Sandur, A., Meseguer, J., Ölveczky, P.C., Wang, Q.: Generating correct-by-construction distributed implementations from formal Maude designs. In: Lee, R., Jha, S., Mavridou, A., Giannakopoulou, D. (eds.) NFM 2020. LNCS, vol. 12229, pp. 22–40. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-55754-6_2
27. Lutz, R.R.: Analyzing software requirements errors in safety-critical embedded systems. In: IEEE International Symposium on Requirements Engineering, San Diego, CA, pp. 126–133, January 1993
28. Meseguer, J.: Conditional rewriting logic as a unified model of concurrency. Theor. Comput. Sci. **96**, 73–155 (1992)
29. Meseguer, J.: Membership algebra as a logical framework for equational specification. In: Presicce, F.P. (ed.) WADT 1997. LNCS, vol. 1376, pp. 18–61. Springer, Heidelberg (1998). https://doi.org/10.1007/3-540-64299-4_26
30. Meseguer, J., Rosu, G.: The rewriting logic semantics project. Theor. Comput. Sci. **373**(3), 213–237 (2007)
31. Meseguer, J.: Twenty years of rewriting logic. J. Log. Algebraic Methods Program **81**(7–8), 721–781 (2012)
32. Meseguer, J., Roşu, G.: The rewriting logic semantics project: a progress report. Inf. Comput. **231**, 38–69 (2013)
33. Meseguer, J., Sasse, R., Wang, H.J., Wang, Y.: A systematic approach to uncover security flaws in GUI logic. In: 2007 IEEE Symposium on Security and Privacy (S&P 2007). IEEE Computer Society (2007)
34. Moller, F., O'Reilly, L., Powell, S.: Teaching them early: formal methods in school. In: Proceedings of FMfun 2019. CCIS, Springer, pp. 173–190 (2020)
35. Newcombe, C., Rath, T., Zhang, F., Munteanu, B., Brooker, M., Deardeuff, M.: How Amazon Web Services uses formal methods. Commun. ACM **58**(4), 66–73 (2015)
36. Ölveczky, P.C.: Real-Time Maude and its applications. In: Escobar, S. (ed.) WRLA 2014. LNCS, vol. 8663, pp. 42–79. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-12904-4_3
37. Ölveczky, P.C., Meseguer, J.: Specification of real-time and hybrid systems in rewriting logic. Theor. Comput. Sci. **285**, 359–405 (2002)
38. Ölveczky, P.C., Meseguer, J.: Semantics and pragmatics of Real-Time Maude. High. Order Symb. Comput. **20**(1–2), 161–196 (2007)
39. Ölveczky, P.C., Meseguer, J., Talcott, C.L.: Specification and analysis of the AER/NCA active network protocol suite in Real-Time Maude. Formal Methods Syst. Des. **29**(3), 253–293 (2006)
40. Ölveczky, P.C., Thorvaldsen, S.: Formal modeling, performance estimation, and model checking of wireless sensor network algorithms in Real-Time Maude. Theor. Comput. Sci. **410**(2–3), 254–280 (2009)
41. Ölveczky, P.C.: Design and validation of cloud storage systems using formal methods. In: Mousavi, M.R., Sgall, J. (eds.) TTCS 2017. LNCS, vol. 10608, pp. 3–8. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-68953-1_1
42. Ölveczky, P.C.: Designing Reliable Distributed Systems: A Formal Methods Approach Based on Executable Modeling in Maude. Undergraduate Topics in Computer Science. Springer, London (2017). https://doi.org/10.1007/978-1-4471-6687-0

43. Park, D., Zhang, Y., Saxena, M., Daian, P., Roşu, G.: A formal verification tool for Ethereum VM bytecode. In: Proceedings of ESEC/FSE 2018, pp. 912–915. ACM (2018)
44. Rocha, C., Cadavid, H., Muñoz, C., Siminiceanu, R.: A formal interactive verification environment for the Plan Execution Interchange Language. In: Derrick, J., Gnesi, S., Latella, D., Treharne, H. (eds.) IFM 2012. LNCS, vol. 7321, pp. 343–357. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-30729-4_24
45. Rocha, C., Meseguer, J.: Proving safety properties of rewrite theories. In: Corradini, A., Klin, B., Cîrstea, C. (eds.) CALCO 2011. LNCS, vol. 6859, pp. 314–328. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22944-2_22
46. Roşu, G.: Matching logic. Logical Methods Comput. Sci. **13**(4), 1–61 (2017)
47. Roşu, G., Şerbănuţă, T.F.: An overview of the K semantic framework. J. Logic Algebraic Program. **79**(6), 397–434 (2010)
48. Rushby, J.: Mechanized formal methods: progress and prospects. In: Chandru, V., Vinay, V. (eds.) FSTTCS 1996. LNCS, vol. 1180, pp. 43–51. Springer, Heidelberg (1996). https://doi.org/10.1007/3-540-62034-6_36
49. Rushby, J.M.: New challenges in certification for aircraft software. In: Proceedings of EMSOFT 2011. ACM (2011)
50. Schlingloff, H.: Teaching model checking via games and puzzles. In: Proceedings of FMfun 2019. CCIS, Springer, pp. 143–158 (2020)
51. Sebastio, S., Vandin, A.: Multivesta: statistical model checking for discrete event simulators. In: ValueTools, pp. 310–315. ICST/ACM (2013)
52. Skeirik, S., Stefanescu, A., Meseguer, J.: A constructor-based reachability logic for rewrite theories. In: Fioravanti, F., Gallagher, J.P. (eds.) LOPSTR 2017. LNCS, vol. 10855, pp. 201–217. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-94460-9_12
53. Spichkova, M., Zamansky, A.: Teaching of formal methods for software engineering. In: Proceedings of ENASE 2016. SciTePress (2016)
54. Talcott, C.L.: The Pathway Logic formal modeling system: diverse views of a formal representation of signal transduction. In: Proceedings of IEEE International Conference on Bioinformatics and Biomedicine, BIBM 2016. IEEE Computer Society (2016)
55. Wing, J.M.: Weaving formal methods into the undergraduate computer science curriculum (extended abstract). In: Rus, T. (ed.) AMAST 2000. LNCS, vol. 1816, pp. 2–7. Springer, Heidelberg (2000). https://doi.org/10.1007/3-540-45499-3_2

Fun with Formal Methods for Better Education

Nikolay V. Shilov(✉), Evgeniy Muravev, and Svetlana Shilova

Innoplolis University, 1, Universitetskaya Str., Innopolis 420500, Russia
`shiloviis@mail.ru`

Abstract. Is there a need to popularize Formal Methods in Software Engineering? Maybe industrial demand in Formal Methods is the best way to explain their utility and importance? – We try to argue educational and emotional role of popularization for a better comprehension and a positive attitude to Formal Methods and discuss several Math Olympiad problems that can be solved using Formal Methods (while, unfortunately, Mathematical education suffers of lack of Theoretical Computer science curricular) .

Keywords: Formal Methods · Popularization · Puzzles · Gamification · Education · Mathematical Olympiad and contests · Ontological argument · Fibonacci words · Operational, denotational and axiomatic semantics · Esoteric languages · Recursion elimination · Algorithm transformation · Partial and total correctness

1 Introduction

1.1 A Semiannual Anniversary

Fifty two years have passed since Robert Floyd had published a paper Assigning Meaning to Programs, a pioneering research on Formal Methods [6], fifty – since C.A.R. Hoare published a paper *An axiomatic basis for computer programming* [10], the first paper on axiomatic of program correctness. During these years people frequently questioned the efficiency, the utility, the industrial strength, the educational value, the understandability of Formal Methods (FM).

For example, in 2010 David Parnas published a very polemical article *Really Rethinking "Formal Methods"* [20]; in particular, he wrote in the article that there are much more FM academic experts than industrial developers using FM, and analyzed the reasons why FM have not became a common practice in Software Engineering.

We believe that this sad picture is not true. Indeed, ACM Turing Prize in 2007 was awarded to Edmund M. Clarke, E. Allen Emerson and Joseph Sifakis for *their role in developing Model-Checking into a highly effective verification technology that is widely adopted in the hardware and software industries* [29]. Later in years 2007-12 model checking was successfully used for verification of on-board software of Mars-rover *Curiosity* [11].

A. Cerone and M. Roggenbach (Eds.): FMFun 2019, CCIS 1301, pp. 92–105, 2021.
https://doi.org/10.1007/978-3-030-71374-4_4

We also think that academic theory and/or industrial practice aren't the only dimensions for Formal Methods, there exists (at least) one more aspect—education. In the next paragraph we explain how we understand education and what role FM may/should play in Software Engineering education.

There exists an opinion (sometimes attributed to Karl Weierstrass) that *the education should bring up minds, not just trains skills.* Also we would like to quote an aphorism (commonly attributed to Mikhailo Lomonosov) that *mathematics should be learned just because it disciplines minds.* By citing these maxims we wouldn't like to claim that the purpose of education is bringing up minds, or that the utility of Mathematics is restricted by mind discipline. We just would like to emphasize a value of Formal Methods for Software Engineering education: to bring up and discipline minds of the future engineers and developers.

1.2 Fun for Better Education

A part of the reason why the Formal Methods moves slowly from the academy to industry is FM education, a "transmission": *some students consider FM too poor (inefficient), other students consider FM too pure.* We need to improve transmission, i.e. to improve FM education.

Ascending approach (from simple and easy to the most complex and complicated) is a common practice in education. Nobody starts teaching arithmetic with Peano axioms and formal derivation of formal statements like (for instance) addition associativity $\forall x.\forall y.\forall z \;\; : \;\; ((x+y)+z) \; = \; (x+(y+z))$; instead the education/teaching starts with elementary exercises/problems like the following: *Dad gave Peter 5 apples and then Peter passes 2 apples to a sister; how many apples Peter has after that?*

If educator would like to engage students with a topic then fun and amusement may be very important and helpful ingredients (maybe, a lubricating oil for the transmission). Even a simple joke can help! (For example, if you think that the answer for the above problem about apples is 3 then you are not right, because the correct answer depends on initial number of apples that Peter had before his Dad gave him these 5 apples;-)

Same should be true for FM education: it should start with simple and easy examples/problems, exploit jokes, fun and amusement for engagement and popularization. Many FM educators use ascending approach altogether with fun and amusement in their educational practice. – Just for example, a very concise, sound and comprehensive textbook [12] on model checking with SPIN is illustrated by many puzzles solved by model checking. (Of course, a renowned *Cabbage, Goat and Wolf* is one of the puzzles used in this book.) But we question is: how common is this practice to engage students with FM via fun, puzzles, games and entertainment?

1.3 FWFM Workshop Series

The primary purpose of the workshop series on *Fun With Formal Methods* (FWFM) was (and still is) to popularize and disseminate the best practice of popularization of Formal Methods. Not an exhaustive list of topics of FWFM follows:

- fascinating examples of use of FM in SE;
- simple but interesting educational examples of FM;
- FM for puzzles, games and entertainment;
- FM and programming contests and Olympiads;
- FM elsewhere (outside software and hardware);
- anything and everything related to popularization of FM.

History of the workshop is depicted in brief in the Fig. 1 and explained below in the next paragraphs.

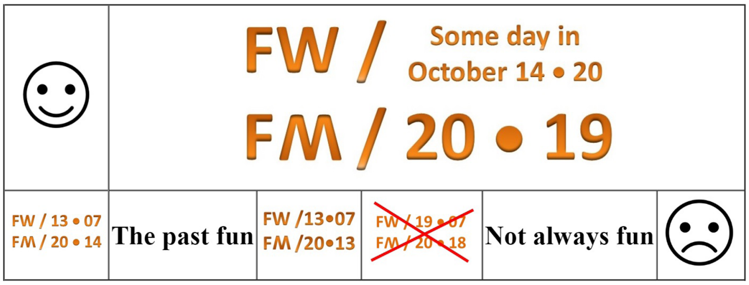


Fig. 1. History of the FWFM workshop series.

The workshops from FWFM series ware organized twice in the years 2013 [30] and 2014 [31] but (for fun!) the same day both times – July 13 – and both times— in affiliation with *the International Conference on Computer Aided Verification* (CAV). Both workshops were successful because of number of submissions, good quality of selected papers and a high attendance.

Then there was non-successful attempt to organize the third workshop in the year 2018 [32]. The attempt had the same affiliation (CAV), but was scheduled for another day then two previous workshops — July 19 instead of July 13. This shift was not fun, it has led to few submissions and cancellation of the workshop. – Of course, we are kidding about the role of the day and its influence on the number of submissions; maybe the main reason of the deficit of interest to the FWFM-2018 was publication policy of the first two workshops: no formal proceedings of the FWFM-2013 and FWFM-2014 have been published.

After an epic failure in 2018, we attempted to revive FWFM series in year 2019 [33]. Because of this intention we had decided to give up fixed affiliation (CAV), fixed day (July 13), and presentation "in person" (offline) policy. (By the way, this *online* move has been implemented months before the outbreak of COVID-19 [34] and hadn't been motivated by the epidemic situation. Maybe, it (the move) had much more in common with climate change concern recently stressed in [28].) In the year 2019 the workshop was organized as a satellite event of another conference *TOOLS 50+1: Technology of Object-Oriented Languages and Systems*, distance (online) presentations (via Skype) were allowed, and live streaming video of the workshop was organized and recorded [35]. – Maybe it is too early to say that the workshop was successful, but it is right time to say that we had the workshop and the FWFM series is alive!

The problem that FWFM-series needs to solve is a proper publication of the workshop proceedings. If not to solve but to compensate a publication deficit,we would like to present in brief in the next section the following 5 talks from FWFM-2013 and FWFM-2014:

– *The Ontological Argument in PVS: What Does This Really Prove?* (John Rushby, FWFM-2013);
– *Tackling Fibonacci words puzzles by finite countermodels* (Alexei Lisitsa, FWFM-2013);
– *Teaching Formal Methods using Magic Tricks* (Paul Curzon and Peter McOwan, FWFM-2013);
– *Chekofv: Crowd-sourced Formal Verification* (Heather Logas et al., FWFM-2014);
– *Using Esoteric Language for Teaching Formal Semantics* (Nikolay Shilov, FWFM-2014).

1.4 Structure of the Paper

In the next Sect. 2 we give a brief overview of selected talks from FWFM-2013 and FWFM-2014. Then in the Sect. 3 we just sketch the programme of FWFM-2019 but present (some) details solutions (with aid of Formal Methods) of two problems from the 60th International Mathematical Olympiad IMO-2019. Finally, we conclude in the last Sect. 4 by summing-up our arguments for popularization of Formal Methods and discussing challenges of further integration of Formal Methods and Artificial Intelligence – Computer Science in general – with Mathematics.

2 FWFM13-14 in Brief: From Ontological Argument to Esoteric Languages

2.1 The Ontological Argument in PVS

An *ontological argument* is a tradition to prove that God existence using ontology. One of known ontological arguments was formulated by Anselm of Canterbury in 1078 in work Proslogion. Anselm defined God as “that than which nothing greater can be thought”. He suggested that, if the greatest possible exists in the mind, it must also exists in reality and proved it by contradiction: if the greatest possible does exist just in the mind, then an even greater must exists in the mind—one which is greater both in the mind and in reality; therefore, this the greatest possible being must do exist in reality.

Please refer [21] for a formalization and verification of the Ontological Argument in PVS [36]. Although the formalization is consistent, the formal verification doesn't compel the Ontological Argument. The educational value of the formalization and verification of the Ontological Argument with PVS is an opportunity to use it as a case study in graduate programs at Philosophy and Humanities Departments for teaching automated theorem proving.

2.2 Countermodels for Fibonacci Words

An infinite sequence of Fibonacci words $w_0, w_1, \ldots$ is defined [17] very similar as the infamous sequence of Fibonacci numbers: let a and b be two distinguishable symbols; then $w_0 = b$, $w_1 = a$, and $w_{i+2} = w_i w_{i+1}$ for all $i \geq 0$. It is easy to see that the sequence of Fibonacci words stars as b; a; ba; aba; $baaba$; $ababaaba$; $baabaababaaba$.

One can observe that none of the first 7 Fibonacci words listed above contains two b's or three a's in a row (i.e. no sub-words bb or aaa). This observation leads to a hypothesis that all Fibonacci words contain neither two b's nor three a's in a row. But then the next question arises: how to prove (ore refute) the hypothesis?

A particular way to prove the hypothesis presented in [17] comprises two steps:

1. first-order sound axiomatization of algebraic systems (first-order models) where all elements of the domain may be generated using Fibonacci words
2. and then automatic generation of finite countermodels that meet the axiomatization but refute that some element may be generated using two b's or three a's in a row.

Surprisingly, the countermodels for each of these properties are quite small,—just 5 elements to refute a possibility of two b's and 11 element to refute a possibility three a's in a row [17].

The educational value of the case-study is popularization of non-standard models for proving properties of the standard ("default" or assumed) models and tools like finite model generators for first-order theories.

2.3 Learning Loop Invariant via Card Magic

After great publications by Martin Gardner like *Mathematics, Magic, and Mystery* (1956) or *Mathematical Puzzles* (1961), it is hard to engage magic tricks with any other discipline than Mathematics. But still many magic tricks are much more dynamic and algorithmic in nature than static and Mathematical. Hence many magic tricks can/may be used to teach Computer Science and Formal Methods.

Some examples of engagement of card magic with CS and FM can be found in paper [1] that summarizes some experience accumulated in the science-popular project *cs4fn* (Computer Science for Fun) [37]. Below we present in brief one example of a card magic (borrowed from [1]) and discuss educational value of the example.

1. Take 10 cards consisting of a series of 5 cards of a suit followed by the same 5 cards of a different suit placed in the same order.
2. Fan the cards to show that you have a mix of cards and then turn the pack over, face down and ask a volunteer to touch the back of any card. Cut the pack at this point, putting the top half to the bottom and fan the cards again. Repeat this several times until the audience becomes happy that the cards are sufficiently mixed.
3. Count out 5 cards into a pile on the table, reversing their order as you do so. Place the remaining 5 cards straight down to make a second pile (non-reversed).
4. Give a volunteer 4 coins and ask to put each down on one of the two piles (i.e. the volunteer may spread coins between the two piles arbitrary). Once coins are placed you now do the same number of moves on a pile as the number of coins on the pile. (A move consists just of moving a card from the top to the bottom of the pile.) Then take the resulting top card of each pile and place them together (face down) at the side together with one coin (from 4 that you use). Repeat the same with the remaining 3 (instead of 4) coins and remaining two piles (each with 4 instead of 5 cards), then – with 2 coins and piles with 3 cards each, and finally – with the last coin and piles with 2 cards each.
5. Turn over all pairs of cards and demonstrate to the audience that cards in pairs match each other!

The correctness of the magic trick can be explained in terms of partial correctness of the non-deterministic algorithm presented above using pre-conditions, invariants and post-conditions:

– Precondition of the first non-deterministic loop on step 2 is formulated in step 1: *the first 5 cards of one suit are followed by the same 5 cards of another suit in the same order.*
– The invariant and the post-condition of the first loop is very similar to the pre-condition: *the first 5 cards are followed by the same 5 cards in the same order.*
– Pre-condition for the loop on steps 4 results from post-condition for step 2 after implementing step 3: *the order of 5 cards in the first pile is reverse of the order of the 5 cards in the second pile.*
– The invariant of the second loop is closely related to the pre-condition: *the order of cards in the first pile is reverse of the order of the cards in the second pile and cards in pairs that are put aside match each other.*
– The post-condition is what we want to demonstrate to the audience: *cards in pairs match each other.*

So, Formal Method's classics is a magic!

2.4 Gamification and Crowd-Sourcing Loop Invariants

Chekofv [18,19] is a system for crowd-sourced formal verification. It starts with an attempt to verify a given C program using the source code analysis platform

Frama-C. Every time the analysis needs a loop invariant (like in the previous subsection) *Chekofv* translates the problem into a puzzle game *Xylem* and presents it to players.

Xylem [19] is an iPad game where players make mathematical observations about synthetic plants, which are turned into predicates used for the construction of loop invariants. The game is a logical induction puzzle game where the player plays a botanist exploring and discovering new forms of plant life on a mysterious island. The player observes patterns in the way a plant grows, and then constructs mathematical equations to express the observations. These equations are considered as candidates for loop invariants and must be verified by any proof-assistance (PVS in particular).

2.5 Formal Semantics Though an Esoteric Language

Teaching different types of formal semantics (at undergraduate level especially) is not a trivial task. A gentleman's set should include some variants of operational, denotational and axiomatic semantics. A common approach to teaching the topics consists in use of toy programming languages. Instead, [23,24] presented an approach with use of an esoteric language [38].

Every language (artificial or natural) may be characterized by its syntax, semantics, and pragmatics. For example, in one of the 56 Sherlock Holmes short stories, *The Adventure of the Dancing Men*, written by Arthur Conan Doyle, Mr. Hilton Cubitt gives Sherlock Holmes a piece of paper with this mysterious sequence of stick figures of dancing men that had driven driving his young wife Elsie to distraction. Holmes realizes that it is a substitution cipher, cracks the code by frequency analysis and realizes that the syntax was just as in English with dancing men instead of letters, the semantics was provided by transformation to English, pragmatics (usage) of the language was to serve as a cryptography for Chicago gangsters.

Toy Esoteric Language (TEL) is not a programming language at all since it is not design for data processing. Its pragmatics is to introduce and explain what different types of formal semantics are, namely: what are operational, denotational, axiomatic, second-order and game semantics and how they may relate to each other. TEL syntax is easy to explain: correct words look like bodies of structured Pascal programs (with integer variables exclusively). TEL informal semantics can be defined as follows. Since every correct TEL word looks like an iterative program, one can draw a flowchart of this program. Every flowchart is a graph with assignments and conditions as nodes and control passing as edges. Let us count length of a path between nodes in a flowchart by number of assignments in this path (i.e. we do not count conditions at all). Then semantics of a correct TEL "program" is the shortest length of a path through the corresponding flowchart (i.e. from start to finish).

3 How FM and Math Can Help and Boost Each-Other?

3.1 FWFM-2019 in Brief

The programme of FWFM-2019 [33,35] comprises the following 5 talks:

1. *Do we need Fun with Formal Methods?* (Nikolay Shilov and Evgeiy Muravev);
2. *Cables, Trains and Types* (Simon Gay);
3. *On programming content of Math contests* (Nikolay Shilov and Svetlana Shilova);
4. *Towards a Broader Acceptance of Formal Varication Tools* (Mansur Khazeev et al.);
5. *Fun with Formal Methods: teaching unambiguous English to avoid confusions* (Maya Stoyanova).

The first talk presented a tool to play with the axiomatic semantics for the esoteric language TEL from papers [23,24].

The second talk presented a dependent type system for cables and toy railroad, a background paper is available [8] and is under publication right now.

The third talk addressed relations between Mathematics and Theory of Programming, its content has not been published anywhere else and because of it is discussed in the following subsections of this paper.

The forth talk presented results of in-class study how a small group of master students in Software Engineering accept formal verification tools in general and *AutoProof* system [7] in particular, a background paper has been published in arXiv [13].

The last talk was English-language experience report how to teach future software developers and engineers to speak and write in unambiguous way (especially when it concerns technical writing and specifications in English).

The workshop FWFM-2019 was concluded by a discussion moderated by Hamna Aslam. The topics of the discussion included (but were not limited by) the following questions:

– Who should not be teaching Math & FM?
– Which Math & FM book(s) are not recommended to be proposed as a textbook for freshmen?
– How to promote Math & FM group learning among students?
– How to teach Math & FM to students in their language?

(The purpose of the "negative" questions wasn't to exclude someone or some book as but to learn student's opinions about a "good" and a "bad" education practice.)

3.2 On Relations Between Program Theory and Mathematics

A discourse about historical, cultural, educational relations and connections between Mathematics and Science and Art of Programming (exactly Programming not Computer Science) is quite old: it originated in early days of computing machinery more than 70 years ago (since, at least, since ENIAC was completed and first put to work in 1945). Many programming pioneers—e.g. Edsger W. Dijkstra, Andrey P. Ershov, Donald E. Knuth—had published their reflections on this topic [2–4,14]. (Unfortunately, we are not aware about reflections of mathematicians on this topic while we know and highly recommend a book of outstanding Russian mathematician Vladimir A. Uspensky [27] where he had promoted and advocated a view on Mathematics as a humanitarian science.)

In the talk *On programming content of Math contests* we drew attention to the importance of introduction of programming art and science [5,9,15,16] to mathematical education not just because of industrial demand and/or employment opportunities for graduates but because of a need of programming culture for solving mathematical problems. We would like to advocate this claim by analysis of the problem set [40] of the most recent International Mathematical Olympiad [39] (which was the 60th in the series).

The Olympiad set [40] comprises 6 problems from which 1.5 (exactly *one and a half*) are good examples to demonstrate programming art and science. Namely, we speak about the following problems from the set.

[Problem IMO-19-1]. Let $\mathbb{Z}$ be the set of integers. Determine all functions $f : \mathbb{Z} \to \mathbb{Z}$ such that, for all integers a and b, $f(2a) + 2f(b) = f(f(a+b))$.

[Problem IMO-19-5]. The Bank of Bath issues coins with an H on one side and a T on the other. Harry has n of these coins arranged in a line from left to right. He repeatedly performs the following operation: if there are exactly $k > 0$ coins showing H, then he turns over the kth coin from the left; otherwise, all coins show T and he stops. For example, if $n = 3$ the process starting with the configuration THT would be $THT \to HHT \to HTT \to TTT$, which stops after three operations.

a) Show that, for each initial configuration, Harry stops after a finite number of operations.
b) For each initial configuration C, let $L(C)$ be the number of operations before Harry stops. For example, $L(THT) = 3$ and $L(TTT) = 0$. Determine the average value of $L(C)$ over all 2^n possible initial configurations.

The problem IMO-19-1 can serve as an example of recursion elimination [15,22] using reduction of a monadic recursion to a tail recursion, we discuss this programming technique and its application to the problem IMO-19-1 in the next subsection. (A pure mathematical solution can be found at [40] in the original *Problems (with solutions)* provided by the 60th International Mathematical Olympiad, and watched at [41] among other Math videos by Presh Talwalkar.)

The problem IMO-19-5(a) is a “typical” problem on algorithm termination to be solved by Floyd method [9,25] (but this time some preliminary equivalent algorithm transformations are required), we present a programming solution of the problem in the subsection after the next one.

3.3 Problem IMO-19-1 via Recursion Elimination

A classic example monadic recursion elimination by reduction to the tail recursion is a so-called John McCarthy function $M_{91} : \mathbb{N} \to \mathbb{N}$ [15,22] that is defined as follows below:

$$M_{91}(n) = \ if\ n > 100\ then\ (n - 10)\ else\ M_{91}\ (M_{91}(n + 11)).$$

It turns out that $M_{91}(n) = \ if\ n > 101\ then\ (n - 10)\ else\ 91$. A key idea in recursion elimination is move from a monadic function $M_{91} : \mathbb{N} \to \mathbb{N}$ to a binary function $M2 : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ such that $M2(n, k) = (M_{91})^k(n)$ for all $n, k \in \mathbb{N}$ (where $(M_{91})^k(n)$ is k-time application of the function, i.e. $(M_{91})^k(n) = \overbrace{M_{91}\,(\dots M_{91}(n) \dots)}^{k}$); of course, $M2(n, 0) = (M_{91})^0(n) = n$ for every $n \in \mathbb{N}$.

Let us apply the idea presented in the previous paragraph to the problem 1. Since $f(2a) + 2f(b) = f\,(f(a + b))$ is true for all $a, b \in \mathbb{Z}$, then $f(0) + 2f(b) = f(f(b))$ for all $b \in \mathbb{Z}$. Let us define a binary function $F : \mathbb{Z} \times \mathbb{N} \to \mathbb{Z}$ such that $F(b, k) = f^k(b)$ and $F(b, 0) = f^0(b) = b$ for all $a \in \mathbb{Z}$ and $k \in \mathbb{N}$. Then for all $a \in \mathbb{Z}$ and $k \in \mathbb{N}$

$$\begin{aligned} F\,(b, (k + 1)) &= 2F(b, k) + f(0) = 2\,(2F\,(b, (k - 1)) + f(0)) + f(0) = \dots \\ &= 2^{(k+1)}F(b, 0) + (2^{(k+1)} - 1)f(0) = 2^{(k+1)}b + (2^{(k+1)} - 1)f(0), \end{aligned}$$

and, hence, $f(b) = f^1(b) = F(b, 1) = 2b + f(0)$ and thus the problem 1 is solved!

3.4 Problem IMO-19-5(a) via Proving Algorithm Termination

Let us start with the following formalization (in pseudo-code) of the algorithm specified in the problem statement 5:

```
var W: a word in the alphabet {T,H};
var k: a natural number;
while H exists in W
do k:= number of H in W;
   if W[k] = T then W[k]:= H else W[k]:= T
od
```

Because of the loop condition `while H exists in W`, the only thing we need to prove is the loop termination.

For this purpose, let us transform the above algorithm as follows:

```
var W: a word in the alphabet {T,H};
var k, i: natural numbers;
while H exists in W
do k:= number of H instances in W;
   i:= k;
   while W[i] = T
```

```
        do W[i]:= H; i:= i+1 od;
   while W[i] = H
        do W[i]:= T; i:= i-1 od;
od
```

This transformed algorithm is equivalent to the original one because it first serializes conversions of `T` to `H` and then serializes conversions of `H` to `T`. Remark that the conjunction of the following three clauses

- the number of `H` in `W` is `i`;
- `k` $\leq$ `i` $\leq$ the index of the rightmost instance of `H` in `W`;
- `W[k..(i-1)]` consists of `H` only (i.e. hasn't any instance of `T`)

forms an invariant [9] of each of both internal loops (i.e. if the conjunction is true before any exercise of a loop body then it remains true after the exercise). It implies that for each legal iteration of the external loop (i.e. when `W` has any instance of `H`)

the number of instances of `H` in `W`
before the loop body exercise
(that is the value of `k`)
is positive and greater than
the number of instances of `H` in `W`
after the loop body exercise
(that is the final value of i);

in other words, the number of instances of `H` in `W` decreases on each legal iteration of the external loop. Hence, the number of instances of `H` in `W` is the loop variance and (according to Floyd method of proving termination [9]) the transformed algorithm as well as the original one always terminates.

4 Conclusion: What Else and Next?

Fun, jokes, puzzles, games and entertainment in teaching is not the unique ingredient needed to improve Formal Method education (more general — Computer Science and Software Engineering education). All these should be used to engage (undergraduate) students with learning/study/comprehension/mastering Formal Methods. We believe that experience of individual educators and expertise of research groups in the field of Formal Method popularization deserves a positive attitude from Computer Science and Software Engineering academic community and industry.

Another opportunity to engage students is a competitive spirit that is so appropriate to young people (in particular — to students of CS and SE departments). International competitions between FM tools (e.g. automated theorem provers and satisfiability solvers) are popular, useful and valuable from industrial and research perspectives, but not from undergraduate education perspective. Unfortunately, competitions especially designed for (undergraduate) students (like Collegiate Programming Contest [42]) are still not involved into education process in general and in FM education in particular. We hope that competitions of this kind may be used better for engaging students with Theory of Computer Science and Formal Methods in Software Engineering [26].

We would like to conclude by drawing attention to a so-called *IMO Grad Challenge* [43]:

> *The International Mathematical Olympiad (IMO) is perhaps the most celebrated mental competition in the world and as such is among the ultimate grand challenges for Artificial Intelligence (AI).*
> *The challenge: build an AI that can win a gold medal in the* ***competition****.*
> *To remove ambiguity about the scoring rules, we propose the formal-to-formal (F2F) variant of the IMO: the AI receives a formal representation of the problem (in the Lean Theorem Prover), and is required to emit a formal (i.e. machine-checkable) proof. We are working on a proposal for encoding IMO problems in Lean and will seek broad consensus on the protocol.*
> ...
> *Challenge. The grand challenge is to develop an AI that earns enough points in the F2F version of the IMO (described above) that, if it were a human competitor, it would have earned a gold medal.*

So, it is a high time for mathematicians not only to learn the art and the science of programming, but technologies and tools of the Artificial Intelligence!

References

1. Curzon, P., McOwan, P.: Teaching formal methods using magic tricks. In: Contributed talk at the CAV Workshop Fun With Formal Methods, St.Petersburg, Russia, 13 July 2013. http://www.chi-med.ac.uk/publicdocs/WP122.pdf. Accessed 20 Jan 2020
2. Dijkstra, E.W.: On a cultural gap. Math. Intell. **8**(1), 48–52 (1986). https://doi.org/10.1007/BF03023921
3. Ershov, A.P.: Aesthetics and the human factor in programming. Commun. ACM **15**(7), 501–505 (1972)
4. Ershov, A.P.: Programming as the second literacy (1980) (In Russian). http://ershov.iis.nsk.su/ru/second_literacy/article. Accessed 20 Jan 2020
5. Ershov, A.P., Knuth, D.E. (eds.): Algorithms in Modern Mathematics and Computer Science. LNCS, vol. 122. Springer, Heidelberg (1981). https://doi.org/10.1007/3-540-11157-3
6. Floyd, R.W.: Assigning Meaning to Programs. In: Proceedings of Symposium on Applied Mathematics, vol 19, pp. 19–32. Amer. Math. Soc. (1967)
7. Furia, C.A., Nordio, M., Polikarpova, N., Tschannen, J.: AutoProof: auto-active functional verification of object-oriented programs. Int. J. Softw. Tools Technol. Transfer **19**(6) 697–716 (2017)
8. Gay, S.J.: Cables, trains and types. In: Chris Hankin's Festschrift (to appear). http://www.dcs.gla.ac.uk/~simon/publications/CablesTrainsTypes.pdf. Accessed 20 Jan 2020

9. Gries, D.: The Science of Programming. Monographs in Computer Science. Springer-Verlag, New York (1981). https://doi.org/10.1007/978-1-4612-5983-1
10. Hoare, C.A.R.: An axiomatic basis for computer programming. Commun. ACM **12**(10), 576–580 (1969)
11. Holzmann, G.J.: Mars code. Commun. ACM **57**(2), 64–73 (2014)
12. Karpov, Y.G.: Model checking: verification of concurrent and distributed systems. BHV-Petersburg (2010) (In Russian)
13. Khazeev, M., Mazzara, M., De Carvalho, D., Aslam, H.: Towards a broader acceptance of formal verification tools: the role of education. arXiv:1906.01430 [cs.SE]. https://arxiv.org/abs/1906.01430. Accessed 20 Jan 2020
14. Knuth, D.E.: Computer science and its relation to mathematics. Am. Math. Mon. **81**(4), 323–343 (1974)
15. Knuth, D.E.: Textbook Examples of Recursion. https://arxiv.org/pdf/cs/9301113.pdf (1991). Accessed 20 Jan 2020
16. Knuth, D.E.: The Art of Computer Programming, Volumes 1–3 Boxed Set, 2nd edn. Addison-Wesley, Reading (1998)
17. Lisitsa, A.: Tackling Fibonacci words puzzles by finite countermodels. In: Contributed talk at the CAV Workshop Fun With Formal Methods, St.Petersburg, Russia, 13 July 2013. http://cgi.csc.liv.ac.uk/ alexei/Fibonacci_Challenge/fun2013.pdf. Accessed 20 Jan 2020
18. Logas, H., Kirchner, F., Murray, J., Schaf, M., Whitehead, E.J. (Jr.): Chekofv: crowd-sourced formal verification. In: Contributed talk at the CAV Workshop Fun With Formal Methods, Vienna, Austria, 13 July 2014
19. Murray, J., Whitehead, J., Kirchner, F.: Crowd-sourced help with emergent knowledge for optimized formal verification (CHEKOFV). SRI INTERNATIONAL, March 2016, FINAL TECHNICAL REPORT. https://users.soe.ucsc.edu/~ejw/papers/Chekofv%20Final%20Report%20Part%20A.pdf. Accessed 20 Jan 2020
20. Parnas, D.L.: Really rethinking "Formal Methods". IEEE Comput. **43**(1), 28–34 (2010)
21. Rushby, J.: The ontological argument in PVS. In: Invited talk at the CAV Workshop Fun With Formal Methods, St.Petersburg, Russia, 13 July 2013. http://www.csl.sri.com/users/rushby/papers/ontological.pdf. Accessed 20 Jan 2020
22. Shilov, N.V.: Etude on recursion elimination. Model. Anal. Inf. Syst. **25**(5), 549–560 (2018)
23. Shilov, N.V.: Using esoteric language for teaching formal semantics. Contributed talk at the CAV Worthe same languagekshop Fun With Formal Methods, Vienna, Austria, 13 July 2014
24. Shilov, N.V.: Make formal semantics popular and useful. Bull. Novosibirsk Comput. Center Ser. Comput. Sci. IIS Special Issue **32**, 107–126 (2011)
25. Shilov, N.V., Shilova, S.O.: On mathematical contents of computer science contests. In: Enhancing University Mathematics: Proceedings of the First KAIST International Symposium on Teaching. American Society, CBMS Issues in Mathematics Education, vol. 14, 193–204 (2007)
26. Shilov, N.V., Yi, K.: Engaging students with theory through ACM collegiate programming contests. Commun. ACM **45**(9) (2002)
27. Uspensky, A.V.: Mathematics Apology. Amphora, Sant-Petersburg (2009) (In Russian)
28. Vardi, M.Y.: Publish and Perish. Commun. ACM 63(1), 7 (2020)

29. A.M. Turing Award Winners. http://amturing.acm.org/award_winners/clarke_1167964.cfm, http://amturing.acm.org/award_winners/emerson_1671460.cfm. http://amturing.acm.org/award_winners/sifakis_1701095.cfm. Accessed 20 Jan 2020
30. Fun With Formal Methods (2013). http://www.iis.nsk.su/fwfm2013. Accessed 20 Jan 2020
31. Fun With Formal Methods (2014). http://www.easychair.org/smart-program/VSL2014/FWFM-cfp.html. Accessed 20 Jan 2020
32. Fun With Formal Methods (2018). https://persons.iis.nsk.su/en/FWFM2018. Accessed 20 Jan 2020
33. Fun With Formal Methods (2019). https://persons.iis.nsk.su/en/FWFM19. Accessed 20 Jan 2020
34. COVID-19 pandemic. https://en.wikipedia.org/wiki/COVID-19_pandemic. Accessed 15 June 2020
35. Tools 50+1 conference. Day 3. Fun With Formal Method Workshop. https://www.youtube.com/watch?v=QqLRUWD9Ngg. Accessed 20 Jan 2020
36. PVS specification and verification system. https://github.com/SRI-CSL/PVS/. Accessed 20 Jan 2020
37. CS4F. www.cs4fn.org. Accessed 20 Jan 2020
38. Esoteric Programming Languages. https://en.wikipedia.org/wiki/Esoteric_programming_language. Accessed 20 Jan 2020
39. International Mathematical Olympiad. https://www.imo-official.org/default.aspx. Accessed 20 Jan 2020
40. Problems (with solutions). In: 60th International Mathematical Olympiad. Bath - UK, 11th-22nd July 2019. https://www.imo2019.uk/wp-content/uploads/2018/07/solutions-r856.pdf. Accessed 20 Jan 2020
41. Solving An Insanely Hard Problem For High School Students. MindYourDecisions - Math videos by Presh Talwalkar. https://www.youtube.com/watch?v=uJqbHaFqjmI. Accessed 15 June 2020
42. ICPC. International Colegiate Programming Contest. https://icpc.baylor.edu/. Accessed 20 Jan 2020
43. IMO Grand Challenge. https://imo-grand-challenge.github.io/. Accessed 20 Jan 2020

Adapting to Different Types of Target Audience in Teaching Formal Methods

Antonio Cerone[1](✉) and Karl Reiner Lermer[2]

[1] Department of Computer Science, Nazarbayev University, Nur-Sultan, Kazakhstan
antonio.cerone@nu.edu.kz

[2] Department of Computer Science, ZHAW Zurich University of Applied Sciences, Winterthur, Switzerland
lrka@zhaw.ch

Abstract. Formal methods can be considered as the area of computer science that most effectively bridges the gap between mathematics and computer science. They are potentially a great educational tool for fostering mathematical reasoning skills and problem-solving abilities in a very wide audience of potential learners from university, industry, school and research.

Unfortunately, this great potential is not exploited in reality. Formal methods are taught only in a limited number of computer science university programmes, mainly at postgraduate level, and are usually presented as such a difficult topic that university students keep away from them and the industry, in general, does not consider them as a worthy research and development investment. Even worse, most of the technicians (electrical or machine engineering) who design and build safety critical systems never had a course in formal methods during their studies.

In this paper we draw upon our experience in teaching formal methods to the heterogeneous audience of potential learners. We report on how teaching methods and materials must be adapted to the specific type of target audience to effectively produce learning outcomes. We observe that motivation, fun and practice are essential dimensions of such an adaptive approach.

Keywords: Formal methods · Teaching approach · Target audience

1 Introduction

There is a widespread misconception that mathematics and computer science are two independent, fully distinct disciplines, with the fact that computers can be used to perform complex mathematical calculations being the only perceived connection between the two disciplines. This misconception determined

Work partly funded by Seed Funding Grant, Project SFG 1447 "Formal Analysis and Verification of Accidents", University of Geneva, Switzerland.

A. Cerone and M. Roggenbach (Eds.): FMFun 2019, CCIS 1301, pp. 106–123, 2021.
https://doi.org/10.1007/978-3-030-71374-4_5

a large gap between computer science and mathematics, mainly in educational and industrial contexts, and partly also in the context of scientific research.

The source of this misconception is that computer science is normally identified with programming, and programming is seen more like a kind of art rather than an applied science. This widespread perception of computer science has its roots in people's common beliefs as well as in school education. Programmers are normally considered like weird people, fully immersed in their work activities and in some sense detached from the real world, just like artists. Even those who see programmers somehow as scientists, actually identify them with 'crazy scientists', in fact, with 'gits', namely unpleasant or contemptible people. This stereotype is so widely accepted that Linus Torvald used the word 'git' to name his versioning system, i.e. *Git*, apparently with the motivation that this was a way he saw himself [2]. Ironically, as a result, the Git-based platform that collects most of nowadays open source software projects, to which programmers contribute in their free time, almost addictively and mostly without been paid, is called *GitHub*.

The gap between computer science and mathematics led to a debate on the centrality of mathematics and logic in computer science curricula: on the one side the claim that rigorous mathematical knowledge is not necessary for computer science practitioners [21] and, on the other side, the belief [39,40] and the empirical evidences [31,36] that learning rigorous discrete mathematics and formal methods has an important impact on problem-solving and programming skills and is perceived by students as useful in practical problems and helpful in improving their mental processes [42].

In the last two decades computers have been heavily introduced in schools. In many schools computer science has even been introduced as a new, stand-alone subject. However, this has been normally done without connecting computer science with mathematics but, instead, by seeing computer science as a "service subject", whose only scope is that of providing tools that facilitate the students in carrying out their homework and class projects [12,20]. The teaching of computer science to school pupils tends, therefore, to focus on using office-oriented tools to write documents, prepare presentations and organise data in spreadsheets.

1.1 Formal Methods and Its Potential Audience

Formal methods is one of the most challenging areas of computer science. It has at least four distinctive aspects that make it unique in several respects. We believe that these distinctive aspects originates from the fact that formal methods is the area that most effectively bridges the gap between mathematics and computer science. From a theoretical point of view we can say that a *formal model* is a mathematical representation of a computer program, namely a mathematical object that we can manipulate with potentially infinite mathematical tools. Thus, moving from computer programs to formal models through an *abstraction process* allows us to exploit the power of mathematics in order to understand what the program does, namely its *semantics*, reason about it, and analyse it statically in a precise way.

Therefore, formal methods, on the one hand, foster those mathematical reasoning skills that are essential in producing correct, effective software and, on the other hand, make computer science, in some way independent of computers. A third distinctive aspect of formal methods, one with very practical consequences, is its potential to provide an effective way to analyse a large number of critical, non-functional properties of software, including safety, security, reliability and usability. In trying to deal with such critical properties software engineering principles, guidelines and methodologies have always been struggling and never managed to fully provide assurance. Meeting these properties is necessary for the most critical and innovative technology in use today and represents a present and future challenge for the exponential increase in system complexity determined by ubiquitous computing and the internet of things. Finally, the fourth distinctive aspect is that formal methods can be applied, beyond computer science and technology, to several disciplines, including physics, chemistry, biology, ecology, psychology, cognitive science and economics.

We can thus claim that formal methods have the potential to address a very wide audience, which comprises

University students in computer science, who need to develop abstraction and reasoning skills needed to produce, understand and analyse software;

School pupils in order to allow them to establish the mathematical problem-solving bases that can enable them to succeed in scientific and technology-oriented university programmes;

Research being formal methods applicable to a wide range of domains, especially to innovative technologies, they must adapt to continuoulsly evolving technologies and to the heterogenuous needs of interdisciplinary research teams.

Industry not only in the software area but also in a number of technology sectors, either safety-critical, such as transportation, avionics, aerospace, chemical plants, nuclear power plants, medical devices, or security-citical, such as e-commerce and defence, and to complex systems encountered in chemistry, biology, ecology, psychology and economics;

Given such a large potential audience, why then aren't formal methods widely taught in universities and schools as well as in industrial training? Why aren't they widely accepted and used in industrial contexts and fully recognised in research areas such as software engineering and human-computer interaction?

1.2 Structure of the Paper

In this paper, after describing our backgrounds and the settings in which we carried out teaching, student supervision, research and collaboration with industry (Sect. 2), we give an account of our experience in teaching formal methods and identify and discuss a number of dimensions that drove the development of our engagement strategy through the years (Sect. 3). Section 4 illustrates how engagement strategy may be adapted to various kinds of audience. Section 5 concludes the paper.

2 Authors' Background and Formal-Methods-Related Activities

The authors of this paper have backgrounds in computer science and mathematics, respectively. They have several years of experience in teaching and supervising students in their respective areas, both at the undergraduate and post-graduate level. They also carry out research in the area of formal methods, especially in terms of applications to safety-critical systems and in the modelling and analysis of complex systems within the domains of biology, ecology and psychology.

Both authors worked several years at the Software Verification Research Centre (SVRC), a special research centre of the Australian Research Council, which was active during 1993–2003 in the areas of formal methods tool development and formal verification of industrial software. The authors were employed in a 50-50 research and technology transfer programme. In addition to carry out research in the area of formal methods, as SVRC employees, they provided consultancy services to several organisations and companies, including DSTO (now DST – Defence Science and Technology), British Aerospace, Foxboro, Computer Science Corporation and Santos, as well as training in form of short intensive courses.

The first author then moved to the International Institute for Software Technology of the United Nations University (UNU-IIST), which was located in Macao SAR, China, where for almost 10 years he continued to carry out research in the area of formal methods and was particularly involved in the diffusion of software technology and formal methods in developping countries through the delivery of short intensive courses and supervision of graduate fellows and PhD students. He has also taught Master and PhD courses in formal methods at UNU-IIST, the University of Pisa, IMT Lucca and Nazarbayev University.

The second author moved to Zurich where he took up a lecturer position at the Zurich University of Applied Sciences (ZHAW). As a member of the University's Safety Critical Systems Research Lab he is doing active research and technology transfer in formal methods in various industrial projects. Under a recently funded University project he developed E-Learning materials to enrich and improve the Bachelor education in mathematics.

3 A Multi-dimensional Engagement Strategy in Formal-Methods Education

This section discusses the lessons learned during the authors' teaching, supervision and consultancy activities in the area of formal methods. Since formal methods are not well received by both the academic and industrial audience [34], the main challenge was to develop a strategy to reverse this trend and keep the learner continuously motivated and engaged in order to retain any acquired form of interest. In the process, we identified a number of dimensions of this engagement strategy, which we illustrate in Sect. 3.1, 3.2, 3.3, 3.4 and 3.5.

3.1 Motivations

Given the widespread reluctance to learn formal methods, the simple strategy of providing potential learners with a list of reasons for being interested in this area would not be effective. We experienced that a better strategy is, instead, that of enabling learners to build themselves their intrinsic and extrinsic motivations. We have identified a number of tools to achieve this objective:

Start With the General Context Rather Than the Foundations
A typical mistake in teaching or even just advertising a challenging subject is that of starting from theoretical foundations and basic technical aspects. Such an approach appears dry and non motivating to potential learners. The result is that the least skilled potential learners will get scared and run away and the most skilled ones will get bored and find little interest in the subject. In our view, based on our experience, motivations can be enabled through an initial, broad presentation of the general context in which formal methods are successfully applied, but without actually putting any emphasis on formal methods themselves and leaving out all technical details. Moreover, when technical details are introduced, in a soft, incremental way, they must always be referred to this motivational context and possibly contribute to extend it. We will present some examples of this approach in Sect. 4.

Present Specific Success Stories and Showcases
In spite of industry's general reluctance in accepting formal methods, some companies have actually used them in research projects or in the verification of their software products or deployed systems. There is a number of success stories that could be presented to potential formal methods learners. Preference should be given to popular companies and the success stories should be presented using high-level descriptions, normally available in newspapers, magazine, short communications or internet resources rather than technical journal papers. However, the success story should not be an unrealistic celebration of a panacea approach but, to be credible, should describe a global positive outcome that includes both pros and cons. A good example in this sense could be the use of formal methods at Amazon Web Services, which is reported as big success but with some remaining caveats [28].

Consider and Incorporate Current Trends
In terms of extrinsic motivation it is important to connect formal methods to the most trendy areas of the moment, which are seen as a must in the job market, a hot topic in research and an essential tool in industrial production, thus appealing to the entire potential audience from students to researcher and industry. A today's example is represented by the hot area of *data science* and its subdisciplines. Providing the intuition on how formal methods connect to data science, using examples on current work [4,15] as well as ideas for future research, is necessary to boost strong intrinsic motivations.

Start Education in Formal Methods Early Enough
The absence of the appropriate mathematical background is the biggest barrier for potential formal methods learners. A common belief is that formal

methods are very far from people's normal way of thinking and reasoning. Actually, the opposite is true. The same problem-solving and reasoning skills needed in real-life can be used to solve problems in the area of formal methods. The only difference is that the reasoning object is not a concrete fact, but an abstract model. *Abstraction skills* are what enables us to move from the reality to its models, to which formal methods can be applied.

Unfortunately, the current status of mathematics teaching around the world is not addressing abstraction skills [20]. In fact, mathematics should be taught using a *mathematical problem-solving* approach since the early school years [6,22,35], in which it is already possible to introduce formal methods [12,20], and continuing with such an approach during the university years. The opposite seems, instead, to happen in the last years, with schools emphasising on calculation rather than reasoning abilities or on repeated pattern recognition problems that are never finalised to a successful abstraction process [20]. Even at the university level, a fundamental mathematics subject like calculus, which was recognised in the past as the best tool to develop abstraction and reasoning skills through the development of proofs in the real spirit of mathematical analysis, is now restricted to the teaching of mere calculation techniques. In fact, the use of calculators and iPads is in the focus of modern calculus teaching. For instance, openly discussed is to remove essential mathematical theories like solving differential equations from engineering postgraduate math courses.

In addition, programming is decoupled from mathematics in many respects and is often taught as a sort of unsystematic,'artistic' skill of syntactic manipulation, within a trial-and-error rather than logical approach. For instance, the formal semantics of programming languages is no longer taught in the early programming courses, which nowadays just focus on syntax.

In such an unfavorable situation, in addition to try to propose innovative approaches to be carried out globally, starting from the early school years, we claim the importance of introducing formal methods already at undergraduate level, both in core subjects, such as programming, software engineering and operating systems as well as in elective subjects, such as human-computer interaction, information security and project-based electives. We will discuss these proposals in Sect. 4.1 and 4.2.

3.2 Fun

One important aspect of formal methods is the possibility of combining notations that support problem specification with powerful tools that, given the specification as an input, provide problem solutions almost automatically. The authors are always impressed by the combined feeling of surprise, happiness and sense of achievement externalised by learners when they realise that their specifications actually "work" with the tool.

Moreover, formal methods can be applied to a large range of problems, basically any problem, well beyond the domain of computer science. In fact, in addition to classical computer science problems, such as the dining philosopher, communication and cryptographic protocols and distributed algorithms, we have a huge range of candidates among classical mathematical puzzles as well as popular games and even video games.

Mathematical puzzles that can be visually represented, such as the "river crossing puzzle" [5], present an easy way to approach formal methods; they allow learners to gradually move from the graphical representation to a mathematical notation. This kind of problems, which have a clear visual representation, are particularly suitable in school and early undergraduate courses [19].

Other more complex mathematical puzzles and popular games, such as sudoku and card games, have the potential to strongly engage learners [18]. At the end of a midtem examination that consisted in the modelling and formal analysis of a card game, the author of this paper had the nice experience to hear a student saying: "This examination was real fun!"

One approach used by the authors is that of providing learners with examples of formal methods descriptions of video games and invite them to create formal models of their favourite video games. This kind of tasks appeared to be very engaging and can be successfully carried out even in high school and early undergraduate courses. In this context, it is particularly important to blur the distinction between learner and instructor by letting the learners drive the choice of exercises and use their creativity to identify and specify potential problems and invent new games.

In general, we can claim that, if "motivation" is the dimension that allows learners to build up interest in formal methods, "fun" is actually the essential dimension to keep learners continuously engaged, thus assuring the retention and possibly increase of their interest over the time [9,14].

3.3 Which Formal Methods?

In our teaching and supervision activities our students have been exposed to a large variety of formal methods including the Z specification language, the refinement calculus, Petri nets, process algebras, and several logic systems, from rewriting logic to temporal logic. And this has been done both theoretically and practically using theorem-proving and model-checking tools.

From our experience we observed that the choice of which formal methods to present to the learners mostly depends on three parameters:

1. The age, level and background of the learners;
2. The application domain, which may be identified with the taught subject in the case of university students;
3. The availability and features of software tools.

Some specific discussion on Parameter 1 will be presented in Sect. 3.4 and 4. Parameter 3 will be discussed in Sect. 3.5.

Unfortunately, we cannot establish general rules for using such parameters to drive our choice, which actually depends not only on the characteristics of the learners, but also on the preference, background and skills of the instructor. Therefore, we limit the discussion in this section to the account of the first author's experience in teaching several variants of a postgraduate course in formal methods to a variety of audiences at postgraduate level

- in several developing countries as part of the United Nations University training programme;
- in PhD courses in Macao SAR (at UNU-IIST) and Italy (at the University of Pisa and at the IMT School for Advanced Studies Lucca); and
- in Master Courses at Nazarbayev University, Kazakstan.

Teaching to university lecturers and postgraduate students in developing countries was very challenging. In addition to the logistic and infrastructural problems of such in-house courses, the biggest challenge was the limited mathematical background of the learners, but still with a large variability, which could not predicted *a priori*. The strategy for dealing with this challenge was an on-the-fly adaptability of such courses, which definitely contributed to the development of the proposals and approaches illustrated in this paper. Being these teaching contexts very specific, a detailed account on such experiences is beyond the scope of this paper.

Three different formal methods approaches used in the course were

- RAISE (Rigorous Approach to Industrial Software Engineering) and its specification language (RSL) and associated tools [29];
- the CSP (Communicating Sequential Processes) process algebra [3,23,33], initially with the support of the CWB-NC (Concurrency Workbench of the New Century) tool [17], later replaced by PAT (Process Analysis Toolkit) [27,37,38];
- rewriting logic and the Maude language and model checker [16,26,30].

The use of RAISE was soon abandoned due to the difficulties encountered by the students in producing consistent specifications and to the poor usability of the associated tools. Therefore, we only compare the process algebra and rewriting logic approaches.

In the PhD course taught at IMT during the academic year 2014–2015, both formal methods approaches were introduced and specifically applied to the modelling of interactive systems. The translations of a description language for human behaviour tasks to both formal methods were presented during the course. However, the way that course was conducted does not reflect the approach proposed in this paper. In fact, the first part of the course was devoted to the theoretical presentation of the two formal methods approaches and to pen and paper modelling exercises. Only in the second part of the course the PAT and Maude tools were introduced. At the end of the first part of the course, students were asked three questions:

1. "In which of the two approaches did you find easier to get the model right?"
2. "Which of the two translations is more elegant?"
3. "In which of the two approaches the resultant behaviour is easier to guess?"

The PhD students unanimously answered “the rewriting logic approach” to Questions 1 and 3, and “the process algebra approach” to Question 2. It is interesting to observe that, in spite of finding the process algebra approach more difficult, the student unanimously agreed that it is more elegant. These answers, as well as further remarks and opinions that emerged in an open discussion that followed, are an indicator that students have a strong interest for solutions that are concise, elegant and abstract, and that they are happy to tackle challenging problems in order to look for elegant rather than easy, but somehow messy solutions. In this specific case, the “elegant challenge” was the use of concurrency in modelling the system in a compositional way, whereas the “easy but messy solution” was the monolithic modelling of the global system using rewrite rules. Given the small number of students and the absence of research design we cannot draw empirical conclusions from the students’ answers and remarks, although these appear to be in line with the results of previous research [42].

With respect to Parameter 2 above, the students’ answers seem to suggest that rewriting logic is more suitable than process algebra and other approaches based on parallel composition to model human behaviour. As we will see in Sect. 3.5 this cannot be a definite conclusion.

3.4 Textual Versus Visual Notations

In the case of school children simple, visual notations, such as Petri nets and finite state machines are obviously the best choices for introducing formal methods. Several formal methods concepts, such as refinement, abstraction and concurrency, can be directly identified on the visual representation, in most cases with no recourse to formal mathematical representations. The important thing for school children is the discovery and internalisation of such concepts rather then their representations in some dry textual notation [12].

Visual notations also help a lot in the case of undergraduate students, but need to be finalised to the “discovery” of the formal semantics and its possible representations in mathematical notations. For example, Petri nets can be first introduced visually together with an informal presentation of their semantics, or actually their possible semantics. Then the students can be guided to represent such semantics in a mathematical way that can be used to calculate the future behaviour of the system. In the case of Petri nets students may visually identify and represent the semantics by

1. decomposing the net to represent each arc in terms of its sources and target;
2. decomposing the net to represent each transition in terms of its sources and targets;
3. drawing a table transitions $\times$ places for the entire net;
4. analising for each transition all markings enabling it.

Students would also observe that places may have their tokens produced by different transitions and that different places may cooperate to the firing of the same transition. Thus it does not make sense to represent a place in terms of the transitions that separately produce token in it and the transition that may separately consume tokens from it. The three representations above correspond respectively to

1. a flow relation, which can be expressed as two boolean functions on incoming and outgoing arcs;
2. the pre-set of post-set of each transition;
3. an incidence matrix;
4. a partial function from a marking to a set of markings, one for each enabled transition.

As the next step, depending on which the four representations above they have worked out, the students can be guided to discover the way to calculate the future behaviour, that is, the formal semantics of the Petri net. Finally, the equivalence of the various representations can be discussed.

3.5 Practice and Tools

In Sect. 3.3 we have reported students' opinions in the comparison of parallel composition and rewriting logic in modelling interactive systems, specifically human behaviour tasks. These opinions were collected after introducing the theory but before introducing the tools and starting using them. However, at the end of the course, after using both PAT and Maude, the opinions of the students were substantially unchanged.

More recently, at Nazarbayev University, the two approaches were used in the same postgraduate course on formal methods and applications as well as separately in two distinct instances of the undergraduate course on human-computer interaction. In these cases the approaches have been introduced together with the usage of the tools. In fact, the tools were used to introduce the language constructs and their semantics. Although a complete comparison cannot be carried out for the undergraduate courses since each students was exposed to only one of the two approaches, a better performance was achieved by the students exposed to the process algebra approach. Moreover, the postgraduate students found easier to use the process algebra approach than the rewriting logic approach.

The general lessons learned from these experiences are that [14]

1. instead of tediously going through the semantics of each construct in a formal language, students should be allowed to experiment with an appropriate tool to discover the semantics by themselves;
2. tools for simulation visualisation are essential to allow students to understand the behaviour associated with their models.

In fact, the introduction and usage of tools appear beneficial only if done early enough. If tools start to be used only after introducing the theory, it is difficult to reverse students' negative opinions and feelings.

Moreover, in general, in order to be beneficial to formal methods learners, tools should not require additional learning time and should, instead, facilitate the learning process. Thus they have to be easy to learn, at least in their basic features, documented in a concise but well-organised way and equipped with visual interfaces.

MAUDE and PAT are somehow complementary in terms of presentation of results, also due to the different characteristics of the formal methods on which they are based. MAUDE does not support any form of graphical representation but, through the use 'auxiliary' rewrite rules, allows the designer to filter the output and easily track which rewrite rule is applied and check the content of all data structures, thus tracking the behaviour back to the architectural view. PAT facilitates the visual representations of the global behaviour in terms of finite state machines, but the form of abstraction introduced by the CSP hiding operator is not very effective due to the possible introduction of nondeterminism, while the represented behaviour does not reflect the structure, in terms of concurrent components and synchronisations, from which the global behaviour has been attained. However, the use of both these tools in our course has allowed students to make use of complementary presentation features: visualisation from PAT and behaviour tracking from MAUDE. Moreover, in our class discussions, students showed the perception that the fact that the two tools are based on two distinct modelling paradigms contributed to stimulate and develop their abstraction and problem solving skills.

MAUDE documentation is well-written and presented at three levels: a primer [26], which allows a user to be able to effectively use the tool within a short time, a textbook [30] specifically designed for an undergraduate course, but also appropriate for postgraduate courses, and a comprehensive manual [16] for consultation and for acquiring a more advanced level of expertise. PAT documentation is unsatisfactory, especially for a novice. It consists of a poorly organised manual [27], several research papers, several talk presentations, materials on experiments and a couple of advertising videos. None of this material is really suitable for a learner.

4 Types of Target Audience

4.1 University Students

University students require a good balance between intrinsic and extrinsic motivations. We have discussed in Sect. 3.1 that motivations can be enabled through an initial, broad presentation of the general context in which formal methods are successfully applied. For university students this can be done at two levels:

1. Sparsely in core subjects, such as programming, software engineering and operating systems, and in elective subjects, such as human-computer interaction and information security.
2. In a focussed way, within a specific formal-methods-related subject.

Level 1 is the most effective in boosting early intrinsic motivations, which can play a decisive role later at the time of choosing elective subjects and thesis topics. In fact at this level, the presentation of the context in which to apply formal methods should be very broad, e.g. the integration of formal verification within the software life-cycle, in a software engineering course, or the simulation and formal analysis of human behaviour, in a human-computer interaction course. Showing the "pleasant aspects" of using formal methods is the actual objective at this level. For example, students of a software engineering undergraduate core subject are more likely to enjoy building a concise formal specification, with which they can also play using a tool, rather than writing a long, verbose and certainly boring specification document. Students of a human-computer interaction undergraduate elective subject taught by the first author enjoyed the formal modelling and analysis of a variety of small human tasks, including classical ones such as the interaction with an ATM (automatic teller machine) and the general car driver's behaviour, but also a number of fun, in some respect even hilarious, examples such as failing a driving test and baking a cake.

Of course, the feasibility, simplicity and fun of the tasks proposed by the teacher or, better, agreed between students and teacher, are essential for determining a pleasant rather than frustrating experience. In fact, although formal methods tend to be time consuming when applied to large systems, for several classes of small examples (e.g. the ones that can be systematically decomposed or are naturally recursive) they are indeed effective in saving time and reducing the workload, thus also boosting extrinsic motivations. However, in our experience among the motivations generated at this level, intrinsic motivation are probably going to be more long-lasting, especially for junior students. Future references to these contexts within the same subjects or even in other subjects would normally bring back intrinsic motivations. Senior students, who are often already looking for a job, are instead also very much affected by extrinsic motivations.

When dealing with specific formal-methods subject (level 2 above), the presentation of the context in which to apply formal methods should be more focussed on the learning objectives of the course. In the Master course on 'formal methods and applications' at Nazarbayev University, formal methods were first presented in the general context of the software life cycle, then in terms of more specific domain-oriented development, specifically their application to interactive systems, and finally in the light of synergetic approaches with trendy areas such as data science.

As a final remark concerning the use of tools, we would like to add that, from the perspective of a university student, it is important to see simulation and model-checking results directly on the low-level semantic structures underlying high-level domain structures. This is, in fact, an effective way for the student to understand and internalise the semantics of the language. Moreover, in the students' perspective, the presentation of results must aim at highlighting relations between behaviour and semantics. In fact, such a capability to output only relevant states and/or events is beneficial in stimulating and developping students' abstraction and problem solving skills.

4.2 School Pupils

For primary school pupils only intrinsic motivations make sense. In intermediate and high school, instead, also extrinsic motivation start to play a significant role.

Although in teaching to school pupils our experience is limited to research projects and practice with our own children, we agree with Gibson [20] about the importance of using formal methods to allow school pupils to establish early enough the mathematical problem-solving bases that can enable them to succeed in scientific and technology-oriented university programmes. Of course this has to be done at the right level and with the appropriate learning objectives [12].

The main challenge in teaching computer-science-related skills to school pupils is to make them aware that such skills have a general value, which is independent of the use of computers. For this reason we support an "unplugged approach" to teaching formal methods to school pupils, using activities that foster children reasoning and do not require the use of a computer [6]. In fact, this should be done in a multidisciplinary context; teaching formal methods should build on all school subjects, which, in the world of the school pupil, represent the most natural reality to be modelled formally. Obviously, mathematics should be the first subject to provide materials to manipulate in a formal fashion. However, all other subjects have also plenty of materials on which students may carry out modelling and analysis [12].

Moreover, as we discussed in Sect. 3.4, the emphasis should be on the discovery of concepts through the use of visual representations and the visual manipulation of such concepts aiming at their internalisation rather than their translation into some dry textual notation. If fact, we note that it would be pointless to just provide children with the definitions of new notions, concepts and processes, such as algorithms, and hope they understand them, remember them and are then able to apply them to practical situations. Children learn best if they are actively involved in the process through problem-solving [35].

4.3 Industry

The most natural use of formal methods should be in the area of industrial software verification. There are a number of recent academic publications that support the need for an extensive use of formal methods in industry [32,34,41]. However, given the high cost required by the use of formal methods, in terms of human and economic resources as well as time, the software development industry partly accepts the use of formal methods only for safety-critical systems. This partial acceptance is often not even a choice but the legal need to comply with the standards (e.g. IEC 61508 [24]) that suggest the use of formal methods for the most dependable software integrity level (SIL). Although no standard prescribes the use of formal methods as mandatory, the appeal to standards' recommendations is an effective incentive to offer practical courses on formal methods to industries working in the area of safety-critical systems. Additional enablers for extrinsic motivations for industry are success stories and showcases as we described in Sect. 3.1.

Moreover, rather than proposing or, even worse, imposing formal methods as a new approach, a better strategy is to present formal methods as integrated, or integrable, within a context which industry is familiar with. For example, ZHAW provides training and consultancy to safety-critical systems industry, such as railway, process industry, pharmaceutical industry, nuclear power plants and transportation, using STAMP (System-Theoretic Accident Model and Processes), a model-based approach centred on system theory to analyse accidents [25]. STAMP has been developed by Nancy Leveson as a model for safety engineering along with the System Theoretic Process Analysis (STPA) method, a hazard analysis method for finding inadequate design. This powerful top-down hazard analysis methodology has proven to be applicable in various industrial applications, including automotive, avionics, health care, power plant, railway, and many others [1]. It has also been successfully applied in the planning and technical system development and to already existing systems. The ZHAW Safety Critical Systems Research Lab experiences a growing demand for STPA analysis in a widespread variety of industrial areas, such as power plants, railway, health care and automotive.

In STPA, safety is viewed as a control problem, which is actually the natural way nuclear power and transportation engineers use to describe problems, and is managed by a control structure embedded in an adaptive socio-technical system and acting as constraints on the system. Therefore, STAMP models, being at the socio-technical system level can be clearly understood by engineers and other domain and safety experts.

The project "Formal Analysis and Verification of Accidents", a collaboration between the ZHAW Zurich University of Applied Sciences, Winterthur, Switzerland, and Nazarbayev University, Nur-Sultan, Kazakhstan, aims at the combination of a cognitive architecture for the formal analysis of human-computer interaction [10] and its associated description language, the Behaviour and Reasoning Description Language (BRDL) [11], with the STAMP approach. The cognitive architecture has been implemented using a formal methods approach based on Maude [8,13], which supports formal verification using model checking. The basic idea is to identify the steps in the STPA method that are suitable for formalisation, develop analysis engines based on model-checking and encapsulate them within tools equipped with high-level editors and interfaces appropriate for the usage by engineers and domain experts. The final objective of the project is to allow our industrial partners to use STPA-based tools, which also provide, in an unintrusive way, automated formal verification capabilities.

Finally, in terms of tools, we observe that the perspective of industry practitioner is very different from that of university students: they prefer tools that hide the formal semantic structures underlying domain structures.

4.4 Interdisciplinary Research Teams

Formal methods methodologies and tools can potentially be applied to several disciplines, not just computer science and the area of sofware-critical systems [7]. Formal modelling and analysis can be potentially exploited as effective research tools in many disciplines such as physics, chemistry, biology, ecology, psychology, cognitive science and economics. Unfortunately, formal methods experts are often so much focussed on the investigation of theoretical aspects of formal notations rather than on their applications to real problems, that they often neglect the needs of practitioners from applicative domains. The result is that methodologies and tools have limited usability for non computer scientists.

This problem has become very actual nowadays, with the launching of large interdisciplinary research projects, especially in areas such as biology, ecology and cognitive science. Different categories of experts within the same interdisciplinary research team may experience difficulties in understanding each other and share thoughts, due to both the different technical languages they use and their different way of reasoning. In research projects involving the application of formal methods to systems biology, ecosystem modelling and analysis of human behaviour and human errors, it is normally the formal methods expert who make the effort to understand the application domain, whereas the domain experts tend to act just as data providers or, in the best case, as consultants. And too often this leads to the development of unrealistic models and analysis tools with limited scope.

There is a need to change this situation. The most effective effort from the formal methods expert should actually be a technology transfer to domain experts in a form suitable to them. The embedding of formal methods within domain specific languages [11] and domain specific tools [13] represents an essential step in this direction. In fact, with this kind of audience it is necessary to use methodologies and tools that support domain specific notations, human-oriented proof and checking techniques, and domain-related formulation of properties. In this sense we could speak of *human-oriented* formal methods [9].

5 Conclusion and Future Work

In this paper we gave an account of our experience in teaching formal methods to various kinds of audience and identified and discussed a number of dimensions that drove the development of our engagement strategy. We observed that, on the one hand, these dimensions apply to such various kinds of audience differently. For example, the use of tools is not recommended for school pupils, is essential for university students in understanding and internalising semantic aspects, requires the hiding of the formal semantic structures underlying domain structures for industry practitioners and applicative domain researcher and, for the latter, also requires domain-specific notations and domain-related formulation of properties. On the other hand, we observed that, for any kind of audience, motivation is the dimension that allows learners to build up interest in formal methods, while fun is actually the essential dimension to keep learners continuously engaged, thus assuring the retention and possibly increase of their interest over the time.

In terms of future work we plan to develop teaching-oriented formal methods tools appropriate to difference audiences and, within the "Formal Analysis and Verification of Accidents" project the embedding of formal methods within methodologies that are widely accepted in industrial contexts (e.g. STPA).

References

1. Partnership for systems approaches to safety and security (PSASS). http://psas.scripts.mit.edu/home/materials/
2. Why the 'git' name? FAQ web page of the Git Wiki. https://git.wiki.kernel.org/index.php/Git_FAQ#Why_the.27Git.27_name.3F. Accessed 23 June 2020
3. Abdallah, Ali E., Jones, Cliff B., Sanders, Jeff W. (eds.): Communicating Sequential Processes. The First 25 Years. LNCS, vol. 3525. Springer, Heidelberg (2005). https://doi.org/10.1007/b136154
4. Aibassova, A., Cerone, A., Tashkenbayev, M.: An instrumented mobile language learning application for the analysis of usability and learning. In: Sekerinski, E., et al. (eds.) FM 2019. LNCS, vol. 12232, pp. 170–185. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-54994-7_13
5. Ascher, M.: A river-crossing problem in cross-cultural perspective. Math. Mag. **63**(1), 26–29 (1990)
6. Bell, T.: A low-cost high-impact computer science show for family audiences. In: 23rd Australian Computer Science Conference, pp. 10–16. ACM (2000)
7. Bowman, H., (ed.) Proceedings of "Formal Methods Elesewhere". Electronic Notes in Theoretical Computer Science, vol. 43. Elesevier (2000)
8. Cerone, A.: A cognitive framework based on rewriting logic for the analysis of interactive systems. In: De Nicola, R., Kühn, E. (eds.) SEFM 2016. LNCS, vol. 9763, pp. 287–303. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-41591-8_20
9. Cerone, A.: Human-oriented formal modelling of human-computer interaction: practitioners' and students' perspectives. In: Milazzo, P., Varró, D., Wimmer, M. (eds.) STAF 2016. LNCS, vol. 9946, pp. 232–241. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-50230-4_17
10. Cerone, A.: Towards a cognitive architecture for the formal analysis of human behaviour and learning. In: Mazzara, M., Ober, I., Salaün, G. (eds.) STAF 2018. LNCS, vol. 11176, pp. 216–232. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-04771-9_17
11. Cerone, A.: Behaviour and reasoning description language (BRDL). In: Camara, J., Steffen, M. (eds.) SEFM 2019. LNCS, vol. 12226, pp. 137–153. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-57506-9_11
12. Cerone, A.: From stories to concurrency: how children can play with formal methods. In: Cerone, A., Roggenbach, M. (eds.) FMFun 2019. CCIS, vol. 1301, pp. 191–207. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-71374-4_10
13. Cerone, A., Ölveczky, P.C.: Modelling human reasoning in practical behavioural contexts using real-time Maude. In: Sekerinski, E., et al. (eds.) FM 2019. LNCS, vol. 12232, pp. 424–442. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-54994-7_32
14. Cerone, A., Roggenbach, M., Schlingloff, B.-H., Schneider, G., Shaikh, S.: Teaching formal methods for software engineering – ten principles. In: Informatica Didactica, p. 9 (2015)
15. Cerone, A., Zhexenbayeva, A.: Using formal methods to validate research hypotheses: The Duolingo case study. In: Mazzara, M., Ober, I., Salaün, G. (eds.) STAF 2018. LNCS, vol. 11176, pp. 163–170. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-04771-9_13
16. Clavel, M., et al.: The Maude 2.0 system. In: Nieuwenhuis, R. (ed.) RTA 2003. LNCS, vol. 2706, pp. 76–87. Springer, Heidelberg (2003). https://doi.org/10.1007/3-540-44881-0_7

17. Cleaveland, R., Li, T., Sims, S.: The Concurrency Workbench of the New Century (Version 1.2) – User's Manual. SUNY at Stony Brook, July 2000
18. Ferreira, J.F., Mendes, A.: The magic of algorithm design and analysis: teaching algorithmic skills using magic card tricks. In: Proceedings of ITiCSE 2014. ACM (2014)
19. Ferreira, J.F., Mendes, A.: Open and interactive learning resources for algorithmic problem solving. In: Sekerinski, E., et al. (eds.) FM 2019. LNCS, vol. 12233, pp. 200–208. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-54997-8_13
20. Gibson., J.P.: Formal methods: never too young to start. In: FORMED 2008, pp. 151–160, Budapest, Hungary, March 2008
21. Glass, R.L.: A new answer to "how important is mathematics to the software practitioner?". IEEE Softw. **17**(6), 136–136 (2000)
22. Hilton, P.: The mathematical component of a good education. In: Hilton, P., Hirzebruch, F., Remmert, R. (eds.) Miscellanea Mathematica, pp. 145–154. Springer, Berlin, Heidelberg (1991). https://doi.org/10.1007/978-3-642-76709-8_9
23. Hoare, C.A.R.: Communicating Sequential Processes. Prentice Hall, Upper Saddle River (1985)
24. IEC 61508–1. Functional safety of electrical/electronic/programmable electronic safety-related systems – Part 1: General requirements, 2.0 edition, April 2010
25. Leveson, N.: A new accident model for engineering safer systems. Saf. Sci. **42**(2), 237–270 (2004)
26. McCombs, T.: Maude 2.0 Primer (Version 1.0). University of Illinois at Urbana-Champaign, August 2004. http://maude.cs.illinois.edu/w/images/6/63/Maude-primer.pdf
27. National University of Singapore. Process Analysis Toolkit (PAT) 3.5 User Manual. https://www.comp.nus.edu.sg/~pat/OnlineHelp/
28. Newcombe, C., Rath, T., Zhang, F., Munteanu, B., Brooker, M., Deardeuff, M.: How amazon web services uses formal methods. Commun. ACM **58**(4), 66–73 (2015)
29. Nielsen, M., Havelund, K., Wagner, K.R., George, C.: The RAISE language, method and tools. Formal Aspects Comput. **1**, 85–114 (1989)
30. Ölveczky, P.C.: Designing Reliable Distributed Systems. A Formal Methods Approach Based on Executable Modeling in Maude. UTCS. Springer, London (2017). https://doi.org/10.1007/978-1-4471-6687-0
31. Page, R.L.: Software in discrete mathematics. In: Proceedings of ICFP 2003, vol. 38 of ACM Sigplan Notices, pp. 79–86. ACM (2003)
32. Quinton, S.: Evaluation and comparison of real-time systems analysis methods and tools. In: Howar, F., Barnat, J. (eds.) FMICS 2018. LNCS, vol. 11119, pp. 284–290. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-00244-2_19
33. Roscoe, A.W.: The Theory and Practice of Concurrency. Prentice Hall, Upper Saddle River (1997)
34. Schlick, R., et al.: A proposal of an example and experiments repository to foster industrial adoption of formal methods. In: Margaria, T., Steffen, B. (eds.) ISoLA 2018. LNCS, vol. 11247, pp. 249–272. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-03427-6_20
35. Schoenfeld, A.H.: Mathematical Problem Solving. Academic Press, Orlando (1985)
36. Sobel, A.E.K., Clarkson, M.R.: Formal methods application: an empirical tale of software development. IEEE Trans. Softw. Eng. **28**(3), 308–320 (2002)

37. Sun, J., Liu, Y., Dong, J.S.: Model checking CSP revisited: introducing a process analysis toolkit. In: Margaria, T., Steffen, B. (eds.) ISoLA 2008. CCIS, vol. 17, pp. 307–322. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-88479-8_22
38. Sun, J., Liu, Y., Dong, J.S., Chen, C.: Integrating specifications and programs for system specification and verification. In: Proceedings of TASE 2009, pp. 127–135. IEEE Computer Society (2009)
39. Wing, J.M.: Teaching mathematics to software engineers. In: Alagar, V.S., Nivat, M. (eds.) AMAST 1995. LNCS, vol. 936, pp. 18–40. Springer, Heidelberg (1995). https://doi.org/10.1007/3-540-60043-4_44
40. Wing, J.M.: Invited talk: weaving formal methods into the undergraduate computer science curriculum (extended abstract). In: Rus, T. (ed.) AMAST 2000. LNCS, vol. 1816, pp. 2–7. Springer, Heidelberg (2000). https://doi.org/10.1007/3-540-45499-3_2
41. Xua, L.D., Xub, E.L., Lia, L.: Industry 4.0: state of the art and future trends. Int. J. Prod. Res. **56**(8), 2941–2962 (2018)
42. Zamansky, A., Farchi, E.: Exploring the role of logic and formal methods in information systems education. In: Bianculli, D., Calinescu, R., Rumpe, B. (eds.) SEFM 2015. LNCS, vol. 9509, pp. 68–74. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-49224-6_7

Prototyping Games Using Formal Methods

Sebastian Krings[1(✉)] and Philipp Körner[2]

[1] Institute for Information Security, Niederrhein University of Applied Sciences, Mönchengladbach, Germany
sebastian@krin.gs

[2] Institut für Informatik, Heinrich-Heine-Universität, Düsseldorf, Germany
p.koerner@uni-duesseldorf.de

Abstract. Courses on formal methods are often based on examples and case studies, which are supposed to show students how to apply formal methods in practice. However, examples often fall into one of two categories: First, many are artificial and thus do not relate to practice. Second, other examples are based on projects of industry partners and therefore often are too involved for students to understand them.

In this paper, we present a different approach. By formalizing the rules of commonly known games, we achieve examples both engaging and suited for students. Furthermore, we broaden the horizon of formal methods, driving research at the same time: we present extensions such as playable visualizations and explore the relationship between game AIs and model checking heuristics.

1 Introduction

Rather than purely focusing on the mathematical foundations, courses on formal methods are often based on examples and case studies, supposed to show students how to apply formal methods in practice. However, the examples used are often quite artificial and do not relate to practice. At the same time, examples based on projects of industry partners are rooted in practice but often are way too involved for students to understand.

In this paper, we present a different approach, relying on games as examples for formal models. The models discussed are used in teaching and have been developed by students both during courses and theses. We deem games particularly suited as teaching examples for two reasons:

1. The games we use are well-known to the students. We can thus focus on the modeling and proving as well as on methodology, rather than having to discuss intended properties of our models. Essentially, reducing the amount of requirements engineering we have to perform by using common examples allows us to focus on the formal method itself.
2. Modern computer games are among the most sophisticated examples of software systems. Due to the high complexity, implementations of game semantics, e.g., rules of movement, can often only be tested scarcely and are thus naturally suited for applying formal methods.

A. Cerone and M. Roggenbach (Eds.): FMFun 2019, CCIS 1301, pp. 124–142, 2021.
https://doi.org/10.1007/978-3-030-71374-4_6

As a simple example, consider the board game checkers. If an implementation somehow allowed moving a piece onto a white field, no valid moves would be defined afterwards. Arbitrary successor states might occur, or the game might never be over, as no other piece can capture it or vice versa. If the invalid move is only seldom possible, it might not be hit be testing procedures.

In the following, we will use the B-method [1] and its successor Event-B [2]. Both represent state-based formal methods used for modeling software and systems and proving their correctness. Models written in B or Event-B can be animated and model checked using ProB [24–26]. Additionally, we discuss the tools used in the development of our prototypes in Sect. 3.

Apart from teaching, using formal methods to prototype games has several advantages for game development itself: First, creation of a working prototype is often faster in B, due to the high level of abstraction[1]. Furthermore, step-wise refinement allows focusing on certain parts of a game. In consequence, our case studies contribute both to teaching and research.

2 A Primer on B and the B-Method

The formal specification language B [1], its successor Event-B [2] and the B-method [1] follow the correct-by-construction approach. Their models consist of a set of machines, which itself contain constants and variables together with corresponding type definitions. A predicate (which might have multiple solutions) is used to describe the initial states.

Different means of composing machines are available. Furthermore, the B-method heavily relies on abstraction, i.e., the step-wise refinement of very abstract machines towards more concrete implementations. In addition to machines, Event-B features contexts, supposed to hold static information.

Machine operations (or events, in case of Event-B) are used to specify transitions between states. A machine operation has a unique name and consists of B substitutions defining the state after execution. An operation can have a precondition allowing or prohibiting execution based on the current state. Operations can be non-deterministic and might be nested. Furthermore, B features a multitude of different substitutions, including if-then-else constructs and while loops. Event-B's events are considerably simpler and can only include guards for execution and variable assignments.

To ensure correctness of a specification, the user can define machine invariants, i.e., safety properties that have to hold in every state. Depending on the tool used, these properties can be verified either by formal proof or using model checking. In addition, LTL properties can be specified to verify temporal behavior.

Besides using the types explicitly provided by the B language specification, one can introduce used-defined types in the form of sets. A set is defined by a unique name and may be initialized by a finite enumeration of distinct elements.

[1] Regarding the trade-off between ease of implementation and efficient execution see [16] for a general point of view and [22] for a perspective on B and Event-B.

Sets not defined by enumeration are called deferred sets and are assumed to be non-empty and finite.

3 Software Used

Both for our courses and the case studies presented below we rely on three tools for development and verification of formal models. Each supports different verification techniques, such as model checking and proof. When writing specifications in Event-B, one often combines all of them instead of using only a single verification tool. Consequently, integrations into one another have been developed. In detail, the tools we use are:

– ProB [24–26], a constraint solver, model finder and model checker for the B family of languages. One of the key features of ProB is fully automatic animation of specifications, i.e., the user can traverse the state space without having to supply values for variables or parameters. In addition, ProB incorporates different model checking techniques, including explicit state and symbolic ones [17]. Both model checking and animation are driven by a backend written in SICStus Prolog [6], relying mainly on its constraint logic programming based solving library [7]. The Prolog kernel is supported by integrating SMT solvers [18] and SAT solvers [31] via Kodkod [34]. ProB supports LTL model checking using a tableau-based algorithm as outlined in [11,30].
– Rodin [3] is an IDE for Event-B implemented on top of Eclipse. It features generation of proof obligations, e.g., for invariant preservation, and can be combined with different provers for discharging them. In particular, one can connect the Atelier-B provers [8] and SMT solvers [12,13]. Rodin does not directly support visualization of models. Instead, ProB is provided as a plugin [5].
– BMotionWeb [20,21] is a tool for the rapid creation of formal prototypes on top of B and Event-B machines. While ProB supports basic visualizations of formulas, individual states and the state space, more involved visualizations and software prototypes are realized using BMotionWeb. In particular, it allows linking a graphical user interface to a formal specification animated by ProB.

Below, we present three case studies in which we applied the formal approach of [20] to prototyping games. In Sect. 4, we present a prototype of Pac-Man, while in Sect. 5, we are modeling chess. Afterwards, in Sect. 6, a version of Lightbot is presented. The case studies outline the broad applicability of formal methods to games: with Pac-Man, we have a game featuring continuous and simultaneous movement. In contrast, chess represents turn-based board games where players move in succession. Finally, the Lightbot game presents how stack-based programming languages can be implemented.

4 Pac-Man

As a first case study, we use the well-known classic arcade game Pac-Man. We will start with the set of requirements to be verified in our model in Sect. 4.1. The requirements are posed to the students in the same way, e.g., as a practical specification task.

The Event-B model[2] is discussed in Sect. 4.2. On top of the model, we used visualization techniques to implement an interactive and playable prototype as discussed in Sect. 4.3. In the background, the model checker drives a simple artificial intelligence controlling the ghosts as described in Sect. 4.4.

4.1 Requirements

We try to keep the set of requirements simple and easy to grasp to help students focus on applying the formal method rather than spending time on implementation or specification details. Furthermore, we abstract further from the original Pac-Man: Instead of being continuous, movement is made discreet, i.e., Pac-Man and ghosts move on a grid of fields.

rq1 Pac-Man can only be moved from one field of the grid to a direct neighbor field. This implies that it cannot jump to another position in the level.
rq2 Each ghost can only be moved from one field to a direct neighbor field.
rq3 Pac-Man can only be moved when every ghost, that already left the ghost's den, has moved at least once after the last movement of Pac-Man.
rq4 Pac-Man can be moved through a tunnel.
rq5 The first two ghosts must start before Pac-Man starts.
rq6 The third/fourth ghost must start as soon as 30/180 dots are collected.
rq7 Each dot can only be collected once.
rq8 If Pac-Man and a ghost are on the same field, one must catch the other.
rq9 If a ghost catches Pac-Man, the player loses a life.

4.2 Model and Refinement Hierarchy

The model of Pac-Man is split into different refinement levels, each introducing additional detail to the game. The first refinement level adds Pac-Man's movement, the second deals with collecting as well as scoring dots and the third and fourth refinement levels consider moving the ghosts and hunting.

Pac-Man's Movement. Initially, on the first refinement level, we specify the movement of Pac-Man, focusing on requirements containing constraints immediately applicable to movement: **rq1** and **rq4**.

The maze in which Pac-Man moves is encoded as a set containing the coordinates of each field. We use two variables to store Pac-Man's position, one for its

[2] A full version of the model can be found at: https://github.com/pkoerner/EventBPacman-Plugin/tree/master/eventb.

current and one for its prior location. Both are members of the set of fields in the maze as stated using type invariants inv101 and inv102 shown below. Combined, the invariants state that Pac-Man never leaves the maze.

The last position is kept track of in order to add an invariant inv103a: it can be used to prohibit jumping over squares. This is sufficient to verify **rq1**. Using LTL rather than state invariants, the requirement could also be checked without introducing an additional variable.

```
inv101:  $pos \in \boldsymbol{maze}$                                                    // current position
inv102:  $prior\_pos \in \boldsymbol{maze}$                                                // last position
inv103a: $(\mathrm{prj}_1(pos) - \mathrm{prj}_1(prior\_pos) \in \{2, -2\} \wedge \mathrm{prj}_2(pos) = \mathrm{prj}_2(prior\_pos)) \vee$
         $(\mathrm{prj}_2(pos) - \mathrm{prj}_2(prior\_pos) \in \{2, -2\} \wedge \mathrm{prj}_1(pos) = \mathrm{prj}_1(prior\_pos))$
```

Additionally, the model includes two Boolean variables indicating whether Pac-Man was moved in the last event and whether it was moved at all. The latter will allow us to verify **rq5** later on. Furthermore, we can already specify that Pac-Man may move through a tunnel to the opposite side of the grid. This is done by introducing another Boolean variable, which indicates whether the last movement was through the tunnel. We add this variable as a guard to invariant inv103a resulting in inv103 to allow jumps in case of tunnel traversal. We also state that if the tunnel is used, Pac-Man has to appear on the other side in inv107.

```
inv106: $tunneled \in \mathrm{BOOL}$                              // last movement was through tunnel
inv103: $(moving = \top \wedge tunneled = \bot) \Rightarrow$
        $(\,(\mathrm{prj}_1(pos) - \mathrm{prj}_1(prior\_pos) \in \{2, -2\} \wedge \mathrm{prj}_2(pos) = \mathrm{prj}_2(prior\_pos)) \vee$
         $(\mathrm{prj}_2(pos) - \mathrm{prj}_2(prior\_pos) \in \{2, -2\} \wedge \mathrm{prj}_1(pos) = \mathrm{prj}_1(prior\_pos)))$
inv107: $tunneled = \top \Leftrightarrow \{pos, prior\_pos\} = \boldsymbol{tunnel}$
```

An example of an Event-B event concerned with movement is shown below.

```
Event move_up ⟨ordinary⟩ $\widehat{=}$
  any
      p
  where
      grd101: $p \in \boldsymbol{accessible}$
      grd102: $(\mathrm{prj}_1(position) = \mathrm{prj}_1(p)) \wedge (\mathrm{prj}_2(position) - \mathrm{prj}_2(p)) = 2$
      grd103: $position \in \boldsymbol{tunnel} \Rightarrow p \notin \boldsymbol{tunnel}$
  then
      act101: $position\_before := position$
      act102: $position := p$
      act103: $moving := \top$
      act104: $tunneled := \bot$
      act105: $moved := \top$
  end
```

Collecting Dots. In a second refinement, we introduce how collecting dots and thus scoring points works. The corresponding requirements are **rq6** and **rq7**. In this step, we also add the four super pills which enable Pac-Man to catch ghosts. Furthermore, we add distinct events for moving Pac-Man to an empty field or to a dot.

The first three invariants introduced in this refinement state that the score is a natural number (inv201 and inv202) and, furthermore, that it can only be increased by either 10 or 200 points or remain the same value (inv203).

inv201: $score \in \mathbb{N}$ // current score
inv202: $score\_before \in \mathbb{N}$ // last score
inv203: $score - score\_before \in \{0, 10, 200\}$ // possible values for increment
inv204: $scoredots\_current \in \mathbb{P}(scoredots)$ // uncollected small dots
inv205: $superpills\_current \in \mathbb{P}(superpills)$ // uncollected super pills
inv206: $counter\_scored = \mathrm{card}(scoredots) - \mathrm{card}(scoredots\_current)$ // dots collected

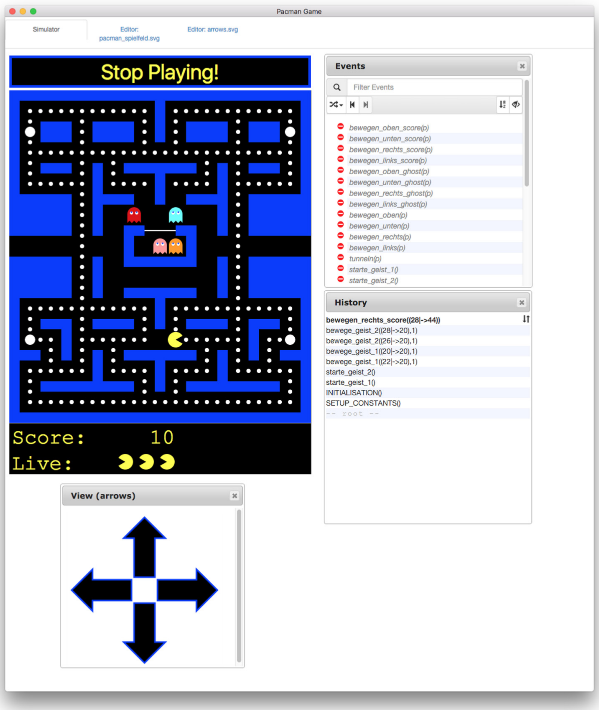
Invariants inv204 and inv205 give type information for the collectible dots. Additionally, we count the amount of dots collected in *counter_scored*. Then, inv206 ensures that no dots get lost, i.e., that every collected dot is awarded to the player's score.

The Ghosts, Lives and Hunting. After movement and scoring, a third refinement is used to add the ghosts. This includes their movement as well as catching Pac-Man. Variables, invariants and events corresponding to the ones we added for Pac-Man's rules of movement are added for each ghost as well.

This refinement step satisfies **rq2** in the same way **rq1** was implemented earlier. Additional invariants are used to ensure **rq6**. Again, we can check **rq5** with an LTL formula: for all possible paths through the state space, the ghosts have to move before the Pac-Man can start. In another refinement step, we introduce that ghosts can be hunted when Pac-Man collects a super pill: once collected, all ghosts are added to a set of currently catchable ghosts. Once caught, they are removed from this set. Additional guards are added to the movement events to ensure only huntable ghosts can actually be caught. Otherwise, the ghost will catch Pac-Man. Furthermore, we added the lives the player has and that he loses one if he gets caught. Invariant inv402 ensures that Pac-Man cannot gain an infinite amount of lives. Moreover, inv404 implies that Pac-Man cannot gain an additional life and only can lose at most one life at a time. Furthermore, in this refinement step, we can check both **rq8** and **rq9** using LTL formulas.

inv402: $lives \leq \boldsymbol{start\_lives}$ // current lives
inv404: $lives\_old - lives \in \{0, 1\}$ // at most one life is removed
inv406: $chased\_ghosts \subseteq \{\boldsymbol{ghost\_1}, \boldsymbol{ghost\_2}, \boldsymbol{ghost\_3}, \boldsymbol{ghost\_4}\}$ // hunted ghosts

The last refinement step adds further movement rules. These rules enforce the order of movement, e.g., that the second ghost moves after the first. Additionally, to satisfy **rq3**, Pac-Man can only be moved once all ghosts have been moved.



Events

```
bewegen_oben_score(p)
bewegen_unten_score(p)
bewegen_rechts_score(p)
bewegen_links_score(p)
bewegen_oben_ghost(p)
bewegen_unten_ghost(p)
bewegen_rechts_ghost(p)
bewegen_links_ghost(p)
bewegen_oben(p)
bewegen_unten(p)
bewegen_rechts(p)
bewegen_links(p)
tunneln(p)
starte_geist_1()
starte_geist_2()
```

History

```
bewegen_rechts_score((28|->44))
bewege_geist_2((28|->20),1)
bewege_geist_2((26|->20),1)
bewege_geist_1((20|->20),1)
bewege_geist_1((22|->20),1)
starte_geist_2()
starte_geist_1()
INITIALISATION()
SETUP_CONSTANTS()
-- root --
```

Fig. 1. Pac-Man visualization

4.3 Visualization

After the model is initialized, the visualization depicted in Fig. 1 shows the maze and Pac-Man as well as the ghosts shortly after their initial positions. In addition, the score value is shown together with the three small Pac-Mans representing the lives of the player. The visualization reacts to changes of the model using the observer pattern of BMotionWeb, i.e., we register observers for the variables and define how the elements of the visualization react to state space changes.

Usually, events are executed by the user by selecting them from a list of enabled events. To get closer to a playable prototype, we added four arrow buttons. We registered all events to move the Pac-Man in a direction to each of the corresponding arrow buttons. When the user clicks on one of the buttons, BMotionWeb executes the event if it is enabled in the current state.

Additionally, we implemented a listener for key events, enabling the user to play the model just like the real game by using arrow keys.

4.4 Adding a Simple Game AI on Top of Formal Models

Pac-Man also served as a playground for two novel research directions:

– Can the prototypical model made playable without further code generation, i.e., by executing it directly using the model checker? In particular, we were interested to see how user interaction could be designed and what level of responsiveness could be reached. This line of research later lead to a general implementation of runtime usage of B models [19].
– Furthermore, Pac-Man allowed us to experiment with state-space search algorithms beyond simple depth-first or breath-first traversal.

To gain a test-bed for both questions, we use the Groovy API of BMotionWeb in order to implement a simple AI that is able to control the Pac-Man and the four ghosts. It supports three different modes of operation: First, if the user plays with the arrow keys, the AI lets Pac-Man move in the given direction until it hits a wall and moves the ghosts each tick. Secondly, the user may click the Play!-button and the AI plays the game against itself. This means the Pac-Man and the ghosts are completely controlled by the AI. Lastly, the AI can be used in order to move the ghosts automatically after the user moved the Pac-Man.

In the first two cases, we run a loop in a thread until the user stops it or the game is over. It moves both the Pac-Man and each of the ghosts. While in the first case, the Pac-Man simply follows its current direction, in the second case the AI decides where Pac-Man should turn. As a heuristic, we use a breadth-first search in order to find the nearest dot to score. This directly corresponds to the search strategy used in the underlying model checker.

In the last case, the model checkers search strategy is only used to control how the four ghosts are moved. After a specific operation is executed by the used controlling Pac-Man, we again use breadth-first search to identify possible paths for the ghosts.

5 Chess

As a second case study, we implemented the well-known board game chess in B[3]. Again, we will discuss our model and the set of requirements and continue with visualization and the integration of a game-playing AI in our model.

5.1 Requirements

When posed as a specification task to students, we usually provide the following set of requirements, leading to a prototype that can be considered playable:

rq1: Pieces can only be moved in their specific way (e.g., a king can only move exactly one field into any direction).

[3] The main B machine can be found at: https://github.com/pkoerner/b-chess-example/blob/master/b/board.mch.

rq2: If the king is in check, only moves getting the king out of check are permitted.
rq3: No piece can be moved outside the 8×8 board.
rq4: Special moves (Castling, En Passant and Promotion) follow the rules.
rq5: If the king cannot be defended immediately, the game is lost.
rq6: If no legal move is possible for one player, the game is considered as a draw.
rq7: Both players have the same set of pieces and the white player has the first move.

5.2 Model and Refinement Hierarchy

Rather than relying on refinement as done with the Event-B specification of Pac-Man, we use the modularization capabilities of classical B and split our model into a model containing the board, another for visualization and a third containing basic variables and sets.

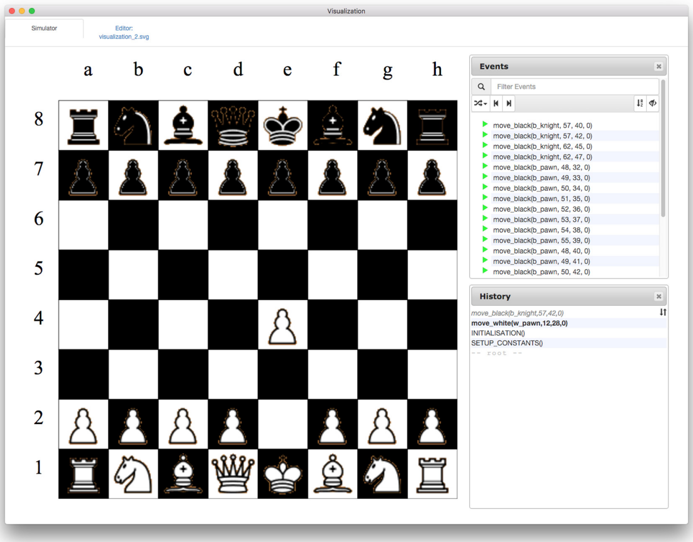
There are two different ways to specify the board: A *piece-centric* approach associates all pieces with the field they occupy, e.g., $white\ king \rightarrow e1$. In contrast, a *square-centric* approach maps each field on the board to the piece on it. This could be done using a partial function (to avoid mapping empty files to placeholders) or using a total function (which can be beneficial for constraint solving and visualization). In this case study, we opted for a square-centric representation using a total function. We accept the corresponding overhead in order to find empty fields more easily.

Movement. Moving pieces is encoded using B operations, i.e., each move results in a state transition. Four different B operations are introduced: movement for black and white pieces each and taking a piece, again with individual operations for black and white. Special moves, such as *castling*, *En Passant* or *promoting a pawn* are added to the model in further refinement steps. Their preconditions share common predicates, checking if the figure/field combination exists, if a movement path is feasible and if the player is not in chess.

`grd_tuple`: $x \in dom(board)$ // Field x exists in board and it maps to a white or black piece
`grd_check`: $not\_in\_check(new\_board)$ // The player is not in check after the move

Furthermore, operations have to distinguish between *moving* and *taking* a piece: When taking a piece, it suffices to check whether the movement is valid, i.e., according to the rules and all fields in between are empty. Simply *moving* a piece, however, requires an additional precondition to check whether the target field is empty.

`grd_move`: move_white_piece(piece,x,y,take,board) // move respects movement rules
`grd_fields`: $\forall field \in between(x, y).free(field)$ // fields on the way and target are empty
`grd_take`: $take = 1 \Rightarrow board(y) =$ opponent_piece // if taking: there is an opponent piece



```
move_black(b_knight, 57, 40, 0)
move_black(b_knight, 57, 42, 0)
move_black(b_knight, 62, 45, 0)
move_black(b_knight, 62, 47, 0)
move_black(b_pawn, 48, 32, 0)
move_black(b_pawn, 49, 33, 0)
move_black(b_pawn, 50, 34, 0)
move_black(b_pawn, 51, 35, 0)
move_black(b_pawn, 52, 36, 0)
move_black(b_pawn, 53, 37, 0)
move_black(b_pawn, 54, 38, 0)
move_black(b_pawn, 55, 39, 0)
move_black(b_pawn, 48, 40, 0)
move_black(b_pawn, 49, 41, 0)
move_black(b_pawn, 50, 42, 0)
```

```
move_black(b_knight,57,42,0)
move_white(w_pawn,12,28,0)
INITIALISATION()
SETUP_CONSTANTS()
-- root --
```

Fig. 2. Chess visualization

Check, Checkmate and Draw. In order to implement check, one needs to look one step ahead to find out if an opponent piece could take the king. This impacts performance, as for every possible move every possible opponent move might have to be calculated twice: first, when checking whether the move should be enabled or not and again after executing the operation. Additionally, we decided to encode checkmate as an invariant violation. One of the invariants claims that one white king and one black king are part of the match at all times. If one of them is taken, the invariant is violated and model checking stops.

A draw can be reached in various ways:

- both players agree,
- 50 moves have passed without moving a pawn or taking a piece,
- the situation on the board is deadlocked and the same position is reached too often.

While the number of moves can be tracked using an integer variable, keeping track of all prior positions leads to a combinatorial explosion, effectively rendering model checking impossible.

5.3 Visualization

To visualize the chess board and let the user play, we again rely on BMotionWeb. On the left side of Fig. 2, the visualization itself is placed. Clicking on a piece and a target field triggers the corresponding operation.

An operation can be evaluated manually by clicking on one of those listed inside the events window on the right-hand side. The history of executed events is shown below, the user can return to a former state by clicking on one of the operations listed there. By doing so, the trace rolls back to state after the operation was executed.

5.4 Minimax as Model Checking Heuristic

Minimax is a game-independent algorithm, i.e., its implementation only differs in the game-specific evaluation functions used to determinate a value for each leaf node. For chess, we could consider the number of pieces left for each player or the value of own pieces compared to opponent pieces (e.g., a queen is more valuable than a pawn). Furthermore, one could evaluate the number of reachable fields or movability.

While using as much information as possible in general leads to a stronger AI, it also renders computing the evaluation function more difficult. As a tradeoff, we decided on the following information and weights:

- Values of figures residing on the board following the valuation by Shannon E. Claude [33].
- Number of pawns in desired positions, e.g., passed pawns as well as number of pawns in undesired positions, e.g., doubled pawns.
- The number of semi-open files, i.e., the number of rows or columns the player's rooks can move at least five fields into one direction on. This is a measure rock movability, which indicates how well players can bring their rooks into play. We multiply the measure with a weight of 2.
- We count how well the fields adjacent to the own king are guarded, again applying a weight of 2.
- We measure to what extent a player controls the four squares in the center of the field. As they are usually crucial to winning the game, we apply a weight of 3.

To prevent the model checker from running too long, a relatively small *search depth* is set. Essentially, this is done by performing a depth-limited exploration of the state space and applying the evaluation function of Minimax to the reached states. The highest ranking states are then explored further, effectively driving the model checker along the path a depth-restricted Minimax would have taken. As commonly done in chess engines, paths which might yield a better situation but are too long are not considered further. At the same time, the value of a state is only influenced by the best movements the opponent can make, i.e., Minimax implicitly follows the best strategy of both players and thus is not influenced by good states that have a single bad successor state.

Since a lost game results in an invariant violation, we can now use model checking to find a playing strategy. Using ProB, we try to find a path leading to a checkmate and, in consequence, a win. Due to the inherent combinatorial complexity of chess, state spaces are usually too large to be explored exhaustively. In the future, we want to study the effect of state space reduction techniques such as partial order reduction on the performance of game prototypes.

6 Lightbot

Lightbot[4] is not as universally known as Pac-Man or chess: It is an educational puzzle game with the aim of programming a robot so that it follows a specific path through a grid. On its way, it has to light up several tiles of the grid to reach the overall objective.

The instruction set to control the robot is fairly small, i.e. moving and turning the robot, letting the robot jump upwards or downwards, toggling special fields and calling specified procedures. However, this small instruction set is sufficient to implement recursive programs as well as loops and builds a Turing-complete language.

While the two former case studies have been created during Bachelor and Master theses, we have used Lightbot as a mandatory assignment in our course on safety critical systems several times. Usually, students had to specify a formal model of Lightbot including a (playable) visualization to be allowed to take the final exam. In particular, we required the model to be parametric, in the sense that it should be possible to add and change the robot's programming during execution. More general, this implies that students had to specify a model of the interpreter of Lightbot's programming language.

6.1 Requirements

The rules of the game can be best explained in form of requirements:

rq1: The robot moves on a three-dimensional board.
rq2: The game is generic, i.e., different levels (boards) are supported and can be provided and switched in some way.
rq3: The robot supports all moves (forward, toggle light, left/right turn, jumping and entering one of two sub-procedures).
rq4: The robot starts execution in the main-procedure.
rq5: A program stack is required to execute the user-defined sub-routines, as the may be mutually recursive. Again, this underlines the idea that students do in fact specify the internal workings of an interpreter.
rq6: The lowest elevation level is 1.
rq7: Starting position and the tiles the robot has to light up to complete the level are described in the level itself, not hard-coded in the interpreter.

6.2 Refinement Hierarchy

As we expect our students to follow the formal modeling process as a whole, we do not provide a particular refinement hierarchy upfront.

Our reference specification[5] however starts with modeling a two-dimensional grid that the robot moves upon. In that stage, moving up, down, left and right

[4] https://lightbot.com/.
[5] Available at: https://www3.hhu.de/stups/models/fmfun19/lb.zip.

is allowed if the robot faces the corresponding direction. It is also possible to turn the robot, and to light up specific tiles.

The second refinement steps adds a third dimension. This adds two different aspects to the game. First, simple movement is now blocked in case the elevation of the adjacent square is different. Second, a new kind of movement is introduced as the robot has to jump in order to move vertically. This refinement level completes the basic execution engine. It is possible to execute all enabled commands whenever one likes, with no constraints concerning a program counter or limited amount of memory.

The third refinement level is used to introduce the actual programming of the robot. We use an Event-B context to describe the level (elevation and tiles to light up), as well as the starting position of the robot and the direction it faces. Additionally, the context is used to constrain how large individual procedures implemented by the player may be.

The corresponding Event-B machine specifies how programs are specified and executed, i.e., the program has to be written beforehand and, upon interpretation, only operations at the current program counter may be executed. Additionally, a program stack is added that stores the program counters once sub-procedures are called and resumes execution upon returning.

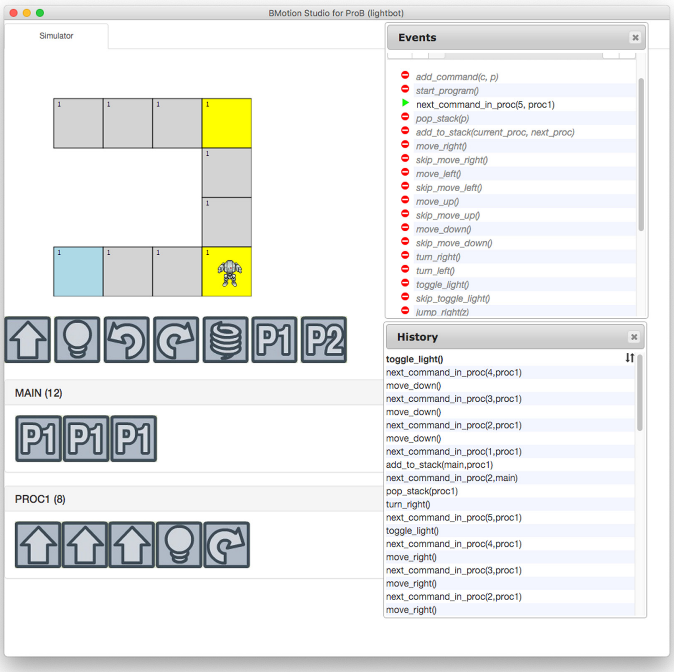
6.3 Visualization

As with chess and Pac-Man, the current state of the game can be immediately identified when looking at a visualization instead of pretty prints of the underlying data structures. The visualization in Fig. 3 also relies on BMotionWeb.

On the top left, the grid with the robot and tiles that are to be switched on (blue) and the ones already lit (yellow) is shown. Underneath, all available commands are given next to each other. Again, the game is fully playable using the visualization, e.g., one can select a procedure to add instructions to and modify it at will. The current code of the main procedure and all sub-procedures is given below.

The history Event-B events executed by the model checker (both during the construction and execution of the program) as well as all events that can be executed in the current state are shown on the right-hand side. As with the original game, once the robot begins executing the player-given code, only two Events are still permitted to be executed: fetching the next instruction and executing it.

Our reference specification and its visualization can be used in order to explain the game to the students, before they have to implement it on their own. As it is much easier to reason about a concept that one is familiar with rather than something given from an informal text-based representation, this assists students a lot in the early phase.



```
Events
add_command(c, p)
start_program()
next_command_in_proc(5, proc1)
pop_stack(p)
add_to_stack(current_proc, next_proc)
move_right()
skip_move_right()
move_left()
skip_move_left()
move_up()
skip_move_up()
move_down()
skip_move_down()
turn_right()
turn_left()
toggle_light()
skip_toggle_light()
jump_right(z)
```

```
History
toggle_light()
next_command_in_proc(4,proc1)
move_down()
next_command_in_proc(3,proc1)
move_down()
next_command_in_proc(2,proc1)
move_down()
next_command_in_proc(1,proc1)
add_to_stack(main,proc1)
next_command_in_proc(2,main)
pop_stack(proc1)
turn_right()
next_command_in_proc(5,proc1)
toggle_light()
next_command_in_proc(4,proc1)
move_right()
next_command_in_proc(3,proc1)
move_right()
next_command_in_proc(2,proc1)
move_right()
```

Fig. 3. Lightbot visualization (Color figure online)

6.4 Models of Virtual Machines

The original game is an educational game on coding. It is used in order to teach basic programming concepts, such as function calls, recursion and loops.

Following this idea, writing a specification of the game itself (as opposed to a specification of the player-given code to solve a level) teaches the same aspects on a meta-level, i.e., how to *model* and *verify* function calls, recursion and loops. Thus, students learn how to model programming languages and their interpreters.

The same concept could later be applied to "real" programming languages with more sophisticated semantics.

7 Related Work

In his literature review on the state of teaching formal methods in academia, Zhumagambetov identifies several challenges in teaching formal methods [35]. First, students are often skeptical of the usefulness of formal methods. We believe, that our programming projects can help overcome this scepticism by stressing the value of formal methods for implementing games, in particular for getting the game mechanics right. Second, the steep learning curve and little feedback of formal method tools is discussed. Again, we believe our strong focus on visualization and immediately playability of prototypes helps overcome this burden. In general, Zhumagambetov suggests to use real-life examples and gamification to improve formal methods courses, which both can easily be achieved following our approach to teaching [35].

At FMFun 2019, Schlingloff suggested to teach model checking by referencing games and puzzles [32]. In particular, he also used chess as an example for motivation model checking.

Teaching formal methods concepts by relying on card games and card tricks rather than artificial examples has been considered by Curzon and McOwan [9]. They discuss numerous tricks and small games that visualize the concept of invariants, etc.

In his dissertation [27], Timo Nummenmaa already considered implementing game prototypes using formal software development techniques. Both in the dissertation and in the related publications [28,29] the possible impact of using formal methods for game development are discussed. In particular, the authors especially mention the benefit of executable formal models, as we provide by the combination of B and ProB. We were able to extend upon the former work thanks to BMotionWeb: we can provide richer and more interactive visualization, closer to the intended game design itself.

Formal verification of properties of checkers has been considered in [4]. The authors encode the game as a finite state system and search for winning strategies using symbolic model checking. In contrast to our work, the focus is on properties of the game itself, rather than creating playable prototypical implementations. However, [4] underlines that state spaces of (albeit simple) board games can be handled by current model checkers.

In [15], the author uses the HOL4 theorem prover [14] to verify chess endgame databases. To do so, an encoding of possible moves similar to the one in Sect. 5 is used. Instead of using a model checker to find and evaluate possible moves, the correctness of predefined move sequences given in endgame databases is verified.

Instead of using verification techniques to encode games, [10] considers the opposite way: Verification tasks are encoded as games, that could later be solved by people unaccustomed to software verification. Software and security constraints are represented by a simple puzzle-like game, which solution represents either failure or successful verification.

Directed model checking using different heuristics has been considered in the context of ProB in [23]. Comparable to the approach used in Sect. 5, the authors use state properties to control which state ProB's model checker expands next. However, heuristics are not as involved as the Minimax algorithm employed in this paper.

8 Conclusions and Future Work

In summary, our three case studies have shown both advantages and shortcomings of the tools introduced in Sect. 3:

- ProB (or any model checker with animation capabilities) is very important during development. (Bounded) model checking of the specification usually gives fast feedback about the correctness of a specification or an implementation. Animation, in particular with an added visualization on top, allows reassuring a developer that changes made to the specification behave as intended. Sometimes, the tool cannot cope with the entire state spaces though: e.g., assumptions about chess based on the rules cannot be model-checked, as the state space is way too large.
- Student feedback concerning Rodin is rather negative: while it provides a type checker, a proof obligation generator and proof system with some automated proof rules, usability is lacking. Sometimes, Rodin is in an inconsistent state where, e.g., POs are not generated as they should be and a cleaning mechanism has to be invoked. Also, as Event-B files are not plain text, structural editors are default. Many students find it uncomfortable to switch between text boxes in the IDE rather than navigating with arrow keys. Furthermore, some functions such as removing certain elements are hidden in context menus that only pop up when right-clicking on very specific positions. Finally, the files do not integrate well with version control systems such as git.
- BMotionWeb is a great tool in order to explain specifications to domain experts or students, once the visualization and the model are complete. An application based entirely on web technologies proved to be hard to use though. When errors occur, it is not clear in which layer the cause is located in: is it an error in the B model? Is an SVG file broken? Is the config file incorrect? Is there a bug in the JavaScript code? As some errors are not reported, development can be cumbersome if one is not an expert in all technologies that are used by BMotionWeb.

When applied to the development of game prototypes, they support using classic formal proof and model checking to verify the correctness of game implementations. In particular, we have proven both high-level properties about the game's implementation itself and the correct representation of the rules of a game.

As we mentioned in Sect. 4.2, playability of game prototypes is limited, because it is hard to achieve continuous movement in an Event-B model. Nevertheless, they make for easy to understand and highly motivating examples for students trying to work their way into formal methods. Turn-based games however, such as chess or Lightbot, are a great match for "slower" execution due to interpretation overhead and can be fun and engaging to interact with.

Using BMotionWeb, we have the possibility to animate and visualize our prototypes. As we have shown, BMotionWeb is able to produce playable prototypes of both real-time and round-based games. However, the visualization behaves quite slow and is thus not usable in presence of time limits. While this is less critical for teaching and for implementing board games like chess, it limits the applicability of our approach to games in general.

8.1 Impact on Student Learning

It is hard to measure the influence on how interest, attention and understanding is enabled for students. There is no clear trend that correlates with introduction of games as examples: overall student feedback remained the same. The average grades improved significantly after introducing mandatory projects based on Lightbot. However, in the following years, exams fell off in quality without changing the contents. Upon introduction of other examples, the average grade improved significantly again.

It may be that breaking the routine of the teaching personnel is more engaging for students. It also is possible that some versions of the projects were shared between students over years, and parts were copied, resulting in students missing crucial learning outcomes.

Overall, we conclude that teaching – as well as learning – formal methods is hard. Thus, efforts should be taken to improve student engagements. Using games as examples is only one of several possible methodologies.

Acknowledgement. We thank Christoph Heinzen who created several versions of the Pac-Man case study, as well as Philip Höfges for the chess model, AI and GUI.

References

1. Abrial, J.-R.: The B-Book. Cambridge University Press, Cambridge (1996)
2. Abrial, J.-R.: Modeling in Event-B: System and Software Engineering. Cambridge University Press, Cambridge (2010)
3. Abrial, J.-R., Butler, M., Hallerstede, S., Hoang, T.S., Mehta, F., Voisin, L.: Rodin: an open toolset for modelling and reasoning in Event-B. STTT **12**(6), 447–466 (2010)
4. Baldamus, M., Schneider, K., Wenz, M., Ziller, R.: Can American checkers be solved by means of symbolic model checking? Electron. Notes Theoret. Comput. Sci. **43**, 3–17 (2000)
5. Bendisposto, J., Leuschel, M., Ligot, O., Samia, M.: La validation de modèles Event-B avec le plug-in ProB pour RODIN. TSI **27**(8), 1065–1084 (2008)
6. Carlsson, M., Mildner, P.: SICStus Prolog-the first 25 years. TPLP **12**(1–2), 35–66 (2012)
7. Carlsson, M., Ottosson, G., Carlson, B.: An open-ended finite domain constraint solver. In: Glaser, H., Hartel, P., Kuchen, H. (eds.) PLILP 1997. LNCS, vol. 1292, pp. 191–206. Springer, Heidelberg (1997). https://doi.org/10.1007/BFb0033845
8. ClearSy. Atelier B, User and Reference Manuals (2016). http://www.atelierb.eu/
9. Curzon, P., McOwan, P.W.: Teaching formal methods using magic tricks. In: Fun with Formal Methods: Workshop at the 25th International Conference on Computer Aided Verification, Number 122 (2013)
10. Dietl, W., et al.: Verification games: making verification fun. In: Proceedings FTfJP 2012, pp. 42–49. ACM (2012)
11. Dobrikov, I., Leuschel, M., Plagge, D.: LTL model checking under fairness in ProB. In: De Nicola, R., Kühn, E. (eds.) SEFM 2016. LNCS, vol. 9763, pp. 204–211. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-41591-8_14

12. Déharbe, D., Fontaine, P., Guyot, Y., Voisin, L.: SMT solvers for Rodin. In: Derrick, J., et al. (eds.) ABZ 2012. LNCS, vol. 7316, pp. 194–207. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-30885-7_14
13. Déharbe, D., Fontaine, P., Guyot, Y., Voisin, L.: Integrating SMT solvers in Rodin. Sci. Comput. Program. **94**, 130–143 (2014). Part 2(0)
14. Gordon, M.J.C.: HOL: a proof generating system for higher-order logic. In: Birtwistle, G., Subrahmanyam, P.A. (eds.) VLSI Specification, Verification and Synthesis. SECS, vol. 35, pp. 73–128. Springer, Boston (1988). https://doi.org/10.1007/978-1-4613-2007-4_3
15. Hurd, J.: Formal verification of chess endgame databases. Technical report, Oxford University Computing Laboratory (2005)
16. Kennedy, K., Koelbel, C., Schreiber, R.: Defining and measuring the productivity of programming languages. Int. J. High Perform. Comput. Appl. **18**(4), 441–448 (2004)
17. Krings, S., Leuschel, M.: Proof assisted symbolic model checking for B and Event-B. In: Butler, M., Schewe, K.-D., Mashkoor, A., Biro, M. (eds.) ABZ 2016. LNCS, vol. 9675, pp. 135–150. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-33600-8_8
18. Krings, S., Leuschel, M.: SMT solvers for validation of B and Event-B models. In: Ábrahám, E., Huisman, M. (eds.) IFM 2016. LNCS, vol. 9681, pp. 361–375. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-33693-0_23
19. Körner, P., Bendisposto, J., Dunkelau, J., Krings, S., Leuschel, M.: Integrating formal specifications into applications: the ProB Java API. Form. Methods Syst. Des. (2020). https://doi.org/10.1007/s10703-020-00351-3
20. Ladenberger, L.: Rapid creation of interactive formal prototypes for validating safety-critical systems. Ph.D. thesis, Heinrich-Heine-Universität Düsseldorf (2017)
21. Ladenberger, L., Leuschel, M.: BMotionWeb: a tool for rapid creation of formal prototypes. In: De Nicola, R., Kühn, E. (eds.) SEFM 2016. LNCS, vol. 9763, pp. 403–417. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-41591-8_27
22. Leuschel, M.: The high road to formal validation. In: Börger, E., Butler, M., Bowen, J.P., Boca, P. (eds.) ABZ 2008. LNCS, vol. 5238, pp. 4–23. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-87603-8_2
23. Leuschel, M., Bendisposto, J.: Directed model checking for B: an evaluation and new techniques. In: Davies, J., Silva, L., Simao, A. (eds.) SBMF 2010. LNCS, vol. 6527, pp. 1–16. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-19829-8_1
24. Leuschel, M., Bendisposto, J., Dobrikov, I., Krings, S., Plagge, D.: From animation to data validation: the ProB constraint solver 10 years on, Chap. 14. In: Boulanger, J.-L. (ed.) Formal Methods Applied to Complex Systems: Implementation of the B Method, pp. 427–446. Wiley ISTE (2014)
25. Leuschel, M., Butler, M.: ProB: a model checker for B. In: Araki, K., Gnesi, S., Mandrioli, D. (eds.) FME 2003. LNCS, vol. 2805, pp. 855–874. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-540-45236-2_46
26. Leuschel, M., Butler, M.: ProB: an automated analysis toolset for the B method. STTT **10**(2), 185–203 (2008)
27. Nummenmaa, T.: Executable formal specifications in game development. Dissertation, University of Tampere (2013)
28. Nummenmaa, T., Berki, E., Mikkonen, T.: Exploring games as formal models. In: Proceedings SEEFM 2009, pp. 60–65 (2009)

29. Nummenmaa, T., Kuittinen, J., Holopainen, J.: Simulation as a game design tool. In: Proceedings ACE 2009, pp. 232–239. ACM (2009)
30. Plagge, D., Leuschel, M.: Seven at one stroke: LTL model checking for high-level specifications in B, Z, CSP, and more. Int. J. Softw. Tools Technol. Transf. **12**(1), 9–21 (2010)
31. Plagge, D., Leuschel, M.: Validating B, Z and TLA+ using ProB and Kodkod. In: Giannakopoulou, D., Méry, D. (eds.) FM 2012. LNCS, vol. 7436, pp. 372–386. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-32759-9_31
32. Schlingloff, B.-H.: Teaching model checking via games and puzzles. In: Pre-proceedings FMFUN 2019 (2019)
33. Shannon, C.E.: Programming a computer for playing chess. In: Levy, D. (ed.) Computer Chess Compendium, pp. 2–13. Springer, New York (1988). https://doi.org/10.1007/978-1-4757-1968-0_1
34. Torlak, E., Jackson, D.: Kodkod: a relational model finder. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 632–647. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-71209-1_49
35. Zhumagambetov, R.: Teaching formal methods in academia: a systematic literature review. In: Pre-proceedings FMFUN 2019 (2019)

Teaching Model Checking via Games and Puzzles

Bernd-Holger Schlingloff[1,2](✉)

[1] Humboldt-Universität zu Berlin, Berlin, Germany
hs@informatik.hu-berlin.de
[2] Fraunhofer FOKUS, Berlin, Germany

Abstract. Puzzles and games give a strong motivation for humans to deal with formal objects: people spend hours and hours in seemingly useless board games, moving pebbles or cards according to prescribed rules, trying to beat their opponent in a game or just solving a puzzle. In this position paper we show how to use this human obsession in order to teach students formal methods, in particular, SAT solving and model checking.

Keywords: Model checking · Puzzles · Games · Formal methods · Education

1 Introduction

From an evolutionary point of view, being able to plan ahead and foresee a sequence of events is an essential survival skill. By considering different alternatives and their outcome, thinking beings are able to accomplish complex tasks and outperform competitors. Games and puzzles are mental exercises which train and improve the planning capacities of the human brain. Winning a game or solving a puzzle triggers a neural reward system, which reinforces the interest in the game or in similar puzzles. Therefore, humankind has always been interested in games. The earliest board games have been dated back to 2600 BC, with the oldest written rule set recorded in 177 BC [BrMus]. Being able to play board games like Chess and Go has been a criterion for machine intelligence at least since 1950, when Turing suggested a simple chess problem as a question in his imitation game [Tur 50]. Currently, the best computer programs for these games play better than any human opponent [SHS+ 17].

We want to demonstrate how the interest in games and puzzles can be used in a graduate course on automated verification, on advanced bachelor's or master's level (3^{rd} to 5^{th} year in computer science). We suggest to represent widely-known challenges as input for verification tools, and to have the students explore the capabilities and limitations of the tools via these examples. For specific model checkers and challenges, this suggestion has been made before (see, e.g., [SY01, vDR07, EJV05] for a certain coin game, an epistemic riddle, and an elevator problem). Here, we claim that this approach can be used systematically to

A. Cerone and M. Roggenbach (Eds.): FMFun 2019, CCIS 1301, pp. 143–158, 2021.
https://doi.org/10.1007/978-3-030-71374-4_7

introduce various modelling formalisms (propositional logic, labelled transition systems and concurrent game structures) as well as specification languages (temporal and strategic logics). The proposed method has been tried in the classroom by the author; however, this paper is not an "experience report". It rather gives concrete suggestions how to teach modelling and model checking on the graduate level in an interesting and intriguing way. Of course, to avoid the impression that formal methods have been invented to solve games, the playful examples presented here should be accompanied by appropriate examples from different industrial domains (which are, however, outside the scope of this paper).

The paper is structured as follows. Section 2 contains some general remarks of modelling puzzles and games with states and transitions. In Sect. 3, we model the well-known Sudoku puzzle with propositional logic and show how to use SAT-solving to find a solution. Section 4 introduces labelled transition systems for modelling and solving solitaire puzzles such as crossing a river and sliding blocks. In Sect. 5, we extend our teaching approach to concurrent game structures for modelling board games such as Tic-tac-toe. As a more scalable example, we introduce Tourality, which is a competitive rally game for two or more players. Finally, in Sect. 6 we conclude with ideas how to extend the approach to SMT modelling, discrete Markov chains, and epistemic structures.

Note: All code used in this paper can be downloaded from the following location.
https://osf.io/jxra3/ (Date: 28.11.2019)

2 Modelling Puzzles and Games

The notion of a state-transition system is fundamental for many formal methods. State-transition systems are the basis for finite automata, Kripke structures, labelled transition systems, and many other formalisms. Basically, a state-transition system is a graph consisting of nodes (i.e., states) and edges (i.e., transitions), where each edge connects two nodes. Alternatively, the set of transitions can be introduced as a relation between states. But what is a state? In the most general setting, a state of a system is defined to be a complete description of the system, consisting of the value of all parameters that determine the properties of the system. For a computer program, its state consists of the values of all variables or memory locations at some point in time. During a computation, a computer program passes through several states, by assigning values to variables. The verification task often amounts to showing that certain states are (un-)reachable. For example, we might want to show that a certain error state never occurs during execution, or that every computation leads to a state where the correct result has been calculated. For distributed systems, we might want to show that the interaction of the programs behaves correctly, i.e., that either no unwanted global state or some desired system goal state is reached, or that the system behaves correctly even in the presence of an intruder.

Many logical puzzles and games can be described as state-transitions systems. Puzzle or game elements are jigsaw pieces, letters, cards, pebbles, pegs, dice, etc. The state variables describe the current distribution of elements on the table or amongst the players. The game ends or the puzzle is solved, if a certain configuration is reached. In a multi-player game, usually the game ends if one of the player has reached a winning state.

Thus, both computations and games can be modelled in terms of states and transitions. This viewpoint is useful to convey the ideas underlying formal verification methods and model checking to graduate students. Subsequently, we show how to teach (some parts of) program verification and formal methods via puzzles and games.

3 Combinatorial Puzzles and SAT Solving

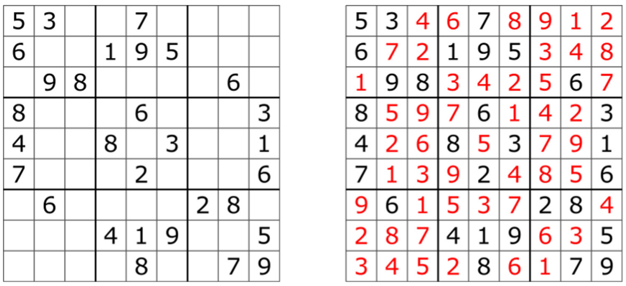
SAT solving is a technique which can be readily used to solve a certain class of combinatorial puzzles. We demonstrate this with the well-known example of Sudoku. This puzzle is given on a square board of 9×9 fields, which is divided into nine 3×3 sub-squares. Some of these fields are marked with a digit from 1 to 9, some are empty. The challenge is to fill the empty fields with digits (1 to 9) such that

1. each row of the board contains each digit exactly once,
2. each column of the board contains each digit exactly once, and
3. each sub-square contains each digit exactly once.

For any given marking of the field, there may exist no, exactly one, or more than one solution. Usually, the given marking is constructed such that there is exactly one solution. An example of a Sudoku puzzle and its solution is given in Fig. 1.

In a classroom, one could discuss strategies for solving such a puzzle after presenting the problem. There are various methods, ranging from the identification of unique candidate digits for a certain field, via placement of digits with backtracking, to various heuristic and brute force methods. Here, one could discuss the complexities of various methods.

We assume that the students have a basic understanding of propositional logic and the notion of satisfiability. Probably it is a good idea to remind them that SAT is the generic NP-complete problem. With this harness, it is possible to describe the problem of finding a solution for a given Sudoku puzzle as a



5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

5	3	4	6	7	8	9	1	2
6	7	2	1	9	5	3	4	8
1	9	8	3	4	2	5	6	7
8	5	9	7	6	1	4	2	3
4	2	6	8	5	3	7	9	1
7	1	3	9	2	4	8	5	6
9	6	1	5	3	7	2	8	4
2	8	7	4	1	9	6	3	5
3	4	5	2	8	6	1	7	9

Fig. 1. A Sudoku puzzle and its solution [Wiki1]

boolean satisfiability problem. The following encoding is inspired by the Sudoku web page and interactive Java applet by Ivor Spence [Spence].

We construct a boolean formula which is satisfiable if and only if the Sudoku puzzle has a solution. Assume that the rows and columns of the board are numbered from 1 to 9. In our formula, there are nine propositions per field: Proposition `p[`i`,`j`,`k`]` indicates that the field on row i and column j contains digit k. Thus, in the above example, `p[1,1,5]` is `TRUE`, and `p[1,1,1]` is `FALSE`. Since there are $9 \times 9 = 81$ fields, this gives 729 propositions in total.

Obviously, it is tedious to write boolean formulas with these many variables explicitly. Therefore, we use an array notation `p[i,j,k]` together with finite quantification `forall i=1..9` and `exists i=1..9` as abbreviations for the respective conjunction and disjunction, respectively. For example, `exists k=1..9 (p[1,1,k])` is short for `(p[1,1,1] | p[1,1,2] | p[1,1,3] | p[1,1,4] | p[1,1,5] | p[1,1,6] | p[1,1,7] | p[1,1,8] | p[1,1,9])`[1]. It is easy to write an appropriate pre-processor which expands formulas containing these finite quantifications into purely propositional formulas.

The following formula asserts that each field contains exactly one digit:

```
( forall i=1..9 forall j=1..9 exists k=1..9 (p[i,j,k]) &
  forall i=1..9 forall j=1..9 forall k=1..9 forall l=1..9, l≠k:
                                              !(p[i,j,k] & p[i,j,l]))
```

The following formula asserts that each digit is contained in each row at least once:

```
    ( forall i=1..9 forall k=1..9 exists j=1..9 (p[i,j,k]) )
```

Similarly, it can be formulated that each digit is contained in each column at least once:

```
    ( forall j=1..9 forall k=1..9 exists i=1..9 (p[i,j,k]) )
```

The formula for the sub-squares looks slightly more complex, but its boolean expansion is not:

```
( forall s=0..2 forall t=0..2 forall k=1..9
            exists i=1..3 exists j=1..3 (p[(3*s+i),(3*t+j),k]) )
```

Recall that the expansion of this formula is as follows:

```
  (p[1,1,1] | p[1,2,1] | p[1,3,1] | p[2,1,1] | ... | p[3,3,1]) &
  (p[1,1,2] | p[1,2,2] | p[1,3,2] | p[2,1,2] | ... | p[3,3,2]) &
    ... &
  (p[1,1,9] | p[1,2,9] | p[1,3,9] | p[2,1,9] | ... | p[3,3,9]) &
  (p[1,4,1] | p[1,5,1] | p[1,6,1] | p[2,4,1] | ... | p[3,6,1]) &
    ... &
  (p[7,7,9] | p[7,8,9] | p[7,9,9] | p[8,7,9] | ... | p[9,9,9])
```

Even though this may look like a long formula, it contains only $9^3 = 729$ literals. Finally, the value of pre-filled fields can be easily expressed by fixing the values of the respective propositions. For the above example, this gives

[1] In this paper, we use the symbols `|`, `&` and `!` for logical disjunction, conjunction, and negation.

```
p[1,1,5] & p[1,2,3] & p[1,5,7] & p[2,1,6] & ... & p[9,9,9].
```

Now, the Sudoku puzzle is solvable if and only if the conjunction of the above formulas is satisfiable, i.e., if there exists an assignment of truth values to propositional variables such that the whole formula becomes true. Modern SAT solver like miniSAT are able to find such an assignment, if it exists, within a few seconds. The above formula contains 729 propositions and approximately 9.000 literals. For such a problem size, satisfiability is decided within a few seconds. Each satisfying model gives a solution of the puzzle, since it determines the value of all propositions in the formula.

It is important to remark that the formula is just a representation of the problem; solving the problem is completely done within the SAT solver. We did not suggest or implement any heuristic or systematic way to solve the puzzle. Therefore, this approach is applicable for a large range of similar examples, where the task is to construct a certain state. However, if the problem size increases, it may be of advantage to explore different strategies implemented within the specific SAT solver which is used. For this, various extensions of the problem (an $n \times n$ board with increasing n, additional conditions on the solution, latin squares, etc.) can be analysed.

Thus, in a verification course, SAT procedures and heuristics for SAT solving can be discussed starting with this example. Furthermore, the behaviour of different SAT solvers on various examples can be measured and discussed.

4 Solitaire Puzzles and Model Checking

A somewhat more intricate sort of puzzles is asking not for a single goal state, but for a sequence of states leading to a certain outcome. In a solitaire game, a single player has to construct this sequence. The case when there is more than one player will be covered in the next section.

A classical example from the late 9^{th} century is the 'wolf, goat and cabbage' problem. It is originally described as follows (see [AoY]): "A certain man needed to take a wolf, a she-goat and a load of cabbage across a river. However, he could only find a boat which would carry two of these [at a time]. Thus, what rule did he employ so as to get all of them across unharmed?"

This puzzle is excellent to introduce the notion of a labelled transition system. If we assume that the man and the boat are always at the same side of the river, then there are four state variables indicating the position of the items: `p_man, p_wolf, p_goat, p_cabb`. Each of these state variables can be `FALSE` (indicating that the respective subject is still on the original side of the river), or `TRUE` (indication that the river has been crossed). Thus, there are $2^4 = 16$ states in the labelled transition system. These are depicted in Fig. 2, where each node label gives the values of `p_man, p_wolf, p_goat, p_cabb`. For example, node label `1100` means that man and wolf have crossed the river, whereas goat and cabbage are still on the original side. Labels `w, g, c,` and `n` indicate a crossing of the river by the ferryman with wolf, goat, cabbage, or no load, respectively.

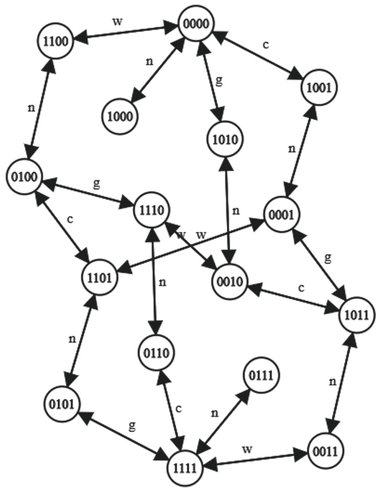


Fig. 2. Labelled transition system for the wolf-goat-cabbage problem

We want to model the problem in SMVL, the input language of the nuSMV model checker (see [CCGR 00]). As SMVL does not allow labels on transitions, we introduce an additional state variable `boat`, which can take any of the values {`w, g, c, n`}. With this, the transition relation is described as follows:

```
init(p_man)  := FALSE;
init(p_wolf):= FALSE;
init(p_goat):= FALSE;
init(p_cabb):= FALSE;

next(p_man):= ! p_man ;
next(p_wolf):= case
  p_wolf = p_man & boat = w : !
p_wolf;
  TRUE : p_wolf; esac;
next(p_goat):= case
  p_goat = p_man & boat = g : !
p_goat;
  TRUE : p_goat;
esac;
next(p_cabb):= case
  p_cabb = p_man & boat = c : !
p_cabb;
  TRUE : p_cabb;
esac;
```

Here are some explanations on this definition. Initially, all state variables are `FALSE`, and the variable `boat` has an arbitrary value. The variable `p_man` oscillates between `TRUE` and `FALSE`; we assume, that the ferryman crosses the river in every step. The next position of the wolf is determined as follows: If the wolf is on the same side as the man, and the `boat` variable indicates that the wolf is to be taken, then in the next state the wolf will be on the other side. In all other cases, the wolf has to remain where it is, i.e., the variable does not change. The same explanation holds for the lines concerning goat and cabbage.

In order to solve the puzzle automatically, we need to express the fact that certain of the states are "harmful". This can be conveniently done by a boolean formula on the state variables:

```
wolf_eats_goat := ((p_wolf = p_goat) & (p_man != p_wolf));
goat_eats_cabb := ((p_goat = p_cabb) & (p_man != p_goat));
harmful := (wolf_eats_goat | goat_eats_cabb);
```

This reflects the fact that the wolf can eat the goat, if they are both on the same side, and the ferryman is on the other side; and likewise for goat and cabbage. Furthermore, we can characterize the goal state(s):

```
all_crossed := (p_man & p_wolf & p_goat & p_cabb);
```

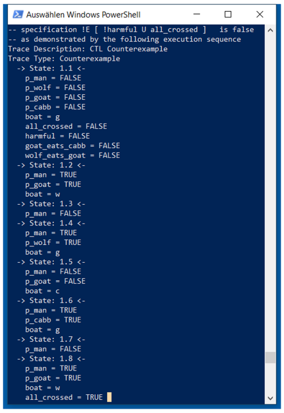
The task is completed if all three objects have crossed the river. Now we can ask nuSMV for the solution of the puzzle:

```
SPEC E [!harmful U all_crossed]
```

`E [`φ` U `ψ`]` is the CTL *existential-until* operator. Thus, the formula can be read as "there is a path from an initial state to a state where all items have crossed which passes no harmful state." The model checker immediately confirms that this is the case. However, this is not very helpful, as we would like to know an example for such a path. Therefore, we ask for the negation of the formula:

```
SPEC !E [!harmful U all_crossed]
```

As expected, the model checker confirms that this formula is false and delivers as proof a counterexample, i.e., a sequence of eight steps falsifying the formula and thus demonstrating our original aim (Fig. 3).



```
-- specification !E [ !harmful U all_crossed ]   is false
-- as demonstrated by the following execution sequence
Trace Description: CTL Counterexample
Trace Type: Counterexample
  -> State: 1.1 <-
    p_man = FALSE
    p_wolf = FALSE
    p_goat = FALSE
    p_cabb = FALSE
    boat = g
    all_crossed = FALSE
    harmful = FALSE
    goat_eats_cabb = FALSE
    wolf_eats_goat = FALSE
  -> State: 1.2 <-
    p_man = TRUE
    p_goat = TRUE
    boat = w
  -> State: 1.3 <-
    p_man = FALSE
  -> State: 1.4 <-
    p_man = TRUE
    p_wolf = TRUE
    boat = g
  -> State: 1.5 <-
    p_man = FALSE
    p_goat = FALSE
    boat = c
  -> State: 1.6 <-
    p_man = TRUE
    p_cabb = TRUE
    boat = g
  -> State: 1.7 <-
    p_man = FALSE
  -> State: 1.8 <-
    p_man = TRUE
    p_goat = TRUE
    boat = w
    all_crossed = TRUE
```

Fig. 3. nuSMV solution for the wolf-goat-cabbage problem

The above example has the disadvantage that it is not easy to extend the problem size. Grid-based games and puzzles like Sudoku are better suited to explore the abilities and limits of verification tools. A well-known example of this kind, allegedly invented by Sam Loyd in the 1870s, is the Fifteen-puzzle, It consists of a $h \times v$ grid in which there are $(h \cdot v) - 1$ numbered tiles and one blank space. Originally, $h = v = 4$, hence the name of the puzzle. A move consists in moving any tile into the position of the blank. The goal is to achieve a certain predetermined order on the tiles (usually ascending).

In contrast to Sudoku, this puzzle can not directly be coded as a boolean satisfiability problem. The set of states is given by the distribution of the tiles on the grid. Similar as in the previous example, solving the puzzle must be done by constructing a sequence of states, where each next state is reachable from the current state via a legal move (Fig. 4).

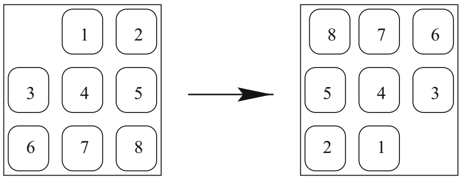


Fig. 4. The 3 × 3–Fifteen-puzzle: Start- and end-state [CS01]

This puzzle can be described by a state-transition system as follows. For each tile there is a program variable which notes its horizontal and vertical position. Furthermore, there is a program variable `move` indicating whether the next move will be a shift up, down, left or right of the blank space. If the move would bring it out of the borders, nothing is changed; otherwise, its position is swapped with the respective adjacent tile.

The SMV code corresponding to this description[2] is shown below.

For $h = 3$ and $v = 3$, the internal representation of the transition relation uses 440.419 nodes. There are $4 \cdot (h \cdot v)! = 1.4 \cdot 10^6$ states, of which 50% are reachable from an initial state. (Note that there are four initial states, since we did not fix the initial value for variable `move`.) As in the previous example, the specification claims that a certain final state is *not* reachable; the model checker contradicts this claim by showing a sequence of moves (ddrruullddrruullddrruullddrr) which gives a solution to the puzzle. The solution is found within a few seconds.

[2] As above, in the actual SMV code, variable array bounds or indices, e.g., vpos[i], are not allowed and have to be replaced by the respective constant values vpos[1],vpos[2],....

```
MODULE main
 DEFINE h := 3; v := 3;
 VAR move: {u,d,l,r};
     hpos: array 0..(h*v-1) of 1..h;
     vpos: array 0..(h*v-1) of 1..v;
 ASSIGN
 next(vpos[0]) := case
   (move=l) & !(vpos[0]=1) : vpos[0] - 1;
   (move=r) & !(vpos[0]=v) : vpos[0] + 1;
   1: vpos[0]; esac;
 next(hpos[0]) := case
   (move=u) & !(hpos[0]=1) : hpos[0] - 1;
   (move=d) & !(hpos[0]=h) : hpos[0] + 1;
   TRUE: hpos[0]; esac;
 forall i=0..8:
 next(vpos[i] := case
   (move=l) & !(vpos[0]=1) & hpos[i]=hpos[0] & vpos[i]=vpos[0]-1 |
   (move=r) & !(vpos[0]=v) & hpos[i]=hpos[0] &
vpos[i]=vpos[0]+1 : vpos[0];
   TRUE: vpos[i]; esac;
 next(hpos[i]) := case
   (move=u) & !(hpos[0]=1) & vpos[i]=vpos[0] & hpos[i]=hpos[0]-1 |
   (move=d) & !(hpos[0]=h) & vpos[i]=vpos[0] &
hpos[i]=hpos[0]+1 : hpos[0];
   TRUE: hpos[i]; esac;
 init(vpos[i]) := i div v; init(hpos[i]) := i mod v;
 DEFINE goal := (vpos[i] = 3 - (i div v) & hpos[i] = 3 - (i mod v));
 SPEC !EF goal
```

For $h = 4$, $v = 3$, there are approximately 10^9 reachable states. Although the symbolic model checker detects rather quickly that some solution must exist, for the construction of a concrete solution sequence the state space has to be partitioned into strongly connected components. This requires significant CPU time and memory. Thus, this example is well-suited to discuss complexity and try the many options which nuSMV offers. In particular, a teacher can explain the idea of bounded model checking, thereby relating solitaire games to SAT solving.

There are several other, comparable puzzles of this type which can be treated in classroom exercises. An example similar to river crossing is the 'water pouring puzzle' from the 17^{th} century, where the objective is to reach a certain distribution of water in three jugs [Bac 1612]. Various sliding block puzzles like Klotski or Sokoban can be treated similar to the 'Fifteen-puzzle'. Another example is peg solitaire, where the objective is to empty a board of pegs [JMMT 06]. It would also be interesting to model 3D-objects like Rubik's Magic, Clock or Cube.

5 Board Games and Strategic Logics

In the previous section, we considered puzzles and solitaire games, which are well-suited of demonstrating the computing paradigm of *reactive systems*. A single player reacts to a challenge posed by the environment. The situation changes if we consider *interactive systems*, where two or more players interact, and the actions of the players are mutually dependent. Interactive systems have been researched under different names: Distributed computing systems, Multi-agent systems, Cyber-physical systems, Multi-player games, and others.

For modelling interactive systems, *concurrent game structures* (CGS) are being used. Basically, a concurrent game structure for n players is an n-tuple of labelled transition systems. Formally, a CGS is a structure $(Agt, S, Act, \pi, \delta, s_0)$, where $Agt = \{a_1, ..., a_n\}$ is a finite set of *agents*, $S = S_1 \times \cdots \times S_n$ is a finite set of *global states* (and each global state s is a tuple $\langle s_1, ..., s_n \rangle$ of local states), Act is a finite set of *actions*, $\pi : Agt \times S \mapsto 2^{Act}$ is the *protocol function* indicating which actions are available to an agent is a certain state, $\delta : S \times Act^n \times S$ is the *evolution function* (i.e., transition relation) indicating how the global state changes if the agents each perform a certain action, and $s_0 \subseteq S$ are the initial states of the structure. The evolution function gives a successor state only for those combination of actions which are available to the agents in a state (i.e., $(s, \langle e_1, ..., e_n \rangle, s') \in \delta$ implies that $e_i \in \pi(a_i, s)$ for all $i \leq n$, and if $e_i \in \pi(a_i, s)$ for all $i \leq n$, then there is exactly one s' such that $(s, \langle e_1, ..., e_n \rangle, s') \in \delta$).

A *strategy* is a plan which tells an agent which of the available actions to choose in a certain situation. Often, the strategy is used to reach a designated goal state, or to stay within a set of safe states. Formally, a strategy σ_i for agent $a_i \in Agt$ is a function $\sigma_i : S \mapsto Act$, such that $\sigma_i(s) \in \pi(sa_i, s)$. Given strategies $\sigma_1, ..., \sigma_n$ for all players of a CGS, there is exactly one execution sequence following all these strategies.

Strategic logics like the alternating temporal logic ATL have been proposed to reason about interactive systems modelled by concurrent game structures. ATL is an extension of CTL which allows quantification on strategies. Formula $\langle A \rangle \varphi$ expresses that there exists a strategy for the players $a \in A$ such that they can force the temporal logic formula φ to hold, no matter what the other players do, This is convenient to express the fact that a player has a winning strategy in a game.

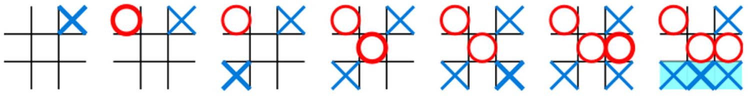


Fig. 5. A Tic-tac-toe game [Wiki2]

As a simple example, we use the classic game of Tic-tac-toe, also called 'Noughts and crosses' or 'Three in a row'. It is played on a 3×3 grid on paper, where two players take turn marking the fields with X and O. The player who first has three of her/his symbols horizontally, vertically or diagonally in a row wins. For an example game, consider Fig. 5.

In contrast to solitaire games, in two-player games the actions of each player depend on the actions of the opponent. Thus, to construct a winning strategy one has to consider all potential strategies of the opponent. This makes strategic model checking often much more complex than reachability analysis.

Such games can be analyzed via strategic model checkers such as MCMAS (see [LQR 09]). Subsequently, we formalize the rules of Tic-tac-toe in (a language similar to)[3] ISPL, which is the input language for MCMAS.

```
Agent Environment
  Obsvars:
    turn: {nought, cross}; -- variable indicating who's turn it is
    b[1..3][1..3]: {x, o, b}; -- the board markings; b means blank
  end Obsvars
  Evolution:
    -- turn switches between every two moves
    turn=nought if turn=cross; turn=cross if turn=nought;
    -- board is marked according to the move
    forall i=1..3 forall j=1..3
      b[i][j] = o if turn = nought & Nought.Action = a_ij;
    forall i=1..3 forall j=1..3
      b[i][j] = x if turn = cross  & Cross.Action  = a_ij;
  end Evolution
end Agent

Agents Nought, Cross
  Actions = {a_ij | i=1..3, j = 1..3};
  Protocol: -- Action a_ij is available if Board b[i][j] is blank:
    forall i=1..3 forall j=1..3 ( b[i][j]=b : {a_ij} );
  end Protocol
end Agent

Evaluation
  noughtwins if
    ( exists i=1..3 b[i][1]=o & b[i][2]=o & b[i][3]=o ) |
    ( exists i=1..3 b[1][i]=o & b[2][i]=o & b[3][i]=o ) |
    b[1][1]=o & b[2][2]=o & b[3][3]=o |
    b[3][1]=o & b[2][2]=o & b[1][3]=o;

  crosswins if
    -- similar, =x instead of =o
end Evaluation
```

[3] ISPL does not admit arrays, these must be expanded by a suitable pre-processor. Furthermore, ISPL has some syntactic peculiarities which do not contribute to the goals of this article and, thus, are left out. The full code of all examples in this article can be obtained from the author.

```
InitStates
  ( forall i=1..m forall j=1..n b[i][j]=b )  & (turn = cross);
end InitStates

Formulae
  <Cross> F crosswins;
  <Nought> F noughtwins;
end Formulae
```

Perhaps surprisingly, the model checker reports that the first formula is `TRUE` and the second one `FALSE`. This is because agent `Cross` indeed has a strategy to reach three `x` in a row if it does not care whether agent `Nought` reaches three `o` in a row first. Thus we have to ask whether one of the following formulas is true:

```
    <cross> F (crosswins & ! noughtwins);
    <nought> F (noughtwins & ! crosswins);
```

This is indeed not the case, as the model checker confirms in 0.372 s.

Tic-tac-toe can be easily generalized to the (m, n, k)-*game*, where two players compete to place k symbols in a row on an $m \times n$ grid. This generalization is mathematically interesting, as it quickly leads to open questions: For example, for $k \geq 9$ it can be shown that even on an infinite board there is no winning strategy for the first player. However, for $k = 6$ or $k = 7$ it is not known whether there are m and n such the first player has a winning strategy in the (m, n, k)-game. Consequently, the complexity of model checking quickly grows with increasing m, n and k: Already for the $(4, 4, 3)$-game there are more than 10^7 reachable states, and it takes more than 4 min to find a winning strategy for agent `Cross`.

Thus, for use in a model checking course, we prefer other examples. Subsequently, we describe a location-based game which has been called Tourality[4]. It is played by two players and resembles the classical computer game PacMan, but without the real-time aspect.

The game is played on an 8×8 droughts/checkers board, with some set-up of black and white tiles. Black tiles are obstacles, which remain in position throughout the game. White tiles are rewards, which are collected by the players. In one corner of the board there is a red peg, and in the opposite corner a blue peg. One player sets up the board, the other one chooses whether to play red or blue. Players take turn moving their peg to an (horizontally or vertically) adjacent field on the board. It is not allowed to move onto a field occupied by an obstacle or the other player's peg. If a player moves the peg onto a field where there is a reward, it is collected. Red begins. The game ends when all rewards are collected, and the player who has collected most rewards wins.

[4] Actually, Tourality is a GPS-based treasure hunt where real people are trying to outperform each other in reaching certain locations. We use a card-board abstraction of this game which was introduced in a German national competition for computer science education.

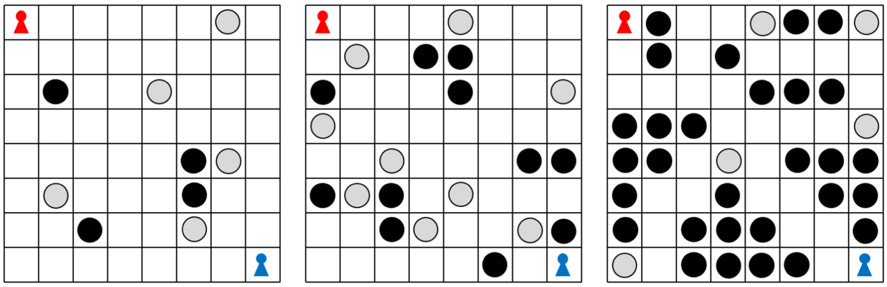


Fig. 6. Three different Tourality set-ups (Color figure online)

Tourality can be easily modelled by a concurrent game structure. For space reasons we give only the main constituents, again in “not quite” ISPL.

```
Agent Environment
  Obsvars:
    turn : {red, blu}; xred, yred, xblu, yblu : 1..8;
    reward[1..5] : {avail, taken}; -- for example, 5 rewards
    points_red, points_blu: 0..5;
    constant b[1..8][1..8] : {empty, block}; -- the board
    constant xreward[1..5], yreward[5]: [1..5]; -- positions of the rewards
  end Obsvars
  Evolution:
    -- turn switches between every two moves
    turn=red if turn=blu; turn=blu if turn=red;
    -- positions are updated according to the move
    yred=yred-1 if turn = red & Red.Action=up;
    yred=yred+1 if turn = red & Red.Action=dn;
    -- and similar for other actions and player Blu;
    -- board and points are updated according to the moves of the players:
    for 1=1..5: reward[i] = taken if reward[i] = avail &
      (turn = blu & xred = xreward[i] & yred = yreward[i] |
       turn = red & xblu = xreward[i] & yblu = yreward[i]);
    for 1=1..5: points_red=points_red+1 if reward[i] = avail &
      turn = blu & xred = xreward[i] & yred = yreward[i];
    for 1=1..5: points_blu=points_blu+1 if reward[i] = avail &
      turn = red & xblu = xreward[i] & yblu = yreward[i];
  end Evolution
end Agent

Agent Red
  Actions = { up, dn, lt, rt };
  Protocol:
    -- if it is red’s turn and at position i,j and target field is not blocked
    -- and target field is not occupied, then the movement action is available
    for some x=1..8: for some y=1..7:
      xred=x & yred=y & b[x,y+1]=empty & !(xblu=x & yblu=y) : { dn };
    for some x=1..8: for some y=2..8:
      xred=x & yred=y & b[x,y-1]=empty & !(xblu=x & yblu=y) : { up };
    for some x=1..7: for some y=1..8:
      xred=x & yred=y & b[x+1,y]=empty & !(xblu=x & yblu=y) : { rt };
    for some x=2..8: for some y=1..8:
      xred=x & yred=y & b[x-1,y]=empty & !(xblu=x & yblu=y) : { rt };
   end Protocol
end Agent

Agent Blu
  -- similar
end Agent

InitStates
  b = [[empty, empty, block, empty, block, empty, empty, empty], ...] &
  xreward=[3,2,5,4,3], yreward=[2,5,7,1,4] &
  xred = 1 & yred = 1 & xblu = 8 & yblu = 8 & turn = red &
  for i=1..5: reward[i] = avail & points_red = 0 & points_blu = 0;
end InitStates

Formulae
  <Red> F (points_red >= 3); -- has Red a winning strategy?
end Formulae
```

MCMAS can check the example set-ups in Fig. 6 within a few seconds (for the first and third board) and a few minutes (for the second one). As can be seen by these examples, already the normal rules of the game allow for many variants. One can arrange the obstacles such that they form a maze, or such that they block certain parts of the board. This can drastically reduce the number of reachable states. Alternatively, one can have few or many rewards on the board, thereby decreasing or increasing the number of possible strategies. Already a few experiments show this effect very clearly: whereas the first example setup in Fig. 6 has 7×10^5 reachable states and uses 3.5 s, the third one has only 3×10^5 states, but needs 12 s. Adding more rewards yields an exponential blowup, e.g., the middle setup in Fig. 6 with 9 rewards has 2×10^7 reachable states and uses 6 m, with 10 rewards the size of the reachable state space is 10^8 and uses 15 m. and with 11 rewards there are 10^9 reachable states calculated in 150 m.

However, in a computer based version of the game, it is also possible to vary other parameters of the game. For example, the vicinity relation can be changed to allow also horizontal moves. Alternatively, one can allow only moves according to the knight's movement in chess. Furthermore, one can replace the turn-based move by concurrent moves of the player. It is also possible to introduce more than two players, and to allow coalitions and competition between them. It is fun to play around with these alternatives and see how the game changes and how the model checker behaves. This can be useful to explain the potential of strategic model checking.

Other locality-based games are likewise well-suited to motivate this technique. For example, formalising the rules of Nine Men's Morris is an interesting exercise. A more challenging task would be to ask students to model and solve end games in chess.

6 Conclusion

In this paper, we have shown how to use puzzles and games to demonstrate the potential and limitations of model checking in a graduate-level course. We used SAT solving for Sudoku, model checking for the 'wolf, goat and cabbage' problem and the Fifteen-puzzle, and strategic model checking for Tic-tac-toe and Tourality.

There are many extensions to the ideas presented here. SMT solving (satisfiability modulo theories) can be used for problems involving integers. An example are alphametic puzzles, where an arithmetic problem is given with letters in place of the digits, and the challenge is to deduce which digit corresponds to each letter.

Discrete Markov chains and probabilistic model checking (e.g., with the PRISM model checker) can be explained with the help of dice games.

An extension to model checking of multi-player games with perfect information is the case of imperfect information. To model such situations, epistemic concurrent game structures have been proposed. MCMAS is able to encode individual knowledge of the agents in the epistemic accessibility relation between states. Furthermore, formulas containing epistemic modalities **E**, **C**, and **D** can be checked. Epistemic logics are often introduced via artificial set-ups such as dining cryptographers ("Who knows what about the payment?"), muddy children ("Who has a spot on the forehead?"), detective puzzles ("Does the murderer know that the detective knows that ..."), or similar. We prefer to use simplified versions of common card games like Bridge, Poker, or Blackjack, where each player has some private knowledge, and additionally there is public knowledge about the distribution of cards. However, a strong factor in these games is the probabilistic aspect brought in by the random distribution of cards. Unfortunately, to our knowledge there are no logics and tools combining probabilistic analysis with strategic and epistemic reasoning. This could turn out to be a fruitful research topic.

Thus, our considerations are not only useful for education. Since games and puzzles often are just abstract representations of real-life challenges, they may give directions on how to further evolve the formal methods tools. As another example, we would like to be able to use strategic model checking in an on-line fashion, to enable the synthesis of controllers for embedded systems which have to react in real time.

Acknowledgements. The author wishes to thank Damian Kurpiewski for helpful discussions on MCMAS, and Stephan Merz for discussions on the game-based approach to model checking education. This research was supported by the German BMBF project "CrESt" on collaborative embedded systems, FKZ 01|S16043G/E.

References

[BrMus] British Museum: The Royal Game of Ur ~2600/~2400. https://artsandculture.google.com/asset/the-royal-game-of-ur/MwE2MMZNSKiTwQ. Clay cuneiform tablet. https://www.britishmuseum.org/research/collection_online/collection_object_details.aspx?objectId=796973&partId=1. Accessed 13 Nov 2019

[Tur 50] Turing, A.: Computing machinery and intelligence. Mind **59**, 433–460 (1950)

[SHS+ 17] Silver, D., et al.: Mastering chess and shogi by self-play with a general reinforcement learning algorithm. arXiv:1712.01815v1 [cs.AI] (2017). https://arxiv.org/pdf/1712.01815.pdf. Accessed 13 Nov 2019

[SY01] Shilov, N.V., Yi, K.: Puzzles for learning model checking, model checking for programming puzzles, puzzles for testing model checkers. Electron. Notes Theoret. Comput. Sci. **43**, 34–49 (2001). http://www.elsevier.nl/locate/entcs/volume43.html. Accessed 13 Nov 2019

[vDR07] van Ditmarsch, H., Ruan, J.: Model checking logic puzzles (2007). https://www.researchgate.net/publication/29646125_Model_Checking_Logic_Puzzles. Accessed 13 Nov 2019

[EJV05] Eskildsen, J., Jensen, L.H., Vester, B.M.: Symbolic model checking in puzzle games - automated reachability analysis. Aalborg Universitet, DAT4, May 2005. https://pdfs.semanticscholar.org/1783/d58420fd6a05b90dc29bbba5fc2cd9e3a113.pdf. Accessed 13 Nov 2019

[Wiki1] Wikipedia: Sudoku. https://en.wikipedia.org/wiki/Sudoku. Puzzle drawn by Tim Stellmach, CC0. https://commons.wikimedia.org/w/index.php?curid=57831926. Solution drawn by en:User:Cburnett, CC BY-SA 3. https://commons.wikimedia.org/w/index.php?curid=57831971. Accessed 13 Nov 2019

[Spence] Spence, I.: The SuDoku puzzle as a satisfiability problem. http://www.cs.qub.ac.uk/~I.Spence/SuDoku/SuDoku.html. Accessed 13 Nov 2019

[AoY] Alcuin of York: Propositiones ad Acuendos Juvenes, Problem XVIII. Propositio de homine et capra et lupo. 9th century a.D., Translation by Burkholder, P. http://www.math.muni.cz/~sisma/alcuin/anglicky1.pdf. Accessed 13 Nov 2019

[CCGR 00] Cimatti, A., Clarke, E., Giunchiglia, F., Roveri, M.: NuSMV: a new symbolic model verifier. Int. J. STTT **2**(4), 410–425 (2000)

[Bac 1612] Bachet, C.-G.: Problemes plaisans et delectables, Published by P. Rigaud, Lyon (1612). Text available at https://www.loc.gov/resource/rbc0001.2009gen48833/. See also https://en.wikipedia.org/wiki/Water_pouring_puzzle. Accessed 13 Nov 2019

[JMMT 06] Jefferson, C., Miguel, A., Miguel, I., Tarim, S.A.: Modelling and solving English Peg Solitaire. Comput. Oper. Res. **33**(10), 2935–2959 (2006)

[CS01] Clarke, E.M., Schlingloff, H.: Model checking. In: Handbook of Automated Reasoning. Elsevier Science Publishers (2001)

[Wiki2] Wikipedia: Tic-tac-toe. https://en.wikipedia.org/wiki/Tic-tac-toe. Drawing by User: Stannered - en:Image:Tic-tac-toe-game-1.png, CC BY-SA 3.0. https://commons.wikimedia.org/w/index.php?curid=1866155. See also https://en.wikipedia.org/wiki/M,n,k-game. Accessed 13 Nov 2019

[LQR 09] Lomuscio, A., Qu, H., Raimondi, F.: MCMAS: a model checker for the verification of multi-agent systems. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 682–688. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02658-4_55

Cybersecurity Education and Formal Methods

James H. Davenport[1](✉) and Tom Crick[2]

[1] Mathematical Foundations Group, Department of Computer Science, University of Bath, Bath, UK
masjhd@bath.ac.uk

[2] Department of Computer Science, Swansea University, Swansea, UK
thomas.crick@swansea.ac.uk

Abstract. Formal methods have been largely thought of in the context of safety-critical systems, where they have achieved major acceptance. Tens of millions of people trust their lives every day to such systems, based on formal proofs rather than "we haven't found a bug" (yet!); but why is "we haven't found a bug" an acceptable basis for systems trusted with hundreds of millions of people's personal data?

This paper looks at some of these issues in cybersecurity, and the extent to which formal methods, ranging from "fully verified" to better tool support, could help. More importantly, recent policy reports and curricula initiatives appear to recommended formal methods in the limited context of "safety critical applications"; we suggest this is too limited in scope and ambition. Not only are formal methods needed in cybersecurity, the repeated and very public weaknesses of the cybersecurity industry provide a powerful motivation for formal methods.

Keywords: Formal methods · Cybersecurity · Curricula

1 Introduction

Formal methods, when they have been thought of at all, have been largely thought of in the context of safety-critical systems, where they have achieved major acceptance in what is, alas, an unsung area of software development. Tens of millions of people trust their lives every day to such systems, but nearly all are unaware of these systems, and the extent to which they are enormous successes. Even people "who ought to know better" don't. One of the authors quoted the Paris Métro Ligne 14 performance figures (software shipped in 1999 and no bugs reported [1]) to a major figure in the commercial software industry, to be told that he was lying, as this was utterly impossible.

Formal methods *ought* to be much more widely used in the cybersecurity industry. This is much more visible (because it has many conspicuous failures) than the largely invisible safety-critical industry. However, formal methods are not currently widely adopted here, and hence there is tremendous scope for growth and adoption of formal methods. In addition, as the cybersecurity industry and its failures are much more visible, emphasising the relevance of formal methods to the cybersecurity industry should encourage more interest at universities.

2 Cybersecurity

Cybersecurity[1] failures abound, and the number of people that can be affected by even a single failure is amazing—148 million for Equifax [2] and probably more for the Starwood[2] breach: a number [3] "downgrades" to 383 million. The financial costs can be substantial: bankruptcy in the case of American Medical Collection Agency [4] and a provisional £183M fine for British Airways [5]. These problems have attracted attention at the highest scientific levels [6].

There are many reasons for cybersecurity failures, and even a given failure may have multiple causes. For example, the U.S. Government investigation [7] into Equifax states "Equifax's investigation of the breach identified four major factors[3] including identification, detection, segmenting of access to databases, and data governance that allowed the attacker ...". However, none of these would have been triggered had it not been for the original bug in the Apache code [9], which was of the well-known (Number 1 Application Security Risk in [10]) family of "Injection" (or "Remote Code Execution") attacks, and which would probably have been detected by an automatic taint analysis tool such as [11].

Though attributing causes at scale is difficult, a well-known textbook [12] claims that about 50% of security breaches are caused by coding errors. Hence it behoves security practitioners to look seriously at coding errors, while recognising that this is only one facet of the problem. This is taken up by the Payments Card Industry in [13], essentially the only world-wide mandatory security standard, in two sub-requirements of "Requirement 6: Develop and maintain secure systems and applications".

6.5 Address common coding vulnerabilities in software-development processes as follows:
- Train developers at least annually in up-to-date secure coding techniques, including how to avoid common coding vulnerabilities;
- Develop applications based on secure coding guidelines.

[1] The precise definition of cybersecurity is debatable: we can take is as failures of security, generally defined as "preserving the CIA—Confidentiality, Integrity and Availability" of digital information, where computer system played a critical part in the failure.

[2] Generally called "Marriott", but in fact due to the Starwood chain before Marriott took it over.

[3] In military parlance, Equifax is being found not to have "defence in depth". Defence in depth is certainly valuable: [8] described how Google was saved from the consequences of an 'awesome' attack on gmail by defence in depth. But the front line is still the first defence: in this case correct code.

6.6 For public-facing web applications, address new threats and vulnerabilities on an ongoing basis and ensure these applications are protected against known attacks by either of the following methods:
  - Reviewing public-facing web applications via manual or automated application vulnerability security assessment tools or methods, at least annually and after any changes;
  - Installing an automated technical solution that detects and prevents web-based attacks (for example, a web-application firewall [WAF]) in front of public-facing web applications, to continually check all traffic.
  - * The authors note that many sites go for the second option, as it's easy to "tick the box" Indeed [14] finds that 34% of customers regard compliance as the main function of their firewall, rather than security. Apart from the fact that firewalls can be misconfigured, as in the Capital One attack [15], they can also be placed in "detect-only" mode and ignored[4].

- A further challenge to the PCI DSS model is provided by the modern "Magecart" attacks, such as that against British Airways [17]: here the JavaScript that is downloaded to the customer's browser is corrupted, and this leaks the data directly, without going near the WAF. This raises the challenge of JavaScript verification Sect. 5.4.

It is noteworthy that, despite apparently insisting on secure coding in 6.5, they require the additional defences in 6.6, realising that *errare humanum est*, and the 6.5-developed code may not actually be secure. Is it possible (the authors think so, but the experiment has yet to be performed) that adding formal methods to 6.5 would render 6.6 redundant, or at least mean that 6.6 should be restricted to finding design errors, rather than debugging 6.5 failures? Full formal verification of a complete system should certainly suffice.

> Complete formal verification is the only known way to guarantee that a system is free of programming errors. [18, describing seL4: a verified operating system]

Such a verified operating system has been used in medical devices, but probably not sufficiently widely, as 500,000 already-fitted pacemakers have had to be upgraded through security weaknesses [19], and insulin pumps are also vulnerable [20], as many others [21,22]. See [23] for a recent update on seL4. However, most of us do not have the opportunity to start from scratch, and have to live on top of imperfect, unverified systems, interoperating with other systems via large, generally unverified, or at least under-verified, protocols, such as TLS [24].

3 Agile Versus Secure

"Agile Development" [25] is a major theme in software development. Mark Zuckerberg can be said to have taken this theme to the extreme in 2009.

[4] [16]: "Midsize and small companies frequently install WAFs just to satisfy a compliance requirement. They don't really care about practical security, and obviously won't care about maintaining their WAF." This is backed up by [14], whose survey states "43% run their WAF in alert-only mode!".

> "Move fast and break things" is Mark's prime directive to his developers and team. "Unless you are breaking stuff," he says, "you are not moving fast enough." [26]

In both safety-critical and security-conscious programming, "breaking things" comes with a very high price. Aeroplanes can't be uncrashed, and data can't be unleaked.

The problems with using "Agile" methods in security are well-documented, at practitioner level, e.g. a recent "Security + Agile = FAIL" presentation [27], in many theoretical analyses as well as the interview-based research in [28] for small teams and [29] for large multi-team projects. Both mention team expertise in security as a significant problem.

From [28] The overall security in a project depends on the security expertise of the individuals, either on the customer or developer side. This corresponds to the agile value of "individuals and interaction over processes and tools" [25, Value 1].

From [29] The interviewees generally agree that more could be done to provide security education and training to employees. Without prompting, several interviewees mentioned training as an important factor for increasing security awareness and expertise.

It is very hard to take security seriously in this setting.

From [28] security "is only of interest [to the customer] when money-aspects are concerned".

From [29] One Test Manager articulated his team view that "security is not currently seen as part of working software, it only costs extra time and it doesn't provide functionality". With less focus on providing extensive (security) documentation typical for agile, ineffective knowledge sharing between security officers and agile team members is especially problematic.

From [30] (A more general survey, but many papers surveyed were "Agile") "Security is often referred to as a NFR [non-functional requirement] in that it is expected to be included as part of high quality code development, but is rarely listed as an explicit requirement. As a result, developers prioritise security below more-visible functional requirements or even easy-to-measure activities such as closing bug tracking tickets."

It would be tempting to conclude that "Agile" and "Secure" are, or at least are close to being, mutually contradictory. But there has been some analysis of the same apparent contradiction in the safety-critical industry [31]. Other than "Embedded Systems"[5] [31, Sect. 3.6], this analysis of the problems is fairly close to the practitioner view in [27], and we could reasonably ask what lessons could be carried across.

[5] Actually, Embedded Systems are a comparatively neglected, but important, cyber-security area. See, for example, [32] for a description of a pervasive design fault in the "home security" market.

4 The Need for Tools

There are two key points.

From [31, Sect. 4.1] Strong static verification tools tend to complement (not replace) human-driven review[6]. The tools are very good at some problems (e.g. global data flow analysis, theorem proving) where humans are hopeless, and vice versa. If we do the static verification first, then we can adjust manual review processes and check-lists to take advantage of this.

From [31, Sect. 6] The sixty-four-million-dollar-question, it seems, is how much "up-front" work is "just right" for a particular project. We doubt there's a one-size-fits-all approach, but surely the answer should be informed by disciplined requirements engineering of non-functional properties (e.g. safety, security and others) that can inform the design of a suitable architecture and its accompanying satisfaction argument.

Facebook grew, security (and "product quality" in general: it is not clear whether security was the main driver here) became more important, and by 2014 Zuckerberg had changed his views.

> "Move fast with stable infrastructure." It "may not be quite as catchy as 'move fast and break things," Zuckerberg said with a smirk. "But it's how we operate now." [34]

One might think his views were converging with the views of [31]. However, the Heartbleed story [35] should remind us that the fact that a modification "has no new security considerations" *as designed* [36] doesn't mean that an implementation of that idea has no new security considerations. Hence the call in [31, Sect. 4.1] for strong static verification tools. Such tools are generally seen as expensive and slowing down the development process, but [37] shows that they need not be. In particular, they show that, for a real application (890,000 physical lines of Ada code), the cost of incremental verification can be reduced from "nightly" to "coffee", and hence can reasonably form part of a continuous integration toolchain, as is done at the company studied in [37]. Readers might comment that their own applications are not in Ada, but [38, Sect. 5.6] discusses mixed-language programming, especially with C. A similar point is made in [39], describing the Infer tool running on Java/Objective C/C++, where moving from overnight reporting to near real-time reporting moved the fix rate from 0% to 70%.

That these techniques are reaching the mainstream of cybersecurity can be seen from Amazon Web Services adoption of them [40], Google [41], Facebook [39], and the recent DefectDojo release by OWASP [42].

5 The Scope of Tools and Formal Methods

There is a substantial range of tools, and degrees of formality, and [31, Sect. 6] is probably correct in saying "We doubt there's a one-size-fits-all approach". At

[6] A point made in the context of XP and Agile in 2004 [33].

one extreme, there are the humble, but still surprisingly effective, `lint` and its equivalents, looking, essentially, for dangerous or dubious, though legal, syntax.

5.1 Ada and SPARK

At the other extreme, there are languages, such as the SPARK Ada subset [38] designed with verification in mind and heavily employed in the safety-critical sector such as railways and air traffic control, which can also be deployed for demanding secure applications, such as an RFC4108-compliant secure download system for embedded systems [43].

5.2 C/C++

There is, however, a large middle ground between these two extremes. Even if the application is required to be in C or C++, there is a lot to be said for sticking to a safer (even if not provably safe) subset of the language *and associated libraries*, such as eschewing `strcpy` in favour of `strncpy`. This can often be enforced by static verification tools. We note that Google's "Zero Day" project reports [44] that 68% of all such zero-day exploits (i.e. exploits discovered in the wild first) were caused by memory corruption errors, and Microsoft report a very similar story [45].

There is a good survey of such subsets and standards in [46, Appendix F]. As that notes, the ISO standard for secure C coding [47] has the unusual (for this middle ground) but important concept of "taint analysis" (as in [11]): input data should be considered "tainted" until it has been sanitised. This is particularly important for network-oriented applications, where it is natural for the programmer to believe that the other party is behaving correctly (as in **Heartbleed** [35]).

After this paper was presented, [48] appeared. That paper's authors had formally proved properties of non-trivial parts of Amazon's core C library. "We proved that key components of AWS C Common are memory safe, i.e. do not suffer from issues such as buffer overflow, use after free, or invalid pointer dereferences. Memory safety errors are routinely listed among the most critical security concerns by industry groups monitoring CVEs".

5.3 Java

Closer to the SPARK Ada end of the spectrum we find Safety-Critical Java [49]. The authors do not have enough experience with this to comment directly. However, the Java ecosystem (Stack Overflow etc.) is far from security-aware [50]. The fact that an application is in Java doesn't mean it's free from security coding errors: see [51] for a recent example.

There is a static analysis security tool for Java described in [11]. As with [47], this has "taint analysis" as its major feature, and at the time it spotted some significant-seeming problems.

5.4 JavaScript

JavaScript is a particular problem for Security. There are some verification tools, e.g. GATEKEEPER as described in [52]. However, even if it were possible to guarantee a particular piece of stand-alone JavaScript, that is not how the current paradigm operates. As [53] writes:

> Much of the power of modern Web comes from the ability of a Web page to combine content and JavaScript code from disparate servers on the same page. While the ability to create such mash-ups is attractive for both the user and the developer because of extra functionality, code inclusion effectively opens the hosting site up for attacks and poor programming practices within every JavaScript library or API it chooses to use.

Though not explicit in this statement, an additional weakness is that this combination is *dynamic*. The obvious solution would be some kind of sandboxing of the external resources relied upon, but the nature of JavaScript makes this difficult. [54] describe one such sandboxing, but it only works for a subset of JavaScript and relies on a combination of filtering, rewriting and wrapping to guarantee security. That it can do so at all is a remarkable feat of formal methods, given that previous attempts such as Facebook's FBJS have subtle flaws [55], and that the formal semantics of JavaScript being relied upon are very much a piece of reverse engineering.

In fact the dynamic loading from multiple sites is often not good for performance, and web performance engineers recommend tools to bundle the pages: this could usefully be combined with the sort of protection described by [54].

An alternative solution is suggested by Google, who are introducing a form of taint analysis into Chrome [56] through run-time typing. When enabled, this means that the 60+ dangerous DOM API functions can only be called with arguments whose type is that emitted by `TrustedTypes` functions. Google expects that these functions would be manually verified, but this does open the door to formal verification of *certain* security policies in what is currently a very challenging environment for formal methods. However, these checks can be easily fudged, and the authors foresee examples of this on StackOverflow analogous to the `csrf().disable()` "suggestion" described below in point 3.

6 Education

[13, Requirement 6.5] called for education of developers. Education of mainstream programmers, as opposed to cybersecurity specialists, in cybersecurity has been neglected until recently, and this neglect has been lamented as far as the Harvard Business Review [57]. Developments in school curricula [58], major national initiatives [59], pedagogy and practice [60–62], and professional accreditation are changing this [63,64]. However, there are limitations, even beyond *errare humanum est*, in relying on education.

1. There is experimental evidence that both trained students [65] and professional developers [66] will ignore security considerations unless *explicitly* instructed to take them into account. Lest this be thought to be a purely academic exercise with little relevance to the real world, consider the recent ¥55M password problem described in [67].
2. There is field evidence that explicit requirements such as [13] are ignored in practice, e.g. the Forever 21 breach [68], or Macy's [69]. They may also not be communicated down the software supply chain, as in the Ticketmaster case [70].
3. Many educational resources, both formal textbooks [71] and informal resources such as Stack Overflow [72], pay very little attention to security, and indeed can be positively harmful. The discussion in Stack Overflow (analysed in [50, Sect. 4.3.1]) of cross-site request forgery (CSRF—this was in the OWASP top 10 in 2013 [73], but dropped from [10] "as many frameworks include CSRF defenses") is especially worrying. By default, Spring implicitly enables protection against this. But all the accepted answers to CSRF-related failures simply suggested disabling the check. There were no negative comments about this, and indeed a typical response is "Adding `csrf().disable()` solved the issue!!! I have no idea why it was enabled by default".

As we have noted, [13] both mandates education and does not rely solely on it.

However, as the safety-critical community laments (at least in the U.K. and U.S.A.: cultures do differ here), there is very little training in formal methods for most undergraduates, and hence it is unrealistic to expect most of those to whom PCI DSS applies to transition suddenly to formal methods: there is a supply/demand "Catch-22" situation here.

7 Conclusions

As the media never tire of saying, there are far too many security breaches, and, though they have multiple causes, [12] claims that about 50% of security breaches are caused by coding errors. There appears to be a culture of accepting these, with the U.S. Government investigation [7] into Equifax blaming many factors but not the actual bug, and [13] taking a "necessary but not sufficient" approach to education in secure coding.

Education Could certainly do better [57], though there are encouraging signs that more cybersecurity, though not necessarily formal methods, is being taught [63] and useful ideas when it comes to improving informal resources [74]. However, informal resources can be dangerous when it comes to security, and [63] recommends giving *all* students the advice in [75]: "If you pick up a SSL/TLS answer from Stack Overflow, there's a 70% chance it's insecure". More training in formal methods would be welcomed, at least in those cultures where it is lacking.

Customers/Managers need to be much more upfront about security requirements [65,66], and enforce (e.g. by requiring tool support during any CI/CD process, such as [37] describe) at least "middle ground" requirements. In the case of outsourced development, explicit penalty clauses for failing penetration tests should concentrate the developers' minds.

C/C++ people These programmers should be much more aware of techniques for secure coding, such as those described in [46, Appendix F], and the various tools for static analysis.

Java people In view of the significance of injection attacks (Number 1 in [10]), programmers should be aware of taint analysis, as in [11].

JavaScript people There are some techniques, such as [54], for protecting JavaScript applications, but they are not deployable in the typical JavaScript "dynamic loading web page" environment. Furthermore this environment is basically antithetical to security, as British Airways is learning to the cost of £183M [5].

(1) Hence the first real challenge of JavaScript lies with the tool makers: there are, as far as the authors know, no JavaScript verifiers in existence, and no page-bundler that checks for version drift, or does incremental verification (which might be comparatively cheap, as in [37]).

(2) An alternative approach might be to change the JavaScript model. This is advocated in [76], based on their analysis of what third-party scripts do in the wild. This is not a completely radical idea: Google is testing its `TrustedTypes` feature [56], with the motivation "The DOM API is insecure by default and requires special treatment to prevent XSS".

Empirical Research There is not much analysis of the efficacy of various techniques in security programming. [77] compares various techniques, and states the following.

> Based on our case study [of two large programs], the most efficient vulnerability discovery technique is automated penetration testing. Static analysis finds more vulnerabilities but the time it takes to classify false positives makes it less efficient than automated testing.

This assumes that "false positives" are acceptable, a debatable point of view. It would be good to have more such research.

Tool developers there is a lack of tools (or at least a lack of awareness of tools) that can be neatly integrated into a security programming toolchain in the way such tools are integrated in safety-critical toolchains [37].

Acknowledgements. A predecessor of this paper was given at the 2019 Working Formal Methods Symposium (FROM2019) in Timișoara, Romania. The authors are grateful to the referees and audiences of FROM2019 and FMFun2019 for useful comments. The first author is grateful to the Fulbright Programme for a Cybersecurity Scholarship at New York University in 2017, and to many correspondents and discussions, notably Alastair Irons, Tom Prickett and Tim French. This paper was partially supported by the Institute of Coding, which received £20m of funding from the Office for Students (OfS), as well as support from the Higher Education Funding Council for Wales (HEFCW).

References

1. Jacquel, M., Berkani, K., Delahaye, D., Dubois, C.: Verifying B proof rules using deep embedding and automated theorem proving. In: Barthe, G., Pardo, A., Schneider, G. (eds.) SEFM 2011. LNCS, vol. 7041, pp. 253–268. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-24690-6_18
2. Bloomberg: Equifax Hack Lasted for 76 Days, Compromised 148 Million People, Government Report Says (2018). http://fortune.com/2018/12/10/equifax-hack-lasted-for-76-days-compromised-148-million-people-government-report-says/
3. Irwin, L.: Marriott downgrades severity of 2018 data breach: 383 million customers affected (2019). https://www.itgovernance.co.uk/blog/marriott-downgrades-severity-of-2018-data-breach-383-million-customers-affected
4. Ford, N.: Medical debt collection agency files for bankruptcy protection after data breach (2019). https://www.itgovernance.co.uk/blog/medical-debt-collection-agency-files-for-bankruptcy-protection-after-data-breach
5. The Guardian: BA faces & #x00A3;183m fine over passenger data breach (2019). https://www.theguardian.com/business/2019/jul/08/ba-fine-customer-data-breach-british-airways
6. Royal Society: Progress and research in cybersecurity: supporting a resilient and trustworthy system for the UK (2016). http://royalsociety.org/cybersecurity
7. United States Government Accountability Office: Actions Taken by Equifax and Federal Agencies in Response to the 2017 Breach (2018). https://www.gao.gov/assets/700/694158.pdf
8. Osborne, C.: Google patches 'awesome' XSS vulnerability in Gmail dynamic email feature (2019). https://www.zdnet.com/article/google-patches-awesome-xss-vulnerability-in-gmail/
9. Lenart, L.: Security Bulletin S2-045 (2017). https://cwiki.apache.org/confluence/display/WW/S2-045
10. Open Web Application Security Project (OWASP): The Ten Most Critical Web Application Security Risks (2017). https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project#tab=Main
11. Livshits, V., Lam, M.: Finding security vulnerabilities in Java applications with static analysis. In: Proceedings USENIX Security Symposium, pp. 271–286 (2005)
12. McGraw, G.: Software Security—Building Security In. Addison-Wesley, Boston (2006)
13. Payment Card Industry Security Standards Council (PCI SSC): Requirements and Security Assessment Procedures Version 3.2.1 (2018). https://www.pcisecuritystandards.org/documents/PCI_DSS_v3-2-1.pdf
14. Ponemon Institute: The State of Web Application Firewalls. Ponemon Institute (2019)
15. Krebs, B.: What We Can Learn from the Capital One Hack (2019). https://krebsonsecurity.com/tag/capital-one-breach/
16. Kolochenko, I.: Web Application Firewall: a must-have security control or an outdated technology? (2016). https://www.csoonline.com/article/3032743/web-application-firewall-a-must-have-security-control-or-an-outdated-technology.html
17. Barth, B.: No fly-by-night operation: Researchers suspect Magecart group behind British Airways breach (2018). https://www.scmagazine.com/home/security-news/no-fly-by-night-operation-researchers-suspect-magecart-group-behind-british-airways-breach/

18. Klein, G., et al.: seL4: formal verification of an OS kernel. In: Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles, pp. 207–220 (2009)
19. The Guardian: Hacking risk leads to recall of 500,000 pacemakers due to patient death fears (2017). https://www.theguardian.com/technology/2017/aug/31/hacking-risk-recall-pacemakers-patient-death-fears-fda-firmware-update
20. Newman, L.: Hackers Made an App That Kills to Prove a Point (2019). https://www.wired.com/story/medtronic-insulin-pump-hack-app
21. Evans, M., Loftus, P.: Rattled by Cyberattacks, Hospitals Push Device Makers to Improve Security (2019). https://www.wsj.com/articles/rattled-by-cyberattacks-hospitals-push-device-makers-to-improve-security-11557662400
22. Food and Drug Administration: FDA Informs Patients, Providers and Manufacturers About Potential Cybersecurity Vulnerabilities in Certain Medical Devices with Bluetooth Low Energy (2020). https://www.fda.gov/news-events/press-announcements/fda-informs-patients-providers-and-manufacturers-about-potential-cybersecurity-vulnerabilities-0
23. Heiser, G.: What's new in the world of seL4 (2019). https://archive.fosdem.org/2019/schedule/event/world_of_sel4/
24. Rescorla, E.: The Transport Layer Security (TLS) Protocol Version 1.3. RFC-8446 (2018)
25. Beck, K., et al.: The Agile Manifesto (2001). http://agilemanifesto.org/
26. Blodget, H.: Mark Zuckerberg on Innovation (2009). https://www.businessinsider.com/mark-zuckerberg-innovation-2009-10
27. Lane, A.: Security + Agile = FAIL (2018). https://securosis.com/assets/library/presentations/Security/AgileFAIL_OWASP.ppt_.pdf
28. Bartsch, S.: Practitioners' perspectives on security in agile development. In: International Conference on Availability Reliability and Security, pp. 479–484 (2011)
29. van der Heijden, A., Broasca, C., Serebrenik, A.: An empirical perspective on security challenges in large-scale agile software development. In: Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM 2018, pp. 45:1–45:4. ACM, New York (2018)
30. Tahaei, M., Vaniea, K.: A Survey on Developer-Centred Security (2019). https://groups.inf.ed.ac.uk/tulips/papers/A_Survey_on_Developer_Centred_Security.pdf
31. Chapman, R.: Industrial experience with Agile in high-integrity software development. In: Parsons, M., Anderson, T. (eds.) Developing Safe Systems: Proceedings of the Twenty-fourth Safety-critical Systems Symposium, Safety-Critical Systems Club, pp. 143–154 (2016)
32. O'Connor, T., Enck, W., Reaves, B.: Blinded and confused: uncovering systemic flaws in device telemetry for smart-home Internet of Things. In: Proceedings of the 12th Conference on Security and Privacy in Wireless and Mobile Networks, pp. 140–150 (2019)
33. Wäyrynen, J., Bodén, M., Boström, G.: Security engineering and eXtreme programming: an impossible marriage? In: Zannier, C., Erdogmus, H., Lindstrom, L. (eds.) XP/Agile Universe 2004. LNCS, vol. 3134, pp. 117–128. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-27777-4_12
34. Statt, N.: Zuckerberg: 'Move fast and break things' isn't how Facebook operates anymore (2014). https://www.cnet.com/news/zuckerberg-move-fast-and-break-things-isnt-how-we-operate-anymore/
35. Salz, R.: Software engineering and OpenSSL is not an oxymoron (presentation at Real World Cryptography 2017) (2017). https://rwc.iacr.org/2017/Slides/rich.saltz.pdf

36. Seggelmann, R., Tuexen, M., Williams, M.: Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS) Heartbeat Extension (2012). https://tools.ietf.org/html/rfc6520
37. Brain, M., Schanda, F.: A lightweight technique for distributed and incremental program verification. In: Joshi, R., Müller, P., Podelski, A. (eds.) VSTTE 2012. LNCS, vol. 7152, pp. 114–129. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-27705-4_10
38. Chapman, R., Moy, Y.: AdaCore Technologies for Cyber Security (2018). https://www.adacore.com/books/adacore-tech-for-cyber-security
39. Distefano, D., Fähndrich, M., Logozzo, F., O'Hearn, P.: Scaling static analyses at Facebook. Commun. ACM **62**, 62–70 (2019)
40. Vogels, W.: Proving security at scale with automated reasoning (2019). https://www.allthingsdistributed.com/2019/05/proving-security-at-scale-with-automated-reasoning.html
41. Sadowski, C., Aftandilian, E., Eagle, A., Miller-Cushion, L., Jaspan, C.: Lessons from building static analysis tools at Google. Commun. ACM **61**(4), 58–66 (2018)
42. Open Web Application Security Project (OWASP): DefectDojo: OpenSource Application Security Management (2019). https://www.defectdojo.org
43. Chapman, R.: Development and Formal Verification of Secure Updates for Embedded Systems (slides from Verification 2018) (2018). http://www.testandverification.com/conferences/verification-futures/vf2018/
44. Google (Project Zero): 0day "In the Wild" (2019). https://googleprojectzero.blogspot.com/p/0day.html
45. Thomas, G.: A proactive approach to more secure code (2019). https://msrc-blog.microsoft.com/2019/07/16/a-proactive-approach-to-more-secure-code/
46. Centre for the Protection of National Infrastructure: Rail Code of Practice for Security-Informed Safety. CPNI (2019)
47. ISO/IEC: TS 17961:2013, Information technology—Programming languages, their environments & system software interfaces—C Secure Coding Rules (2013). https://www.iso.org/standard/61134.html
48. Chong, N., et al.: Code-level model checking in the software development workflow. In: ICSE-SEIP 2020 (2020, to appear)
49. Cavalcanti, A., Miyazawa, A., Wellings, A., Woodcock, J., Zhao, S.: Java in the safety-critical domain. In: Bowen, J.P., Liu, Z., Zhang, Z. (eds.) SETSS 2016. LNCS, vol. 10215, pp. 110–150. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-56841-6_4
50. Meng, N., Nagy, S., Yao, D., Zhuang, W., Arango Argoty, G.: Secure coding practices in Java: challenges and vulnerabilities. In: 2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE), pp. 372–383 (2018)
51. Google (Chris Povirk): Denial of Service vulnerability for servers that use Guava and deserialize attacker data (2018). https://groups.google.com/forum/#!topic/guava-announce/xqWALw4W1vs/discussion
52. Guarnieri, S., Livshits, B.: GATEKEEPER: mostly static enforcement of security and reliability policies for JavaScript code. In: USENIX Security Symposium, vol. 10, pp. 76–85 (2009)
53. Meyerovich, L., Livshits, B.: ConScript: specifying and enforcing fine-grained security policies for JavaScript in the browser. In: 2010 IEEE Symposium on Security and Privacy, pp. 481–496. IEEE (2010)
54. Maffeis, S., Mitchell, J.C., Taly, A.: Isolating JavaScript with filters, rewriting, and wrappers. In: Backes, M., Ning, P. (eds.) ESORICS 2009. LNCS, vol. 5789, pp. 505–522. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-04444-1_31

55. Maffeis, S., Taly, A.: Language-based isolation of untrusted JavaScript. In: Proceedings 22nd IEEE Computer Security Foundations Symposium, pp. 77–91 (2009)
56. Kotowicz, K.: Trusted Types help prevent Cross-Site Scripting (2019). https://developers.google.com/web/updates/2019/02/trusted-types
57. Cable, J.: Every Computer Science Degree Should Require a Course in Cybersecurity (2019). https://hbr.org/2019/08/every-computer-science-degree-should-require-a-course-in-cybersecurity
58. Brown, N.C.C., Sentance, S., Crick, T., Humphreys, S.: Restart: the resurgence of computer science in UK schools. ACM Trans. Comp. Sci. Edu. **14**(2), 1–22 (2014). https://doi.org/10.1145/2602484
59. Davenport, J.H., Crick, T., Hourizi, R.: The institute of coding: a university-industry collaboration to address the UK's digital skills crisis. In: Proceedings of IEEE Global Engineering Education Conference, pp. 1400–1408. IEEE Press (2020). https://doi.org/10.1109/EDUCON45650.2020.9125272
60. Davenport, J.H., Hayes, A., Hourizi, R., Crick, T.: Innovative pedagogical practices in the craft of computing. In: Proceedings of 4th International Conference on Learning and Teaching in Computing and Engineering (2016). https://doi.org/10.1109/LaTiCE.2016.38
61. Murphy, E., Crick, T., Davenport, J.H.: An analysis of introductory programming courses at UK universities. Art Sci. Eng. Prog. **1**(2), 18 (2017). https://doi.org/10.22152/programming-journal.org/2017/1/18
62. Crick, T., Davenport, J.H., Hanna, P., Irons, A., Prickett, T.: Overcoming the challenges of teaching cybersecurity in UK computer science degree programmes. In: Proceedings of 50th Annual Frontiers in Education Conference, IEEE Press (2020). https://doi.org/10.1109/FIE44824.2020.9274033
63. Crick, T., Davenport, J., Irons, A., Prickett, T.: A UK case study on cybersecurity education and accreditation. In: Proceedings of FIE 2019 (2019)
64. Crick, T., Davenport, J.H., Hanna, P., Irons, A., Pearce, S., Prickett, T.: Repositioning BCS degree accreditation. ITNOW **62**(1), 50–51 (2020). https://doi.org/10.1093/itnow/bwaa023
65. Naiakshina, A., Danilova, A., Tiefenau, C., Smith, M.: Deception task design in developer password studies: exploring a student sample. In: Fourteenth Symposium on Usable Privacy and Security (SOUPS 2018), pp. 297–313. USENIX Association (2018)
66. Naiakshina, A., Danilova, A., Gerlitz, E., von Zezschwitz, E., Smith, M.: "If you want, I can store the encrypted password": a password-storage field study with freelance developers. In: Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems, pp. 140:1–140:12. ACM (2019)
67. Cimpanu, C.: 7-Eleven Japanese customers lose $500,000 due to mobile app flaw (2019). https://www.zdnet.com/article/7-eleven-japanese-customers-lose-500000-due-to-mobile-app-flaw/
68. Biscoe, C.: MyFitnessPal data breach: 150 million app users affected (2018). https://www.itgovernance.co.uk/blog/myfitnesspal-data-breach-150-million-app-users-affected/
69. Blackmon, A.: Macy's hit by data breach (2018). https://eu.freep.com/story/money/business/2018/07/06/macys-data-breach-online/763074002/
70. Inbenta (CEO): Inbenta and the Ticketmaster Data Breach (2018). http://web.archive.org/web/20181121184620/

71. Taylor, C., Sakharkar, S.: ');DROP TABLE textbooks;–: an argument for SQL injection coverage in database textbooks. In: Proceedings of the 50th ACM Technical Symposium on Computer Science Education (SIGCSE 2019), pp. 191–197. ACM (2019)
72. Fischer, F., et al.: Stack overflow considered harmful? The impact of copy&paste on Android application security. In: 38th IEEE Symposium on Security and Privacy (SP), pp. 121–136 (2017)
73. Open Web Application Security Project (OWASP): The Ten Most Critical Web Application Security Risks (2013). https://www.owasp.org/images/f/f8/OWASP_Top_10_-_2013.pdf
74. Fischer, F., et al.: Stack overflow considered helpful! Deep learning security nudges towards stronger cryptography. In: 28th USENIX Security Symposium (USENIX Security 2019), pp. 339–356 (2019)
75. Chen, M., Fischer, F., Meng, N., Wang, X., Grossklags, J.: How reliable is the crowdsourced knowledge of security implementation? https://arxiv.org/abs/1901.01327 (2019)
76. Zhang, M., Meng, W., Lee, S., Lee, B., Xing, X.: All Your Clicks Belong to Me: Investigating Click Interception on the Web (2019). https://www.microsoft.com/en-us/research/uploads/prod/2019/03/zhang-observer.pdf
77. Austin, A., Williams, L.: One technique is not enough: a comparison of vulnerability discovery techniques. In: Proceedings 2011 International Symposium on Empirical Software Engineering and Measurement, pp. 97–106 (2011)

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

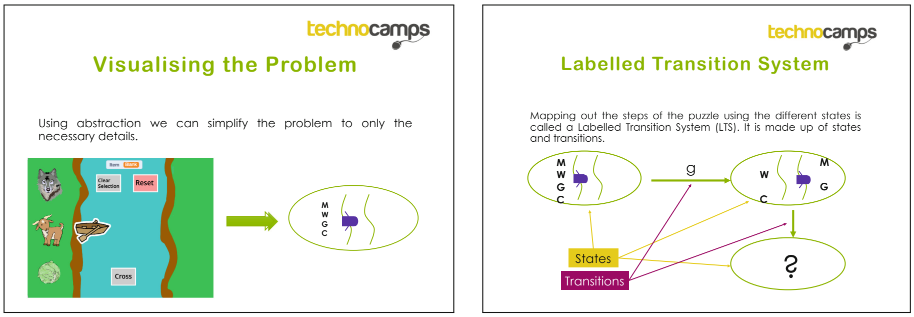
The following riddle was posed by Alcuin of York in the 8th century, and more recently tackled by Homer Simpson in a 2009 episode of The Simpsons titled Gone Maggie Gone (which provides the ideal way to introduce the problem).

> *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.


Visualising the Problem

Using abstraction we can simplify the problem to only the necessary details.

Labelled Transition System

Mapping out the steps of the puzzle using the different states is called a Labelled Transition System (LTS). It is made up of states and transitions.

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.

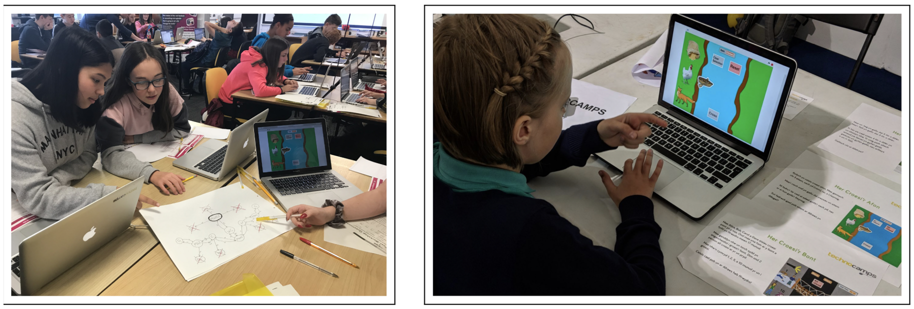


Fig. 3. LTS modelling in the classroom

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.

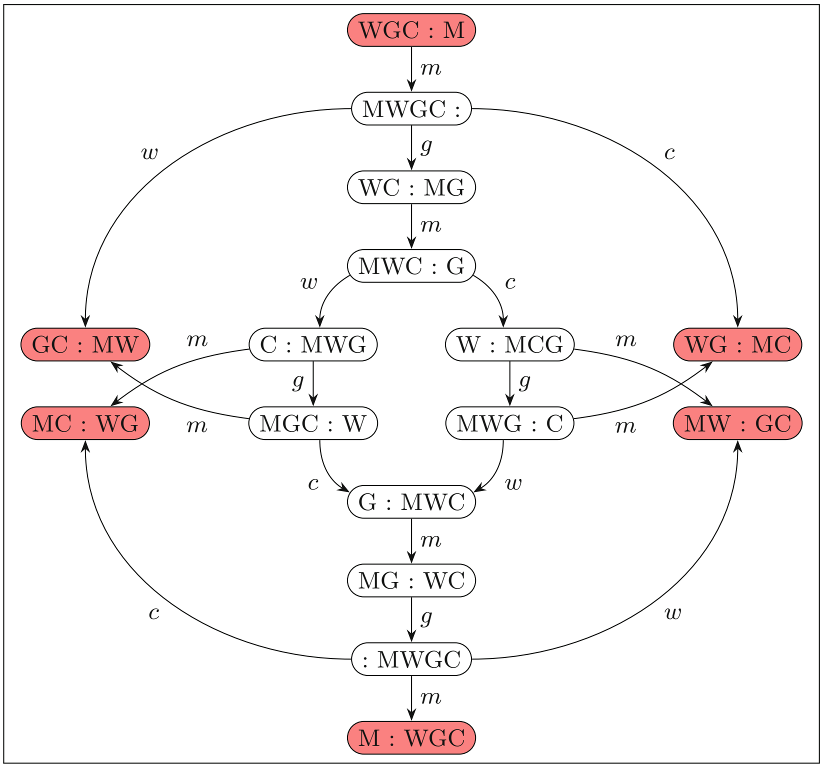


Fig. 4. The Man-Wolf-Goat-Cabbage LTS.

The initial state is $\boxed{\text{MWGC :}}$ (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 $\boxed{\text{: MWGC}}$ (meaning all are on the right bank of the river). However, we want to avoid going through any of the six dangerous (red) states:

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

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.)

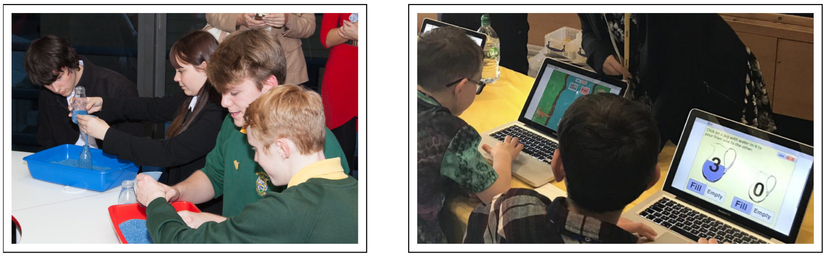


Fig. 5. Solving the water jug puzzle in Computational Thinking workshops

> *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):

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

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

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

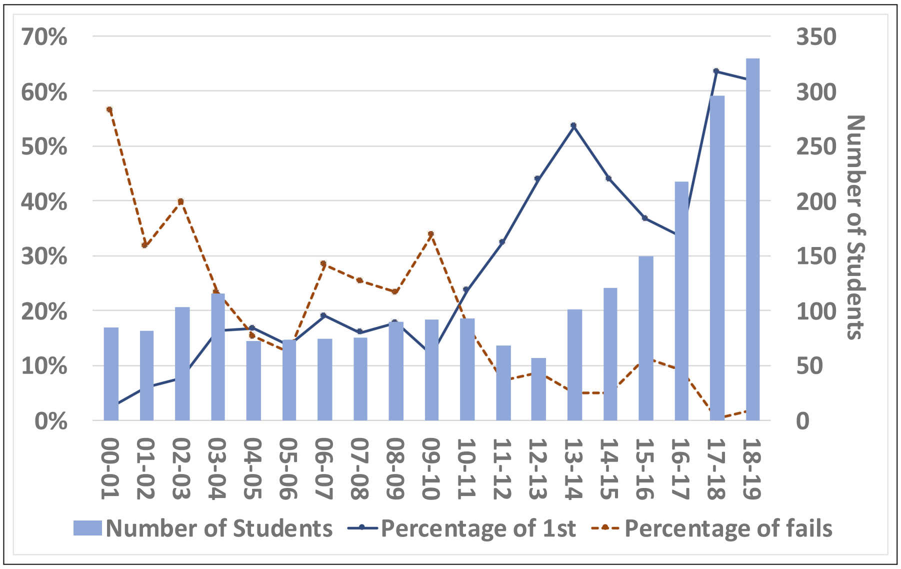


Fig. 6. Trends of students achieving 1st-class and failing results; and class sizes.

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 $(x, y) \in R$:

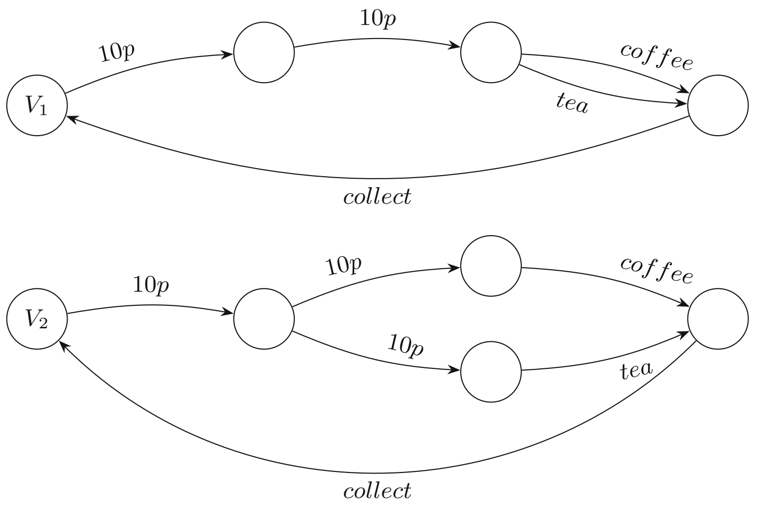


Fig. 7. Two Vending Machines.

- 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.

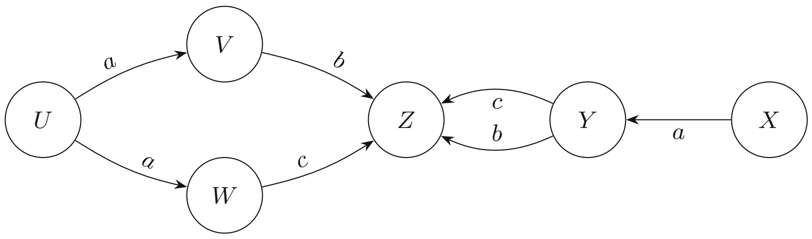


Fig. 8. A simple LTS.

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

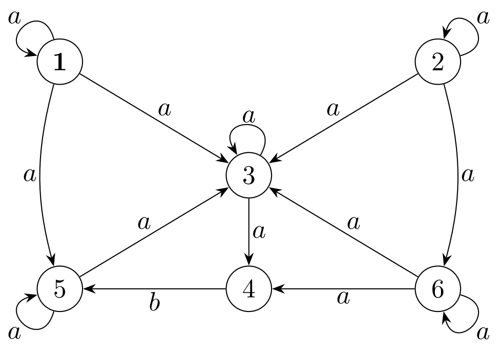
Once the notion of equivalence is understood in terms of winning strategies in the copy-cat game, the question then arises as to how to determine if two particular states are equivalent, i.e., if Bob has a winning strategy starting with the tokens on the two given states. This isn't generally a simple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

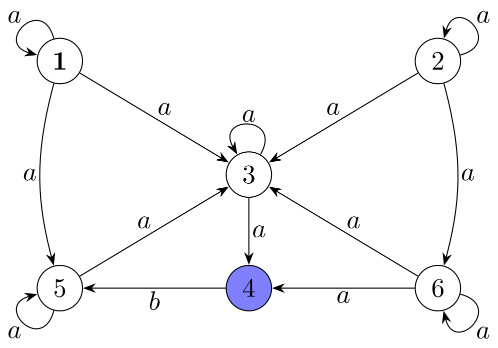
> *If any state with some colour C has a transition leading out of it into a state with some colour C', then* every *state with colour C has an identically-labelled transition leading out of it into a state coloured C'.*

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*.

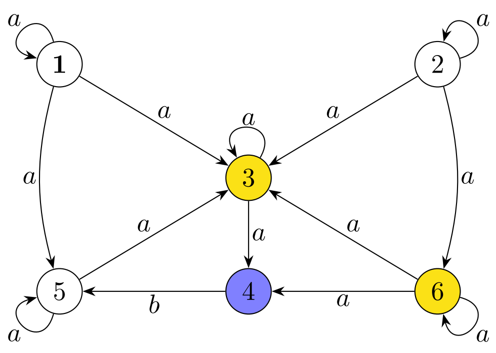
To demonstrate, consider the following LTS.



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)

From Stories to Concurrency: How Children Can Play with Formal Methods

Antonio Cerone(✉)

Department of Computer Science, Nazarbayev University, Nur-Sultan, Kazakhstan
antonio.cerone@nu.edu.kz
https://cs-sst.github.io/faculty/cerone

Abstract. This position paper presents an unplugged, problem-solving-based approach for teaching computer science to children. Our approach is based on story telling, where each story consists of parallel parts, and aims at developing children's observation and reasoning skills. The aim is to understand the global plot by identifying the interaction occurring among different characters in terms of synchronisation, collaboration and information sharing. In this sense we focus on concurrency, a very challenging computer science area, to show that children aged 7–14 can be exposed to real-life instantiations of a number of computer science concepts, understand them and even apply them in modelling and analysis contexts.

Keywords: Children Education · Problem-solving · Computer science · Concurrency · Finite State Machines

1 Introduction

Computer science lectures tend to point out that their first year students have poor mathematical skills. It is important to stress that mathematics should not be confused with elementary arithmetics and mathematical skills should not be confused with the ability to quickly perform complex calculations mentally [7]. It is emblematic that the ability to mentally perform complex calculations was portrayed by the idiot-savant protagonist of the famous film "The Rain Man", who was an autistic person rather than a genius of mathematics.

In fact, when lecturers describe their students' poor mathematical skills, they do not refer to mere calculation skills, but to the large amount of reasoning skills that enable us to solve general problems, within and outside the domain of mathematics [6]. An essential, though non exhaustive list of such skills is:

– be able to abstract away from irrelevant details (*abstraction*) and model the reality in a symbolic/visual way (*modelling*), not necessarily on paper but even just mentally;

Work partly funded by Seed Funding Grant, Project SFG 1447 "Formal Analysis and Verification of Accidents", University of Geneve, Switzerland.

- understand the difference between a visual description (a sort of *formal syntax*) and the reality it represents (its *semantics*);
- find similarities between a problem with a known solution and a new, unsolved problem (*analogy*);
- reduce a complex problem to a smaller, easier one (*divide et impera*, leading to the concept of *recursion*);
- reason top-down from a model to a more articulated, efficient solution (*refinement*);
- reason bottom-up from observed specific cases and pattens to a general law (*induction* or *generalisation*);
- compose components, possibly modifying them to achieve *compositionality*;
- move from causes to consequences (*deduction*);
- distinguish between efficient and inefficient solutions (*complexity*);
- understand the difference between a solution of a problem and the *proof* that such a solution is *correct*, as well as develop the abilities to perform such proofs and describe them to others.

A number of these skills will be extensively addressed in this paper.

Given that computers have been heavily introduced in schools with the expectation of having a positive impact on the students' computer science skills, the fact that this expectation has not been met at all, as observed by many university lecturers, sounds like a paradox [6]. In many schools computer science has even been introduced as a new, stand-alone subject. However, there are two fundamental problems in this innovative process:

1. computer science is often seen as a "service subject", namely to provide tools that are supposed to enhance learning by facilitating the students in carrying out their homework and class projects;
2. computer science is normally seen as intrinsically tied to the use of computers.

We do not consider here the most extreme, but not uncommon situation in which computer science is taught, either as part of another subject or as a stand-alone subject, by unwilling teachers who did not undergo a proper training.

As a consequence of Problem 1, the teaching of computer science tends to focus on using office-oriented tools to write documents, prepare presentations, organise data in spreadsheets. There is no need here to mention the names of the most taught tools.

In this sense the relation of the subject "computer science" with other subjects is only in one direction. It is, in fact, the other direction, the one normally neglected, that should be taken. Computer science should build on other school subjects, which, in the world of the school pupil, represent the most natural reality to be modelled formally. Obviously, mathematics should be the first subject to provide materials to manipulate in a computer science fashion. However, all other subjects also have plenty of materials on which students may carry out modelling and analysis.

Problem 2 created the misconception of a computer scientist as a programmer. This attitude also contributed to create the belief that computer science does not require mathematical skills. As a consequence, high school students who are not skilled in mathematics and are aware of this, are still confident to pursue a computer science degree.

In this paper we adopt an "unplugged approach" to teaching computer science [1,2], presenting activities that foster children reasoning and do not require the use of a computer. In fact, we show how, starting from what at first sight appears as a purely literary and linguistic exercise, namely story telling, children carry out observations and perform reasoning leading to the acquisition of important aspects of concurrency, develop a formal model of the story, which allows them to solve the puzzle embedded in the story, and are even enabled to describe the reasoning process that led to the solution as a formal proof. We consider children between 7 and 14 years old and propose slightly different approaches for the two age groups, 7–10, approximately corresponding to primary school, and 11–14, approximately corresponding to middle school.

It is pointless to just provide children with the definitions of new notions, concepts and processes, such as algorithms, and hope they understand them, remember them and are then able to apply them to practical situations. Children learn best if they are actively involved in the process through problem-solving [14]. This is the main idea of *constructivism*, which suggests that humans construct knowledge and meaning from their experiences [4,5]. And to engage children, experiences have to be fun and challenging. Learning from experience means that children have to discover an algorithm, starting from a game, and progressively perfect their discovery through further experiences in a fashion similar to *iterative refinement*.

However, in mathematics, it is essential to find appropriate challenges for the age and cognitive development of the child to avoid loss of interest or even frustration, with the consequent end of the fun. Mathematical puzzles are in general very motivating, but may have the drawback to degenerate in frustration. It is also important to make sure that the higher the effort needed to solve a puzzle, the greater the learning outcome.

Moreover, although some form of competition is necessary for keeping the children involved, the competition must be in the game itself, not in the mathematical skill the game addresses. It is fundamental that the competition does not appear to children as an assessment of their skills. There are plenty of evidences in psychology research that assessments increase anxiety in children and hinder their learning [12]. One way to overcome this problem is to organise competitions among teams rather than between individuals, with a balanced composition of the teams and a frequent mixing of the members, and trying to make the winning objective as much as possible distinct from the winning skills.

Although the author aims to test the approached proposed by this paper in a real classroom context, this has not been possible yet due to the lengthy ethical clearance processes as well as the difficulty in finding a willing hosting school. Therefore, the author experimented his approach with his own children, Claudio and Chiara, respectively 9 years old and 13 years old at the moment when the paper was written. All the drawings presented in the paper are by Claudio and Chiara.

1.1 Playing with Concurrency

Concurrency [9–11] is a challenging topic even for postgraduate students. Understanding the global behaviour of two or more synchronising processes may not be intuitive also for simple models. It is therefore very important to develop the skills that allow us to visualise and then understand and model concurrent behaviour.

Children are very much interested in complex stories acted by many characters, who have their own personal stories, but also interact and synchronise on specific situations, collaborate to solve mysteries and fight together to defeat antagonists. The fact that children can follow and enjoy such articulated stories is evidence that they can make sense of the composition of individual stories and understand the resultant global plot. They can, therefore, visualise and understand concurrent behaviour. We aim at exploiting these abilities and prompt children with questions that enable them to reason about the composition of parallel stories, to understand how two or more characters may agree or collaborate, by sharing information and sychronising their decisions and behaviour in specific situations. In this way children can become aware of important aspects of concurrency.

Many children can naturally solve complex problems but, when they are asked about how they achieved the solution, they normally have difficulties in explaining their reasoning process. Formal methods [15] provide rigorous ways of describing problems normally occurring in computer science contexts and they are also very effective in making the reasoning processes that lead to the discovery of the solutions explicit. Among the computer science sub-disciplines, concurrency is probably the one that relies most on formal methods.

In our work we are inspired by the *Choose Your Own Adventure* series of children's gamebooks. The concept underlying this book series was created by Edward Packard, who published the first book in 1976 [13]. In Packard's books, stories are written from a second-person point of view and require the reader to impersonate the protagonist by making choices among a number of different alternatives. It is thus the reader who actually determines the character's behaviour and drives the story plot.

In our approach we consider two parallel stories, whose protagonists are the reader and a friend. We believe that the use of the second-person point of view in both the person's and friend's stories increases the involvement of the child, fosters the expression of personal opinions and the realistic making of informed decisions.

1.2 Structure of the Paper

In Sect. 2 we start from the example of a story featuring one single character, who can choose among various alternatives. We then add a parallel story, whose single character is a friend of the character of the previous story (Sect. 2.1), and we make the children reason about the knowledge acquired by the two characters, how it may affect their decisions and how combining what the two characters know allows us to predict the outcome of each decision.

In Sect. 3 we introduce interactions between the protagonists of the two stories and guide the children to reason about the impact of such interactions on the global plot. We also make important considerations on the teacher's role and the learning outcome of this exercise (Sect. 3.1).

In Sect. 4 we investigate how to enable children to visually model stories in a sort of formal way and, especially the ones in the age group 10–14, to get familiar at an intuitive and visual way with some important aspects of the modelling and analysis of concurrent systems (Sects. 4.1–4.3).

In Sect. 5, we show how children can be guided to use the models they developed to find solutions of a problem and, most important, to prove the correctness of such solutions. Section 6 concludes the paper.

2 Choose Your Own Adventure

Let us consider the following story.

> You are looking for a treasure hidden in an abandoned castle. You enter the castle and you have in front of you a long corridor with many windows on the right side. At the end of the corridor there is a large door guarded by two parrots on their tripods. They both speak but you understand only the one on the left. The other parrot speaks a language unknown to you. The parrot on the left tells you that behind the large door there is a wide room with three small doors of different colours: the green and blue doors are not locked and you can open them and go through; the red door is locked and you do not have the key. The parrot also tells you that one of the two unlocked doors safely leads to the treasure and that if you go through the other unlocked door you will certainly die without finding the treasure.

After the story is presented to the class, children are then asked a number of questions, such as:

1. Which are your possible choices?
2. Which of these choices will certainly lead you to death?
3. Which are your reasonable choices?
4. Which choice would you make?

The questions are put to the entire class through a discussion session that aims at unfolding the logic of the story and understanding which decisions are favourable to the protagonist among the set of possible decisions. New, unplanned questions are likely to be raised during the discussion.

For children between 7 and 10 years old it is important to give a multidisciplinary flavour to the discussion by considering also literary and linguistic aspect of the story. In fact, the logical analysis of the text also contributes to these aspects.

2.1 Parallel Adventures

We add now the following story in parallel to the one presented in Sect. 2.

> Your friend is also looking for the treasure. You both start at the same time but following different paths and getting to the castle at different times. Your friend understands only the parrot on the right of the large door. The other parrot speaks a language unknown to your friend. The parrot on the right tells your friend that inside the vase next to the last window of the corridor your friend can find the key that opens the red door. In addition, the parrot tells your friend that going through the green unlocked door will certainly lead to death without finding the treasure and that, regarding the other two doors, one will safely lead to the treasure and the other will certainly lead to death. Obviously you do not know what the parrot on the right tells your friend and your friend does not know what the parrot on the left tells you. Moreover, neither you nor your friend are aware of each other search for the treasure.

Children are again asked the questions from Sect. 2, this time obviously referred to their friend. In discussing and answering the questions, the children should not take into account what they know about the first story. Although this could be effectively achieved with a variant of the game in which the two stories are told to two distinct groups of children, for simplicity we assume that the entire class is told the two stories in sequence.

In a second phase of the discussion the children are urged to combine the information of the two stories. Typical questions during this phase could be:

1. Is the key needed to reach the treasure?
2. Which door leads to the treasure?
3. Can you be sure that you reach the treasure without dying?

However, such questions should not be provided by the teacher and the expectation is that they are raised spontaneously (and correctly answered) during the discussion. The teacher's role in leading the discussion with no coercion or bias is central here, as we will discuss in Sect. 3.1.

Most children, independently of the age group, would find the solution following a sort of deductive approach by considering the two persons' options, extracting a person's knowledge about the negative outcome, and using it to rule out one of the other person's possible options, thus leaving the other option as the globally positive outcome. In our simple story example, the child can exploit the information known by the protagonist's friend that the green door leads to death to rule out such a door in the protagonist's options and leave the blue door as the solution.

It is always important to urge the children to describe the reasoning process they followed to find the solution of a problem. We will illustrate in Sect. 4 how children can develop a formal model of a problem and in Sect. 5 how they can use and enrich the model to represent a formal proof.

Some Children aged 11–14, who possibly had been trained to develop tabular representations of problems and their solutions, may perform a systematic analysis of the global plot using tools similar to Table 1. In fact, from the contents of Table 1 it is immediate to deduce that the blue door leads to the treasure from the premises that the green door leads to death and either the green or the blue door leads to the treasure.

Table 1. Knowledge of the two protagonists

door $\longrightarrow$	green door	blue door	red door
$\downarrow$ person	(unlocked)	(unlocked)	(locked)
I (without the key)	one of the two leads to the treasure and the other leads to death		not accessible
My friend (with the key)	it leads to death	one of the two leads to the treasure and the other leads to death	

The discussion on parallel adventures should provide answers to questions 1 and 2. Question 3 may or may not be raised during the discussion, but cannot have a positive answer at this stage. In fact, the two friends are not aware of each other looking for the treasure. However, at this point, the discussion may spontaneously identify the possibility of a collaboration between the two friends and investigate how to enable and carry out such a collaboration. One problem is that the two friends arrive at the castle at different times, thus they are unlikely to meet each other unless they deliberately wait for each other. But they would be willing to wait for each other only if they were aware of each other looking for the treasure and they knew that sharing the information they know will enable them to safely get to the treasure. This leads to Sect. 3.

3 Synchronisation Through Collaboration and Agreement

If the children identify the possibility of a collaboration between the two friends, the discussion can be finalised to discover ways to change the story to make this collaboration as a possible decision. Otherwise the teacher will need to explicitly introduce in the story new conditions.

An example of conditions that enable collaborations is:

The parrot on the left also tells you that your friends knows
- which between the green and the blue door will certainly lead to death, and
- that what you know contains the additional information your friend needs in order to be sure to safely reach the treasure.

Children are asked once again to answer the four questions from Sect. 2, first referred to themselves, then referred to their friend. They will now notice that they will need to wait for their friend or be sure to find their friend waiting for them in order to be able to restrict the number of reasonable choices. However, this would mean to share the treasure with their friend. Therefore it might be reasonable to take the risk to die aiming to own the entire treasure. Some children might be willing to take this risk, others might not, but both the risky and the safe alternatives should be considered reasonable.

Some temporal reasoning can be carried out at this point. The children will have already identified their two possible decisions:

1. wait for their friend, unless their friend is already waiting for them, and use the combined information to choose the right door (which will appear to be the blue one);
2. randomly choose one between the green and blue doors.

At this point the children will be asked how many and which the outcomes of each of such decisions are.

For Decision 1 they would normally identify (1.1) finding the treasure and sharing it with the friend as the unique outcome. However, what if they decide to wait for their friend, but their friend does not? The friend might have already arrived and proceeded through one of the doors alone. Then (1.2) the friend would be waited for forever. This is actually a typical concurrency problem known as *starvation*. The other possibility is (1.3) that the friend arrives later but does not agree on sharing the treasure. Here a lot of potential alternatives are possible for what is going to happen, but discussing them is outside the scope of this paper, although it might be worthy from a didactical point of view.

For Decision 2 the children would normally identify two possible outcomes, (2.1) finding the treasure when going through the blue door, thus opening a number of alternatives for the friend, and (2.2) dying when going through the green door. In fact, it is likely that the children implicitly assume that they arrive at the castle before their friend. Thus, they may neglect the fact that if the friend reaches the castle before them, they might choose the blue door that definitely allows them to avoid dying but (2.3) they might no longer find the treasure, if this has already been taken away by their friend, or (2.4) they may find it, if their friend has gone through the deadly door. Finally, symmetric to the previous decision case, there is a further alternative: (2.5) the friend arrives later, which is a case of starvation, this time for the friend.

As a conclusion of this discussion, we can introduce the concept of *assumption*: when the number of possible decisions is too big, and even unclear as in alternative 1.3 above, we can assume only the most *plausible* alternatives. In our story, we could actually assume that the two friends decide to wait for each other, which, in fact, was probably the implicit assumption of most children. Some care is important here, since we should take into account the children's opinions and not making assumptions that might upset any of them. Note that a majority vote might not be always be the best in this case. Common sense and knowledge of the children's personalities are essential in this situation.

3.1 Teacher's Role and Learning Outcome

All questions considered in Sects. 2–3 should be put to the entire class rather than individually and should result in a discussion in which children can freely express their opinions and show their attitudes as risk takers or safe players. Here the teacher needs to play a neutral role, as a moderator who accepts all opinions and attitudes, possibly helping children to provide justification and rationale but without expressing any form of judgement. Furthermore, as we have noted at the end of Sect. 3 concerning assumptions, not everything can be planned in advance and common sense should be used in choosing the next steps of the game.

The aim of the exercise carried out in Sect. 3 is for the children to understand that collaboration and agreement are important in solving problems and achieving objectives, although they may require some form of compromise leading, in general, to approximate solutions of the problem or to the partial achievement of the objective (in our story the partial objective is that of getting only half of the treasure). However, the possibility of collaboration does not preclude independent actions, which might lead to better but uncertain results. The uncertainty may be not only due to randomness but also to timing issues. In computer science this is the case of *real-time* and *time-critical* systems.

The broad learning outcome we described goes well beyond mathematics and computer science, but it is definitely worthy that the class discussion covers such general, interdisciplinary aspects, although this may result in lengthy digressions. After all computer science is both a theoretical and practical/applied science with also philosophical and ethical consequences, and it is important to expose children to the practical aspects and consequences of using computer science and, more important, computer science theory and principles.

From a technical point of view, the two parallel stories are actually two concurrent processes, which may evolve independently (without collaboration) or synchronise (through collaboration).

As a final note to this section, the number of possible contexts and variations of stories is infinite. The same concepts can be illustrated through completely different story settings, as the result of the teacher's creativity or, even better, produced by the children through class discussions or working groups.

4 Modelling Stories

Throughout the discussion described in Sects. 2–3 children should be invited to illustrate the story in a visual form. This should happen with an interdisciplinary approach and may also involve visual arts, especially for the age group 7–10.

We have to note that the stories include a large number of details that are irrelevant for the offered choices. The presence of such details is important to make the stories realistic and engaging. In addition, these "literary" details offer an important context for developing *abstraction* and modelling skills.

Some guidance is needed to enable the children to identify the appropriate representation, namely the appropriate *visual formalism*, with which to create

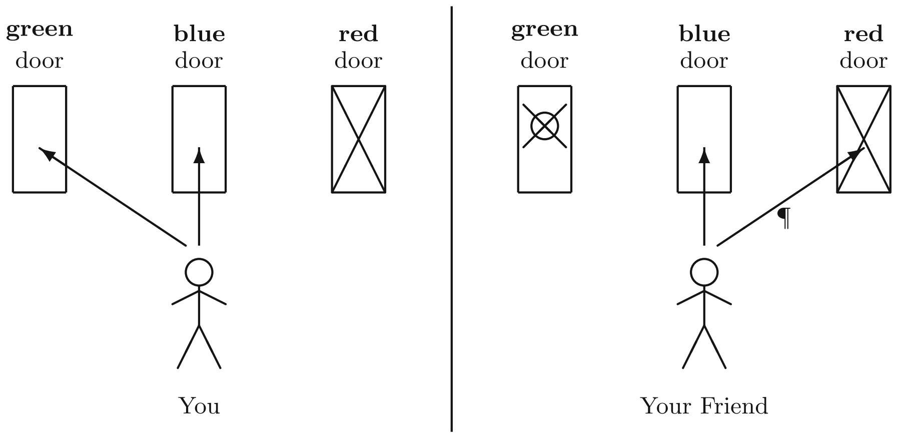


Fig. 1. Story models of reasonable choices from your perspective and your friend's perspective.

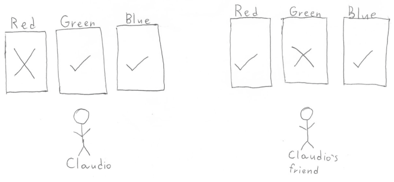
the model. The model in Fig. 1 is a typical representation for children aged 7–14. It may still contain some irrelevant details. For example, the reason why a door is not chosen, whether because it is locked and I do not have the key or because I know that it will lead me to death, is irrelevant, but it is likely that the child would still depict such details. In fact, these semantic details represent the concrete rationale for the choice, which otherwise might lose meaning in the child's mind. That is why some children may still keep such details in their models. However, more irrelevant details, such as the corridor, the vase where the key is normally hidden and even the "enchanting" detail of the two parrots are likely to be abstracted away, though this may require a number of iterations.

Some children, especially in the age group 7–9, may focus on the solution and provide a concise, abstract model, in which the relation between the person and the door, expressed by arrows in Fig. 1, is not represented. This is the case of Claudio, 9 years old, whose models for himself and his friend are given in Fig. 2(a). Chiara, 13 years old, instead, explicitly includes arrows from her to the doors. Her models for herself and her friend are given in Fig. 3(a).

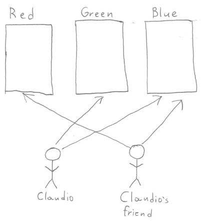
4.1 Finite State Machines, Composition and Complexity

The age group 11–14 children should be also guided to come out with a more formal model such as the finite state machine [8] in Fig. 4.

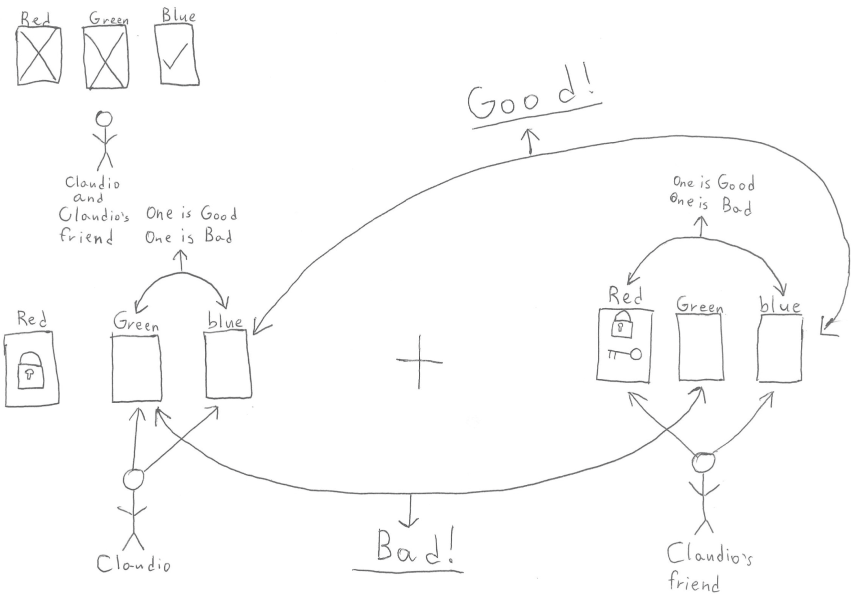
In fact, for this age group, it is also important to enable the child to understand that concurrency may quickly increase the complexity of the modelled system. Children should be guided to combine the two models in Fig. 4, The result should be something like the finite state machine in Fig. 5. Although children of this age should be able to develop this model, they are likely to feel that it is useless, due to the spaghetti-like interwinding of arrows. This is a good chance to show that, even with small systems as the ones in Fig. 4, concurrent



(a) Models of Claudio's and his friend's perspectives.



(b) Combined model of the two pespectives.

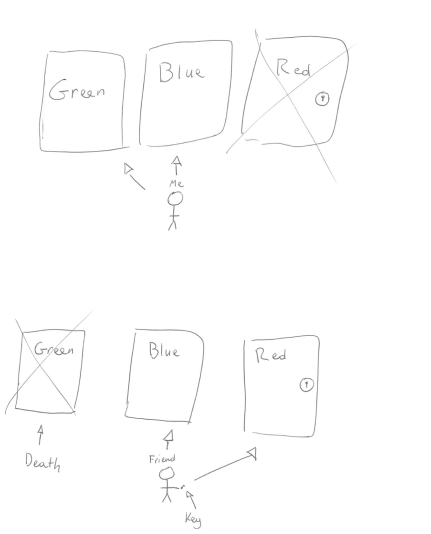


(c) Claudio's proof.

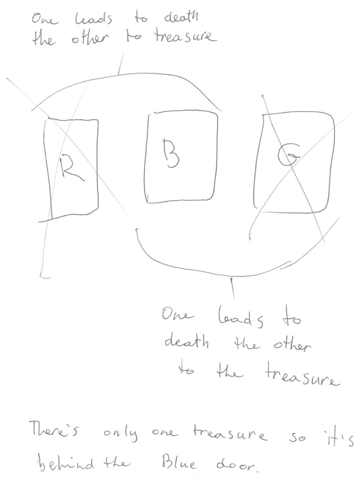
Fig. 2. Claudio's drawings.

composition may lead to a very complex global finite state machine. Depending on the interest expressed by the children, their previous knowledge and their reactions throughout the exercise, the discussion may now more deeply involve the notion of complexity. This may involve *algorithmic complexity* and/or the *state explosion problem*.

However, we must always avoid that the children become frustrated in unsuccessfully trying to develop the global finite state machine. If children are lost in the complexity of arrows, the teacher should help them to reach the solution in Fig. 5.



(a) Models of Chiara's and her friend's perspectives.


One leads to death the other to treasure
R
B
G
One leads to death the other to the treasure
There's only one treasure so it's behind the Blue door.

(b) Chiara's proof.

Fig. 3. Chiara's drawings.

4.2 Temporal Reasoning Using Finite State Machines

A number of observations and exercises may be carried out on the complex visual representation depicted in Fig. 5:

1. The model illustrates all possible temporal orderings in which the two friends arrive at the castle and choose the door.
2. Some children might note the fact that the model does not capture the case that the two friends arrive at the castle at the same time, with further questions arised
   (a) how to modify the finite state machine to cover this case?
   (b) how much the complexity would increase after such a modification?
   (c) is such a modification necessary? if so, why?
   and the chance to introduce and discuss the difference between *true concurrency* and *interleaving*.
3. The model can be enriched with further information, for example by colouring the states (the circles) in which a certain property is true. (Examples of properties are: you find the treasure, your friend finds the treasure, you die, your friend dies, or a combination of some of them using logical connectives 'or' or 'and'.)
4. Perform a temporal analysis on the model coloured as in Item 3 to find out whether, starting from the initial state, a property is true [3]
   (a) for some state (temporal modality: $\exists\Diamond$);
   (b) for all states (temporal modality: $\forall\Box$);

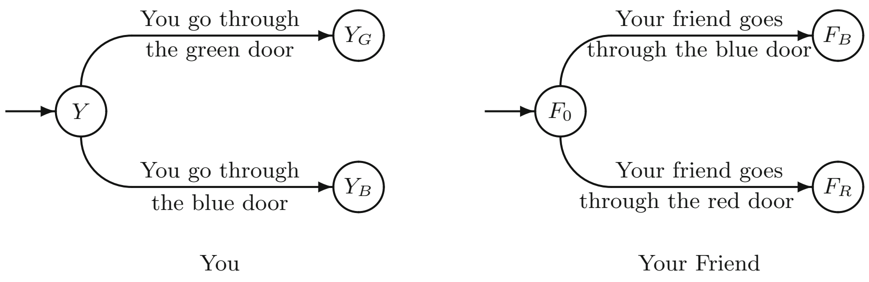

You go through the green door
Y_G
Y
You go through the blue door
Y_B
You
Your friend goes through the blue door
F_B
F_0
Your friend goes through the red door
F_R
Your Friend

Fig. 4. Age group 11–14: Story models of reasonable choices from your perspective and your friend's perspective using finite state machines.

(c) for all states along some path (temporal modality: $\exists\Box$);
(d) for some states along each path (temporal modality: $\forall\Diamond$).

If the discussion covers the difference between true concurrency and interleaving, it may be worthy to note that the model in Fig. 5 is based on interleaving.

Furthermore, as a result of the temporal reasoning carried out in Sect. 3, we can observe that the models in Figs. 4 and 5 do not carry any information about the story outcomes in terms of finding the treasure or die.

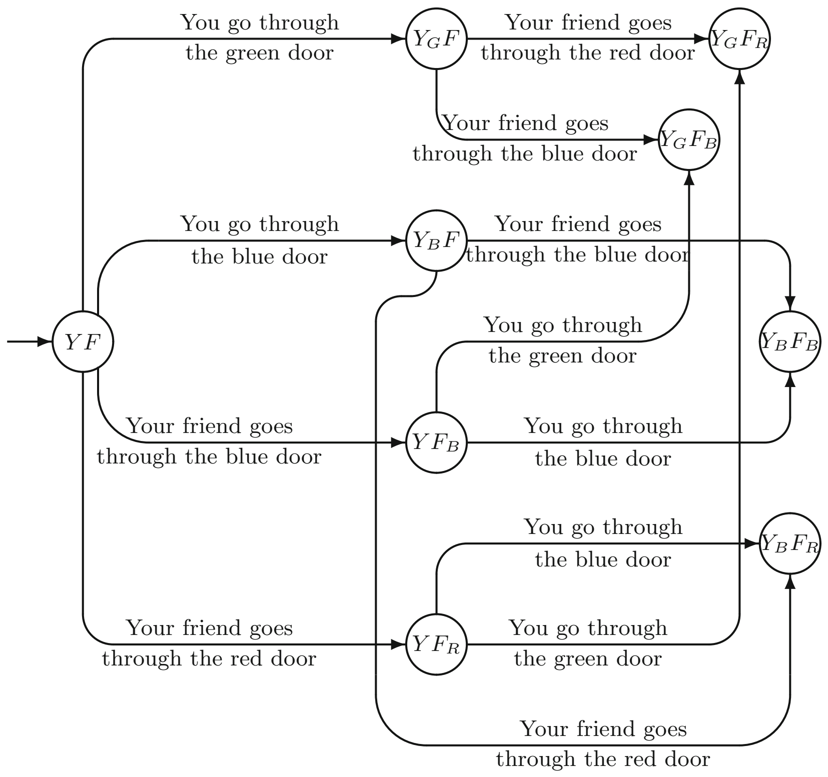
4.3 Refinement and Formal Verification

A next step for the children is to add final states to the models in Fig. 4 to describe the problem possible outcomes: you find the treasure (Y_T), you do not find the treasure and you do not die (Y_N), you die (Y_D), your friend finds the treasure (F_T), your friend does not find the treasure and does not die (F_N) and your friend dies (F_D). This is clearly a form of *model refinement.*

Here the issue is whether states Y_N and F_N are needed. After all, in our story, the parrot on the left side tells you that one of the two unlocked doors safely leads to the treasure and the other leads to death. Can we avoid death but not find the treasure? We have seen in Sect. 3 that alternatives 2.1 and 2.3 allow for this situation. However, the point to be made here is that this situation was observed only when we tried to compose the two stories. There is a double moral here.

On the one hand, refinement is not an easy task and it is likely to miss some essential behaviour while refining a model. In fact, there are normally many possible refinements, but in order to get a *correct refinement* we need to choose one of those that allow us to achieve our objective. Furthermore, it is important to choose the *best refinement* among all correct refinements. What is "best" is then a matter of *efficiency* and other *non-functional* system properties. But we are now going too far.

On the other hand, a missing *requirement* of a component can be identified when analysing the global behaviour through *formal verification.*



YF
You go through the green door
Y_GF
Your friend goes through the red door
Y_GF_R
Your friend goes through the blue door
Y_GF_B
You go through the blue door
Y_BF
Your friend goes through the blue door
Y_BF_B
You go through the green door
Your friend goes through the blue door
YF_B
You go through the blue door
You go through the blue door
Y_BF_R
Your friend goes through the red door
YF_R
You go through the green door
Your friend goes through the red door

Fig. 5. Age group 11–14: Global model of the two independent perspectives.

After this discussion we ask the children to compose the two new, extended finite state machines into a global one, observing that this can be carried out by just modifying the finite state machine in Fig. 5 through the addition of the appropriate composition of the final states. The spaghetti-like interwinding of arrows makes the model unreadable, but the children would still be able to build it. Trying to compose together the new, refined finite state machine components directly would actually be impossible, whereas adding the composition of the refined parts to the global machine is actually feasible. The moral here is that refinement makes the building of complex systems feasible.

The discussion considered in this section has not taken into account synchronisation yet. Once we reduce the number of reasonable choices with the additional conditions introduced in Sect. 3 by synchronising on the blue door through collaboration, then the age group 11–14 should come out with the use of a direct arc between state YF_R and state Y_BF_B, as shown in Fig. 6, to be added to the finite state machine in Fig. 5.

Both age groups can also work with the component models in Fig. 1 and come out with something like the representation in Fig. 7. If the discussion context is

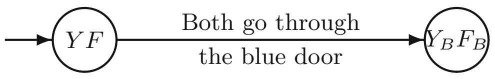


Fig. 6. Age group 11–14: Global model of the two synchronised perspectives.

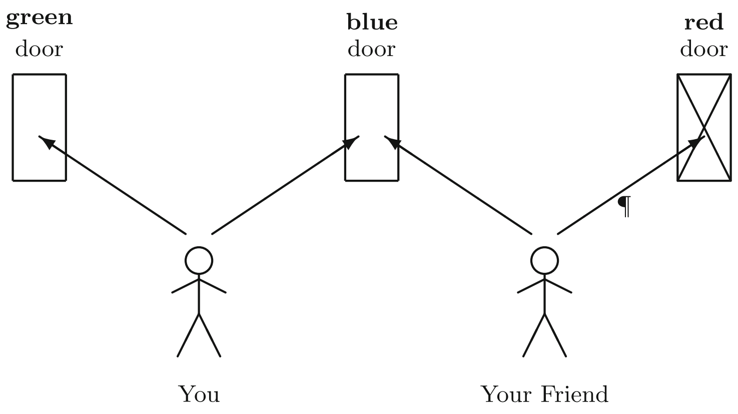


Fig. 7. Age group 7–10: Global model of the two synchronised perspectives.

appropriate, a final observation here may be that the model in Fig. 7 is based on true concurrency, in contrast to the finite state machine model in Fig. 5, which was based on interleaving.

It is important to note that when Claudio was asked to draw the global model, he realised that his models of the two separate perspectives, given in Fig. 2(a), were inadequate to be combined into a global model. In order to make his models *compositional*, he replaced the markers ✓ and × on the doors with arrows between persons and doors, thus getting the global model in Fig. 2(b), which is very similar with the expected model given in Fig. 7.

5 Representing Proofs

When Claudio was asked to show the solution of the problem he came up with the three marked doors in the top left corner of Fig. 2(c). A discussion followed to understand how he obtained such a solution. When asked to show the way he reached the solution he was initially puzzled. Then following the suggestion to use his previous models (drawings), combining them in some way and showing on them his reasoning, Claudio worked out the proof illustration in Fig. 2(c). It is interesting to note that some of the information abstracted in the models in Fig. 2(a) and 2(b), namely the padlock and the key, reappear in this representation of the proof.

A final note is that proofs developed by children of the age group 7–10 normally have purely visual representations, whereas children of the age group 11–14 can already articulate reasoning in a textual form. Their proof are likely to be in a hybrid visual and textual form as the one in Fig. 3(b), which was developed by Chiara.

6 Conclusion and Future Work

In this paper we have adopted an "unplugged approach" in teaching computer science to children [1,2] and taken inspiration from the *Choose Your Own Adventure* series of children's gamebooks, in which the reader may experience various alternatives about the characters' actions. We used the example of a story consisting of two parallel parts and made the children reason about the knowledge acquired by the protagonists of the two parts. Then we showed how to guide the children, on the one hand, to combine what the two protagonists know in order to predict the outcome of each decision and, on the other hand, to explore how the plot would evolve in the absence or in the presence of collaboration and information sharing between the two protagonists.

In this exploratory process, children have been exposed to real-life instantiations of a number of computer science concepts, especially from the theory of concurrency. Instances of concepts, such has synchronisation, assumption, starvation, complexity, state explosion, true concurrency, interleaving, refinement, correctness, efficiency, property and formal verification have been observed in the story plot. Such observations have been used to foster discussion and debate among the children, and enable reasoning, modelling as well as awareness and externalisation of their reasoning process throughout some form of written proof.

Three fundamental remarks are:

– The fact that we use a single example to explore a large variety of concepts through the paper is purely illustrative. In real classroom work a single, sequential or parallel story would probably be used to introduce one concept or a few strictly related concepts in a very targeted way. Obviously stories can also be revisited, expanded and compared at a later stage.
– Although for the benefit of the teacher, who might pursue a deeper understanding of the concepts underlying the observations, we have introduced technical computer science terminology, such a technical jargon should be avoided with the children, unless it is important for future topics or may appear curious or interesting for the children (e.g. the use of the word "starvation").
– The focus on concurrency has been used to show that an unplugged, problem-based approach can successfully work well beyond the most basic mathematical and computer science concepts, and cover one of the most challenging areas of computer science. It is by no means our intention to propose a children's course on "formal methods for concurrency" but, instead, to integrate the approach we presented within a more general unplugged, problem-based approach to be used in an interdisciplinary way.

We recall that the author experimented the approach presented in this paper with his own children, Claudio and Chiara, respectively 9 years old and 13 years old at the moment when the paper was written. Some aspects of the approach, especially the development of problem-solving skills, however, have been used with both children since they were 5–6 years old. Obviously, due to the lack of a "neutral" relationship between the children and researcher in this study, we cannot advance any claim on the validity of the approach we presented. Therefore this work has to be intended as a position paper proposing a methodology which still requires validation.

As future work we plan to experiment our proposal as part of a general unplugged, problem-based approach in a real classroom context thus providing reasonably sized, unbiased case studies.

Acknowledgments. Many thanks to my children Claudio and Chiara who represented the age groups 7–10 and 11–14, respectively, in carrying out the activities described in this paper. A special thank to my friend and colleague Karl Lermer whose conversations with his talkative parrots and with myself inspired the children story presented in this paper. Karl also hosted me in Zurich during my visit at ZAHW, when I was writing this paper.

References

1. Computer science without a computer. https://www.csunplugged.org/en/
2. Bell, T.: A low-cost high-impact computer science show for family audiences. In: 23rd Australian Computer Science Conference, pp. 10–16. ACM (2000)
3. Ben-Ari, M., Pnueli, Z.M.A.: The temporal logic of branching time. In: POPL 1981, pp. 164–176. ACM (1981)
4. Brainerd, C.J.: Piaget's Theory of Intelligence. Prentice Hall, Englewood Cliffs (1978)
5. Bruner, J.S.: Toward a Theory of Instruction. Bwelknap Press, Cambridge (1966)
6. Gibson, J.P.: Formal methods: never too young to start. In: FORMED 2008, Budapest, Hungary, pp. 151–160 (2008)
7. Hilton, P.: The mathematical component of a good education. In: Miscellanea Mathematica, pp. 145–154 (1991)
8. Hopcroft, J.E., Ullman, J.D.: Introduction to Automata Theory, Languages, and Computation. Addison-Wesley, Reading (1979)
9. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. Commun. ACM **21**(7), 558–565 (1978)
10. Lamport, L.: Turing lecture: "The computer science of concurrency: The early years". Commun. ACM **58**(6), 71–76 (2015)
11. Magee, J., Kramer, J.: Concurrency: State Models and Java Programs. Wiley, New York (2006)
12. Murphy Paul, A.: Researchers Find That Frequent Tests Can Boost Learning. Scientific American (2015)
13. Packard, E.: Sugarcane Island. Vermont Crossroads Press (1976)
14. Schoenfeld, A.H.: Mathematical Problem Solving. Academic Press, New York (1985)
15. Wang , J., Tepfenhart, W.: Formal Methods in Computer Science. Chapman and Hall/CRC, Boca Raton (2019)

When the Student Becomes the Teacher

Marie Farrell[1](✉) and Hao Wu[2]

[1] Department of Computer Science, University of Liverpool, Liverpool, UK
marie.farrell@liverpool.ac.uk

[2] Department of Computer Science, Maynooth University, Maynooth, Ireland

Abstract. Making formal methods accessible and appealing to future software engineers is vital to promote their uptake in industry and to increase participation in formal methods research. In this paper, we report on our initial experience of both studying and, subsequently, teaching the same software verification module at Maynooth University, Ireland. By analysing on our own teaching and learning experiences along with the students' grades from the 2018–2019 academic year, we present our four initial observations and two hypotheses that we intend to investigate during the 2019–2020 academic year.

1 Introduction

Encouraging students to take an interest in formal methods has generally been perceived as a difficult task [4,5,8–10,18]. Though there has been some success in convincing software developers to use formal methods, it is still quite challenging [20]. In order to increase the uptake of formal methods in industry, we believe that we must first begin by convincing our students that formal methods are useful and relevant for industrial use [4,10,13,17].

In this short paper, we report on our experiences of both studying and teaching the Software Verification module at Maynooth University. We provide some analysis and discussion which we use as a basis for identifying ways to improve this module and to capture the students' interests.

We summarise our contributions as follows:

1. We report on our experience of studying, during our time as undergraduates, and subsequently, teaching a formal methods module to undergraduate students at Maynooth University. To this end, we analyse and discuss this module in light of the associated exam results from the 2018–2019 academic year.
2. We present our observations and form two hypotheses to be further investigated to improve both the teaching and learning experience for this module.

The remainder of this paper is structured as follows. In Sect. 2, we provide an overview of the formal verification module that we both studied and taught

This work is partially supported through EPSRC Hubs for Robotics and AI in Hazardous Environments: EP/R026092 (FAIR-SPACE).

at Maynooth University. We describe the assessment process for this module in Sect. 3 where we briefly analyse the exam results from the 2018–2019 academic year. In Sect. 4, we reflect upon our own experience both as students and as teachers of this module. We make four observations about the module's current status by combining our reflection with the analysis of the exam results described in Sect. 3. Based on these observations, we form two hypotheses in Sect. 4.3 to be investigated during the current (2019–2020) academic year. Finally, Sect. 5 concludes and outlines future research directions.

2 Module Overview

The Software Verification module (CS357) at Maynooth University aims to provide students with an understanding of both the theoretical and practical applications of formal software verification techniques[1]. The majority of the students taking this module are third-year (Bachelor's degree) students studying Computer Science. For these students, and those studying for the Computational Thinking Bachelor's degree (usually approx. 10 students), this module is compulsory. This module is optional for those studying General Science where Computer Science is a chosen subject. This module assumes that the students have already taken modules in basic Java programming and discrete mathematics (or equivalent).

This module runs over 12 weeks (2 lecture hours and 2 laboratory hours per week) and covers a wide range of different topics. The topics that are covered and the duration spent on each is outlined below:

1. Design by Contract (1 week) [15]
2. Natural Deduction Proofs and the Coq theorem prover (3 weeks) [3]
3. Hoare Logic (2 weeks) [12]
4. Spec# (2 weeks) [1]
5. SAT/SMT (2 weeks) [7]
6. Model Checking (2 weeks) [12]

Upon successful completion of this module, the students should be able to:

1. Explain the role of verification in software engineering.
2. Create mathematically precise specifications.
3. Prove the correctness of programs using Hoare logic.
4. Use different tools to analyse and verify properties of specifications.

These four learning objectives are reflected in the exam structure and continuous assessment (CA) that we describe in the next section.

[1] Full module description is available at: http://apps.maynoothuniversity.ie/courses/?TARGET=MODULE&MODE=VIEW&MODULE_CODE=CS357&YEAR=2020.

Table 1. There are four questions on the exam and each reflects different aspects of the module as outlined in Sect. 2. These questions are designed to assess the learning objectives described in Sect. 2.

Question	Examined topics	Weight (marks)
Q1	Design by contract Propositional and predicate logic Natural deduction proofs	25
Q2	Satisfiability CNF Translation DPLL (Pure literal, Unit clause and Unit propagation)	25
Q3	Hoare Logic	25
Q4	Basic SMT encoding Spec# Programming (Pre/Post conditions, Loop invariants) Linear temporal logic encoding	25

3 Assessment

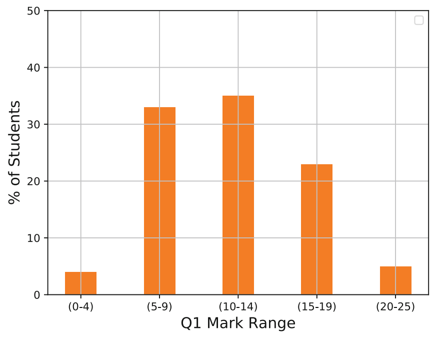
In this section, we describe how this module is assessed. In particular, each student's final grade consists of 30% for continuous assessment (CA) with 70% for the final examination. To obtain CA, each student is required to attend one 2-hour laboratory session every week in order to complete their weekly assignment and to get it graded by one of the tutors. These assignments, 11 in total, are based on the material covered during the lectures each week. The first 3 assignments focus on assessing basic understanding of natural deduction proofs using the Coq theorem prover. The next 2 assignments are based on Hoare Logic. The remaining ones examine a range of verification tools such as Spec# and Z3. At the end of term, the students must complete their final exam.

3.1 Exam Structure

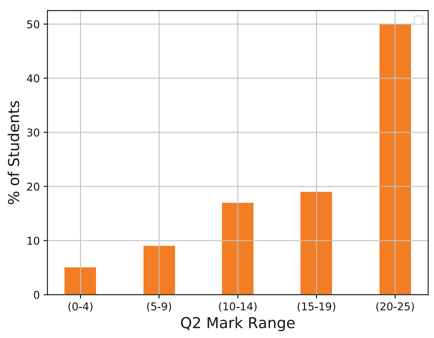
The final exam is 2-h long and pen & paper based. For full marks, students must correctly answer three out of four questions on the paper. In the case that a student answers all four questions, the best three are combined for their final grade. The overall exam structure is outlined in Table 1. Each question is weighted equally (25 marks) and focuses on examining a different topic. For example, Q2 in Table 1 is designed to examine the basic knowledge of two algorithms: Tseitin transformation [19] and DPLL [6], while Q3 is designed to examine Hoare Logic [11].

3.2 Exam Results

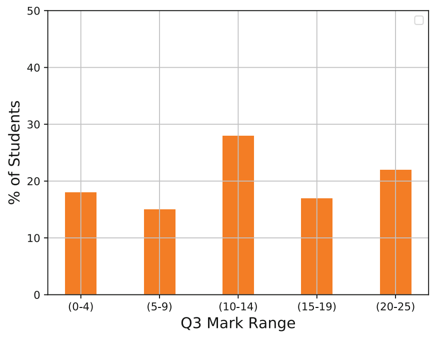
Overall, a total of 92 students, during the 2018–2019 academic year, participated in the module. In total, 23 of them failed resulting in a 25% failure rate. We



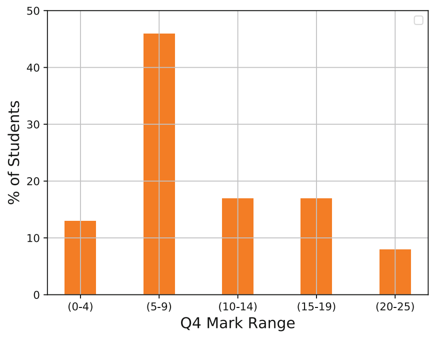
(a) Mark distribution for Question 1.



(b) Mark distribution for Question 2.



(c) Mark distribution for Question 3.



(d) Mark distribution for Question 4.

Fig. 1. Marks distribution for four questions in Table 1.

analyse the exam results on a per question basis as illustrated in Fig. 1. For each of these graphs, we plot the mark range (x-axis) against the percentage of students that answered this particular question in the exam (y-axis).

We can see from Fig. 1 that the students performed the best on Q2. In fact, Q2 was the most popular question with 88 out of 92 students attempting it. We believe that this is due to the mechanical nature of Q2. Once a student masters applying the corresponding rules, he/she is able to solve the basic problems on the fly. Hence, Q2 was the most popular and highest scoring out of the four exam questions.

Conversely, Q4 was the least popular. Only 24 out of 92 students attempted this question. Q4 was designed to challenge students on the following topics:

– Encode a simple specification into SMT formulas.
– Write a Spec# program with the appropriate specifications corresponding to a simple C# function that computes the sum of an integer array.
– Show basic SAT-encodings for reachability, safety and liveness properties.

We believe that this question was the least popular because it requires students to understand low-level SAT/SMT encoding plus writing specifications for a piece of code. In general, this type of question was not appealing to the students and this was also observed by the tutors during the weekly laboratory assignments.

4 Reflecting on Teaching and Learning

In this section, we outline our own experience as both student and educator. We reflect on this experience in light of the above examination results and outline our observations. Based on these observations we develop two hypotheses in relation to making this module more accessible and enjoyable from the students' perspective.

4.1 Our Experience

We summarise our own experience of learning (from our time as students studying this module) and our experience of teaching this module below. We note that the content of this module has not changed significantly in the time that has passed since we originally studied it during our undergraduate degrees.

Learning:

- The content of this module is generally challenging for students. Particularly, identifying loop invariants, presenting Hoare logic proofs and understanding low-level SAT/SMT encodings.
- Verification tools in general are not very reliable and the online versions of the tools frequently stopped responding during the lab sessions. Furthermore, the feedback from the tools is normally not very helpful in terms of figuring out what to prove or where one went wrong in the specification.
- Practical applications of formal verification are not very clear to the students.

Teaching:

- It is very difficult to help the students to see the value in the module since lots of these techniques are not widely used in industry.
- Many tools are not scalable for real-world examples and this makes it difficult to demonstrate their usefulness to students.
- It is necessary to use a combination of slides and manually working through examples on the whiteboard to explain the detailed computation steps (e.g. Hoare Logic) to the students.

We studied this module ourselves some years ago as students, and upon reflection, we believe that the content of this module has always been quite challenging. Our experience of teaching this module (both as lab tutor and lecturer), interacting with the students and students' feedback forms[2] have revealed that this perception has not changed. Therefore, to encourage students, it is necessary to make improvements to this module. In order to identify such improvements we first outline four observations and then form two hypotheses in the next subsections.

4.2 Observations

By combining the examination results (Sect. 3) and our own experiences above, we make the following four observations:

Observation 1. Automated verification tools are not appealing to the students even though we described real world disasters that could have been avoided by using formal methods [14]. The tools that we use in this module for automated reasoning are Spec# [1] and Z3 [7]. The students use the online versions of Spec#[3] and Z3[4] as part of their practical lab work. In the beginning, the students found the click-button and go style interesting. However, they subsequently discovered that the feedback from the tool was not usually helpful for fixing bugs in the source code. We believe that this makes the tools harder to use and causes the students to lose interest. This also explains the reason for so few students, 24 out of 92, attempting Q4 as described in Table 1.

Observation 2. Most of the students performed reasonably well on natural deduction proofs but not using the Coq interactive theorem prover [3]. For natural deduction proofs, most of the students have already studied the material from their discrete structures module. However, they feel that it is difficult to find connections between the proofs worked out on a piece of paper and the corresponding sequence of Coq commands although multiple live Coq proof sessions are given throughout the lectures.

Observation 3. The verification tools and techniques that we have developed during our research were not integrated into this module. As such, the students were not given the opportunity to learn about our work. A portion of our research focuses on using SAT/SMT solving techniques to tackle problems from software engineering domains which is particularly relevant for this module [21,22]. Unfortunately, we never had the chance to present our work due to time constraints.

[2] At the end of the semester, the university distributes feedback forms for the students to fill in for each module that they have taken. It is not compulsory for the students to complete them and they ask broad, non module specific questions. We received a small number of responses and have used these to inform our discussion but we rely more heavily on the exam results and our interactions with the students.

[3] https://rise4fun.com/SpecSharp.

[4] https://rise4fun.com/Z3.

Observation 4. In general, the students had mixed reactions to the Hoare logic part of the module. In particular, the worked out whiteboard examples showing how to discover loop invariants were difficult to digest for some students. This usually involved interactive sessions during the lectures where the lecturer and students worked together to solve the problems. Others found the whiteboard examples to be extremely helpful when studying Hoare Logic. These students typically had a strong background in mathematics. We speculate that these students were accustomed to whiteboard style teaching whereas pure computer science students were more likely familiar with electronic slides.

These four observations reveal a number of shortcomings for this module. In particular, Observations 1 and 2 point to a lack of tool usability. This is a challenge for the formal methods community at large and can also hinder the uptake of formal methods in industry. Observation 4 noted that the students generally found Hoare logic difficult to grasp but this may also be exacerbated by the students' difficulty in using tools such as Spec#.

As a result of Observation 3, we have already started to integrate our research tool into current teaching. For example, we introduced our own tool, MaxUSE [21,22][5], into one of the classes and showed the students how to use it to find conflicting class invariants for a UML class diagram. A number of students clearly showed interest in the tool and would like to know more about its underlying algorithms and theories.

Based on these observations, we derive two hypotheses for improving this module in the next subsection.

4.3 Hypotheses

In this subsection, we develop two hypotheses that are based on the observations derived in the previous subsection. We intend to use these hypotheses to guide future improvements to be made to this module that we plan to investigate during the current (2019–2020) academic year.

Hypothesis 1. The development of an online repository that contains a collection of real world examples would be useful for both teaching and illustrating industrial uses of formal methods to the students. These examples could be proved by either using automated or interactive verification tools such as Z3 and Coq. We believe that this would create a strong connection between the theory taught in the class and practical, real world applications. However, the challenge here is that the examples collected or manually created should be small, but detailed enough to be suitable for educational use. One way to begin is to design and distribute a survey among the past students in order to identify the most interesting and educational examples to be used in the class.

[5] https://github.com/classicwuhao/maxuse.

Hypothesis 2. A platform such as Tarski's world that turns different kinds of logical reasoning proofs into games would increase the interactions between lecturers and students [2]. Hence, we believe that this is a good way to attract students to formal methods. For example, a live coding session that works with the students using SMT solvers to solve a Sudoku puzzle would be much more enjoyable and interesting than simply elaborating on different SMT constructs in the slides. However, there are two primary challenges that arise from this: (1) it may not be possible for each student to bring a laptop to the lecture, and, (2) students who miss the pre-setup steps may break the pace of a lecture. One potential solution is for the lecturer to show the code (solving games) running on their own machine and to upload the source code after the lecture so that the students can try it in their own time. However, in this way the interactions between the lecturers and students might be significantly reduced.

We have derived these hypotheses from our own experiences and the observations that we have made. We intend to investigate these hypotheses as future work to see if they improve the student experience and the exam results.

5 Conclusions and Future Work

Teaching formal methods is quite challenging and making formal methods appealing to younger generations is very important for continuously expanding the formal methods community in both industry and academia. In this paper, we discuss our own experience of both studying and teaching the same software verification module at Maynooth University. Based on our experiences and analysis of the exam results from the 2018–2019 academic year, we have derived four key observations in Sect. 4.2, from which we construct two hypotheses (Sect. 4.3) that we will investigate during the current (2019–2020) academic year.

Furthermore, we plan to work with the education research group within the department (at Maynooth University) to design interesting experiments in order to figure out the best way of teaching formal methods and to let students have fun with it. These experiments include interviewing students about specific topics covered during the lectures, gathering and analysing real feedback from the current academic year and using game based strategies to teach students to use different verification tools [16]. We believe that these experiments can help us to encourage the students to use formal methods/verification tools in their careers after their university studies.

Since Maynooth University also offers a similar module at Master's level, we plan to investigate the corresponding exam results and compare them with those presented in this paper. Furthermore, a much more detailed student feedback form will be distributed at the end of the module for further analysis.

References

1. Barnett, M., Leino, K.R.M., Schulte, W.: The spec# programming system: an overview. In: Barthe, G., Burdy, L., Huisman, M., Lanet, J.-L., Muntean, T. (eds.) CASSIS 2004. LNCS, vol. 3362, pp. 49–69. Springer, Heidelberg (2005). https://doi.org/10.1007/978-3-540-30569-9_3
2. Barwise, J., Etchemendy, J.: Tarski's World: Version 4.0 for Macintosh (Center for the Study of Language and Information - Lecture Notes). Center for the Study of Language and Information/SRI (1993)
3. Bertot, Y., Castran, P.: Interactive Theorem Proving and Program Development: Coq'Art The Calculus of Inductive Constructions, 1st edn. Springer (2010)
4. Cataño, N.: Teaching formal methods: lessons learnt from using event-B. In: Dongol, B., Petre, L., Smith, G. (eds.) FMTea 2019. LNCS, vol. 11758, pp. 212–227. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-32441-4_14
5. Creuse, L., Dross, C., Garion, C., Hugues, J., Huguet, J.: Teaching deductive verification through Frama-C and SPARK for non computer scientists. In: Dongol, B., Petre, L., Smith, G. (eds.) FMTea 2019. LNCS, vol. 11758, pp. 23–36. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-32441-4_2
6. Davis, M., Logemann, G., Loveland, D.: A machine program for theorem-proving. Commun. ACM **5**(7), 394–397 (1962)
7. de Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_24
8. Dean, C.N., Hinchey, M.G.: Teaching and Learning Formal Methods. Morgan Kaufmann, San Francisco (1996)
9. Gallardo, M.M., Panizo, L.: Teaching formal methods: from software in the small to software in the large. In: Dongol, B., Petre, L., Smith, G. (eds.) FMTea 2019. LNCS, vol. 11758, pp. 97–110. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-32441-4_7
10. Gibson, J.P., Méry, D.: Teaching formal methods: lessons to learn. In: IWFM. Citeseer (1998)
11. Hoare, C.A.R.: An axiomatic basis for computer programming. Commun. ACM **12**(10), 576–580 (1969)
12. Huth, M., Ryan, M.: Logic in Computer Science: Modelling and Reasoning About Systems. Cambridge University Press (2004)
13. Jaume, M., Laurent, T.: Teaching formal methods and discrete mathematics. In: 1st Workshop on Formal Integrated Development Environment, vol. 149, pp. 30–43. EPTCS (2014)
14. Jazequel, J., Meyer, B.: Design by contract: the lessons of Ariane. Computer **30**(1), 129–130 (1997)
15. Meyer, B.: Object-Oriented Software Construction, 1st edn. Prentice-Hall (1988)
16. Moller, F., O'Reilly, L.: Teaching discrete mathematics to computer science students. In: Dongol, B., Petre, L., Smith, G. (eds.) FMTea 2019. LNCS, vol. 11758, pp. 150–164. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-32441-4_10
17. Oliveira, J.N.: A survey of formal methods courses in European higher education. In: Dean, C.N., Boute, R.T. (eds.) TFM 2004. LNCS, vol. 3294, pp. 235–248. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-30472-2_16
18. Rozier, K.Y.: On teaching applied formal methods in aerospace engineering. In: Dongol, B., Petre, L., Smith, G. (eds.) FMTea 2019. LNCS, vol. 11758, pp. 111–131. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-32441-4_8

19. Tseitin, G.S.: On the complexity of derivation in propositional calculus. Stud. Math. Math. Logic **2**, 115–125 (1968)
20. Woodcock, J., Larsen, P.G., Bicarregui, J., Fitzgerald, J.: Formal methods: practice and experience. ACM Comput. Surveys (CSUR) **41**(4), 19 (2009)
21. Wu, H.: Finding achievable features and constraint conflicts for inconsistent metamodels. In: Anjorin, A., Espinoza, H. (eds.) ECMFA 2017. LNCS, vol. 10376, pp. 179–196. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-61482-3_11
22. Wu, H.: MaxUSE: a tool for finding achievable constraints and conflicts for inconsistent UML class diagrams. In: Polikarpova, N., Schneider, S. (eds.) IFM 2017. LNCS, vol. 10510, pp. 348–356. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-66845-1_23

Teaching Formal Methods in Academia: A Systematic Literature Review

Rustam Zhumagambetov(✉)

Nazarbayev University, Nur-Sultan 010000, Kazakhstan
rustam.zhumagambetov@nu.edu.kz

Abstract. The amount of literature on teaching formal methods has been growing. However, there is a lack of attempts to systematically review existing practices. This study attempts to identify challenges related to teaching formal methods by examining the literature on this topic. The literature review categorizes and systematizes the existing experience of teaching formal methods to students at universities. It presents obstacles reported as well as the strategies to deal with them that are expected to help current practitioners.

Keywords: Formal methods · Education · Programming · Teaching

1 Introduction

Current practices of developing software include its testing. However, from a formal point of view, it is not feasible to cover every possible piece of software functionality with tests. Especially this concern relates to safety or mission critical software, such as software for medical machines and satellites.

Formal methods in software engineering address this problem by utilizing mathematical frameworks of reasoning about programs, which is hard to understand for the majority of students. Courses that involve formal methods become the most unpopular.

To the best knowledge of the author, there exists no systematic literature review that addresses challenges of teaching formal methods. One of the notable works that addresses challenges of teaching formal methods is the survey by Spichkova and Zamansky [24]. However, this survey is not comprehensive as it lacks the papers from the conferences on teaching formal methods. Thus, there is a need for systematic synthesis of the literature on the practices of teaching formal methods.

Despite the importance of formal methods for testing and verification of critical software, universities fail in teaching this aspect of computer science to students. This systematic literature review will categorize and systematize the existing experience of teaching formal methods to students at the university level.

2 Review Questions

To carry out this systematic literature review a number of activities, including database search, quality assessment as well as data extraction and synthesis, were conducted to formulate a conclusion.

The objective of this study is to identify the struggles that educators encounter during teaching formal methods in the higher education. A particular interest is strategies that used to overcome the struggles.

So, to achieve the objective the following research questions were formulated:

– *RQ1.* What are the challenges of teaching formal methods in the universities?
– *RQ2.* What are the strategies used to deal with these challenges?

3 Review Methods

According to Kitchenham and Charters [11], the following review components should be present to establish replicability of the review:

– Study search strategy
– Study selection criteria
– Study quality assessment
– Data extraction strategy

3.1 Study Search Strategy

Based on the objective of the paper, the following search query was used for the search: "teaching" AND "formal methods". The terms were matched to titles, abstracts and keywords.

The following databases were used for search:

– IEEEXplore (https://ieeexplore.ieee.org/)
– ACM Digital library (https://dl.acm.org/)
– Elsevier Science Direct (https://www.sciencedirect.com/)

The author has decided to include papers from conference proceedings. Proceedings of the following conferences were included:

– TFM 2009
– FMET 2008
– TFM 2006
– FMED 2006

As can be seen in Fig. 1, the primary search have found 151 studies.

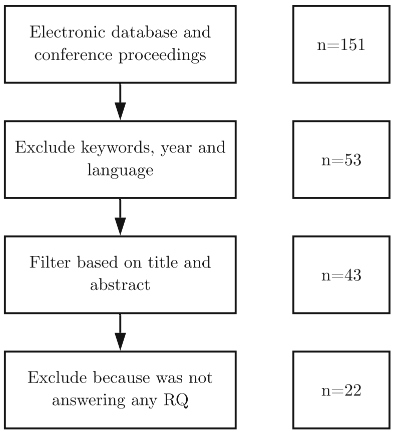

Electronic database and conference proceedings
n=151
Exclude keywords, year and language
n=53
Filter based on title and abstract
n=43
Exclude because was not answering any RQ
n=22

Fig. 1. A flowchart that represents research protocol

3.2 Study Selection Criteria

After the initial review of the queries, some keywords were identified as irrelevant to the study. Here is the list of excluded keywords: statistic, statistics, Grammar, Second Language Learning, formal methods of estimation, Economics education, Law students, Physical education, Physicians, Emergency Medicine Education. Other criteria for selection are papers should be published not later than 2001 and be written in English language. Number of studies at this stage is 53.

After keywords exclusion, all papers were reviewed by the author to perform third stage of selection. During this stage we filtered papers based on their relevance to the research questions.

Then, papers were filtered based on title and abstract. During this filtering process studies that claimed to teach formal methods outside of universities, presented formal methods in other contexts were excluded. For example, title of [25] states about experience in middle school and [7] about formal methods in electronic government. At this stage the number of papers was reduced to 43.

3.3 Study Quality Assessment

In this stage rest of the papers were read and assessed on how they answer at least one of the research questions. At this stage, primary targets for filtering out were papers that have not neither evidence nor opinion regarding how challenges of teaching formal methods or what strategies were used improve experience of the students that participate in the course. Examples of the filtered out papers are [18] and [15]. The former paper mostly consists of anecdotes that are aimed to motivate anyone, who reads them, to use formal methods. The latter paper is the one, which focused more on the tool, rather then how the tool solves the problem of students. The final number of papers used to answer research questions is 22.

3.4 Data Extraction Strategy

The resulting set of the papers is a mixture of teaching experience that was recorded either through a session or within a certain period, when course was offered. Other papers demonstrate the use of software or teaching approaches to solve general or a particular problem along with student's feedback.

Due to the nature of the "raw" material, structured data extraction becomes either very difficult or impossible to conduct in the objective manner. So the author used a word processor to identify some of the themes presented in the studies, grouping them as necessary.

4 Results

In this section we report findings about research questions. The purpose is to provide themes discovered in the studies.

4.1 What Are the Challenges of Teaching Formal Methods?

We found 17 studies that identified 3 main themes that answer this question: students scepticism, difficulties with auxiliary software and challenges with materials.

Students' Scepticism. This is one of the biggest challenges identified by majority of researchers. Throughout the literature there were two prominent problems that are related: student's scepticism about formal methods as well as isolation of formal methods from application in other courses.

Students Are Skeptical About the Usefulness of Formal Methods. This is the major problem identified in 8 studies [2–4,6,9,17,19,26]. Blanco et al. [2] discusses how program verification is viewed by students as a "an additional burden". Brakman et al. [4] perceives that students see little practicality in use of formal methods.

Isolation of Formal Methods from Application in Other Courses. Boute [3] observes often such behavior is motivated by little exposure to formal methods across curriculum. Supporting, Sekerinski [22], reporting his experience of teaching a course on software design with formal methods, identifies an isolated use of formal methods as a main obstacle that prevents its flourishing.

Difficulties with Auxiliary Software. Another theme that was identified throughout review is difficulties with tools that are used to teach formal methods. Steep learning curve and bad user experience are major challenges that support the theme.

Steep Learning Curve for Tools. Lipaczewski and Ortmeier [13] recognized a steep learning curve as one of the failures of tools that use formal methods. Schreiner [21] supports them adding that teaching time is too scarce to consider teaching how to use this "difficult to learn and/or inconvenient to use" software.

Tools Give Little Feedback. In their course Bayley, Lightfoot and Martin [1] avoid using software tools, because tools can "produce a fairly cryptic and discouraging response". Gopalakrishnan [9] observed that most tools lack a good user experience and failing do not provide insight at why they fail.

Challenges with Course Materials. The third theme that was common to a number of studies is challenges with course materials. Such challenges include dryness of the course, diversity of materials, lack of comprehensive textbook and course require students to have better mathematical knowledge than they have.

Dryness of the Course. Two studies, [4] and [19] identified dryness of the course as a problem of teaching formal methods. In particular, Brakman et al. [4] reports that "lack of visualization" is the primary reason of ineffectiveness of teaching formal methods.

Diversity of Materials. Gopalakrishnan [9] notes that there are a variety of tools that use formal methods; however, there are few initiatives to adopt them. These tools "face the uphill battle of vying for attention and commitment from teachers" [9]. In other word, there are plenty of resources on formal methods that are yet to be adopted by educators.

No Comprehensive Textbook. While there exists a diversity of materials, some authors find it difficult to navigate through them. In particular, Kofroň, Parízek and Šerý [12] report that they were not satisfied with comprehensiveness of one of the textbooks used for one of their courses.

Inadequate Background in Mathematics. Several authors [8,14,16,23] report students that take courses in formal methods often do not know a certain concepts that are used in the course. Consequently, such inadequacy requires inclusion of additional materials into the course.

4.2 What Are the Strategies Used to Deal with These Challenges?

We have found 15 papers and 4 major themes that answer this question that are formulated as a form of advice: practice, engage students, avoid isolation of formal methods and simplify material.

Practice. This advice was formulated because several authors attributed the success of the formal methods courses they ran to the extensive practice. They provided a large number of examples to the students to facilitate the ingestion of theoretical and practical components of formal methods.

Use a Lot of Examples. Aceto et al. [6] report that they avoid presenting general theory, preferring investigation of particular instances. They declare that examples play "primary role" in their practice.

Real-life Examples and Projects. Some practitioners [4,5,20] make stronger claims by saying that examples should involve "real-life", not artificial instances of the problem. For instance, Brakman et al. [4] discusses how presenting verification of Bluetooth protocol as a course project helped their students to increase understanding of formal methods.

Engage Students. This strategy was devised to describe several reported practices that increased engagement and motivation of the students with formal methods.

Gamify. Prasetya et al. [19] presents a game, FormalZ, that presents concepts of formal methods in fun and engaging manner. It also introduces a competitive environment by introducing a leaderboard, a ranking list. Their preliminary results show positive feedback from students.

Use Electronic Voting systems. Miller and Cutts [16] introduce an electronic voting system (EVS) to help students to actively participate. EVS often consists of a several buttons that represent answer choices and a display that presents a question. Often EVS are anonymous and are instruments to increase student engagement rather than test of the knowledge. Miller and Cutts [16] report use of EVS increased confidence of student to "to answer questions within the class".

Avoid Isolation of Formal Methods. This direction came from several studies that report an underrepresentation of the formal methods across curricula. Such situation leads to formal methods looking esoteric, decreasing student's motivation [3,22].

Start Early in Introductory Course. Blanco et al. [2] reports that the course on introductory programming that was based on formal methods was helpful for students to learn appreciation of formal methods. They report that the course had lasting effect based on "the optional courses they choose later on in the programmes, and the topics they choose for their final projects" [2].

Use a Combination of Formal Techniques. Some educators claim that they achieve best results when they explain several formal techniques. Hallerstede and Leuschel [10] utilize combination of "formal proof, model-checking, and animation" to improve accuracy of a formal model.

Juxtapose of Taught Formal Methods with Other Methodologies. To overcome students' scepticism about formal methods Catano and Rueda [6] ask students throughout the course to reflect upon pros and cons of the formal methods compared to the already learned traditional approaches.

Simplify Material. This strategy is intended to simplify not formal methods, but the material used to deliver the lesson. While it is difficult, or impossible to make formal methods accessible for everyone, some educators have found ways of making their delivery friendlier to the listeners.

Use Another, Simpler Tool. Some studies [13,21] suggest that to overcome challenges associated with software, which uses formal methods, it is the best to use other, simpler tools. For example, Lipaczewski and Ortmeier [13] propose SAML with its plugins for IDE and web-based user interface. SAML is intended as tool for building models that are more friendly to the user than existing alternatives.

Use Common Languages that Have Built-in Expressivity. Another study avoids specialized formal methods software and languages. Instead, Gopalakrishnan [9] uses a general purpose language, like Python, for "Models of Computation" course. It has features, like list comprehension that helps writing definitions with mathematical notation.

Use Less Formal Testing as a Medium for Teaching Some of the Formal Methods. Utting and Reeves [26] suggest that interweaving of traditional testing approach and elements of formal methods helped their students to have a better satisfaction with using formal methods.

5 Discussion

This systematic review have demonstrated that there is a number of challenges, like students' scepticism, difficulties with computer-aided assistants and challenges with the course materials. There are a few practices that can help to overcome these challenges, like use of examples, gamification, teaching of combination of formal methods and simplification of the delivery. The author believes that some of the discovered challenges are valid and hope that provided strategies will help to overcome them.

5.1 Limitations

Even though this literature review attempts to be objective, there is no ideal, objective criteria that would quantify success or failure of the provided solutions. Most of the researchers relied on the students feedback, course enrollment number or their own senses to draw a conclusion about usefulness of certain techniques.

6 Conclusion

A systematic literature review was performed on 22 papers that report practices of teaching formal methods in universities. The aim of this review was to identify the struggles that educators encounter while teaching formal methods, as well as exploring strategies that are used to deal with these struggles. The findings allowed to formulate 8 themes of challenges that were further grouped into 3. We have found 11 themes of strategies that were grouped into 4.

References

1. Bayley, I., Lightfoot, D., Martin, C.: Teaching the Oxford Brookes formal specification module. In: Teaching Formal Methods, p. 5 (2006)
2. Blanco, J., Losano, L., Aguirre, N., Novaira, M.M., Permigiani, S., Scilingo, G.: An introductory course on programming based on formal specification and program calculation. SIGCSE Bull. **41**(2), 31–37 (2009). https://doi.org/10.1145/1595453.1595459
3. Boute, R.: Teaching and practicing computer science at the university level. SIGCSE Bull. **41**(2), 24–30 (2009). https://doi.org/10.1145/1595453.1595458
4. Brakman, H., Driessen, V., Kavuma, J., Bijvank, L.N., Vermolen, S.: Supporting formal method teaching with real-life protocols. In: Formal Methods in the Teaching Lab: Examples, Cases, Assignments and Projects Enhancing Formal Methods Education, pp. 59–68 (2006)
5. Catano, N.: An empirical study on teaching formal methods to millennials. In: 2017 IEEE/ACM 1st International Workshop on Software Engineering Curricula for Millennials (SECM), pp. 3–8. IEEE, Buenos Aires, Argentina (2017). https://doi.org/10.1109/SECM.2017.1
6. Catano, N., Rueda, C.: Teaching formal methods for the unconquered territory. In: Gibbons, J., Oliveira, J.N. (eds.) TFM 2009. LNCS, vol. 5846, pp. 2–19. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-04912-5_2
7. Davies, J., Gibbons, J.: Formal methods for future interoperability. ACM Inroads **41**(2), 60–64 (2009). https://doi.org/10.1145/1595453.1595463
8. Feinerer, I., Salzer, G.: Automated tools for teaching formal software verification. In: Teaching Formal Methods, p. 5 (2006)
9. Gopalakrishnan, G.: Formal methods for surviving the jungle of heterogeneous parallelism. In: 2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum, pp. 1321–1324. IEEE, Shanghai, China (2012). https://doi.org/10.1109/IPDPSW.2012.164
10. Hallerstede, S., Leuschel, M.: How to explain mistakes. In: Gibbons, J., Oliveira, J.N. (eds.) TFM 2009. LNCS, vol. 5846, pp. 105–124. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-04912-5_8
11. Kitchenham, B., Charters, S.: Guidelines for performing systematic literature reviews in software engineering (2007)
12. Kofroň, J., Parízek, P., Šerý, O.: On teaching formal methods: behavior models and code analysis. In: Gibbons, J., Oliveira, J.N. (eds.) TFM 2009. LNCS, vol. 5846, pp. 144–157. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-04912-5_10
13. Lipaczewski, M., Ortmeier, F.: Teaching and training formal methods for safety critical systems. In: 2013 39th Euromicro Conference on Software Engineering and Advanced Applications, pp. 408–413. IEEE, Santander, Spain (2013). https://doi.org/10.1109/SEAA.2013.54
14. Loukanova, R.: Teaching formal methods for computational linguistics at uppsala university. In: Teaching Formal Methods, p. 6 (2006)
15. Méry, D.: A simple refinement-based method for constructing algorithms. SIGCSE Bull. **41**(2), 51–59 (2009). https://doi.org/10.1145/1595453.1595462
16. Miller, A., Cutts, Q.: The use of an electronic voting system in a formal methods course. In: Formal Methods in the Teaching Lab: Examples, Cases, Assignments and Projects Enhancing Formal Methods Education. A Workshop at the Formal Methods 2006 Symposium, Hamilton, Ontario, Canada, 26 Aug 2006, pp. 3–8. McMaster University, Hamilton (2006)

17. Ölveczky, P.C.: Teaching formal methods based on rewriting logic and Maude. In: Gibbons, J., Oliveira, J.N. (eds.) TFM 2009. LNCS, vol. 5846, pp. 20–38. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-04912-5_3
18. Parnas, D.L., Eng, P., Soltys, M.: Basic science for software developers (2006)
19. Prasetya, W., et al.: Having fun in learning formal specifications. In: 2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering Education and Training (ICSE-SEET), pp. 192–196. IEEE, Montreal (2019). https://doi.org/10.1109/ICSE-SEET.2019.00028
20. Roychoudhury, A.: Introducing model checking to undergraduates. In: Formal Methods Education Workshop, pp. 9–15 (2006)
21. Schreiner, W.: The RISC ProofNavigator: a proving assistant for program verification in the classroom. Formal Aspects Comput. **21**(3), 277–291 (2009). https://doi.org/10.1007/s00165-008-0069-4
22. Sekerinski, E.: Teaching the mathematics of software design. In: Formal Methods in the Teaching Lab, p. 53 (2006)
23. Shilov, N.V.: Kwangkeun Yi: engaging students with theory through ACM collegiate programming contests. Commun. ACM **45**(9), 98–101 (2002)
24. Spichkova, M., Zamansky, A.: Teaching of formal methods for software engineering. In: Proceedings of the 11th International Conference on Evaluation of Novel Software Approaches to Software Engineering, pp. 370–376. SCITEPRESS - Science and and Technology Publications, Rome (2016). https://doi.org/10.5220/0005928503700376
25. Spies, K., Schätz, B.: A playful approach to formal models a field report on teaching modeling fundamentals at middle school. In: Formal Methods in the Teaching Lab, p. 45 (2006)
26. Utting, M., Reeves, S.: Teaching formal methods lite via testing. Softw. Test. Verif. Reliab. **11**(3), 181–195 (2001). https://doi.org/10.1002/stvr.223

Author Index