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]:

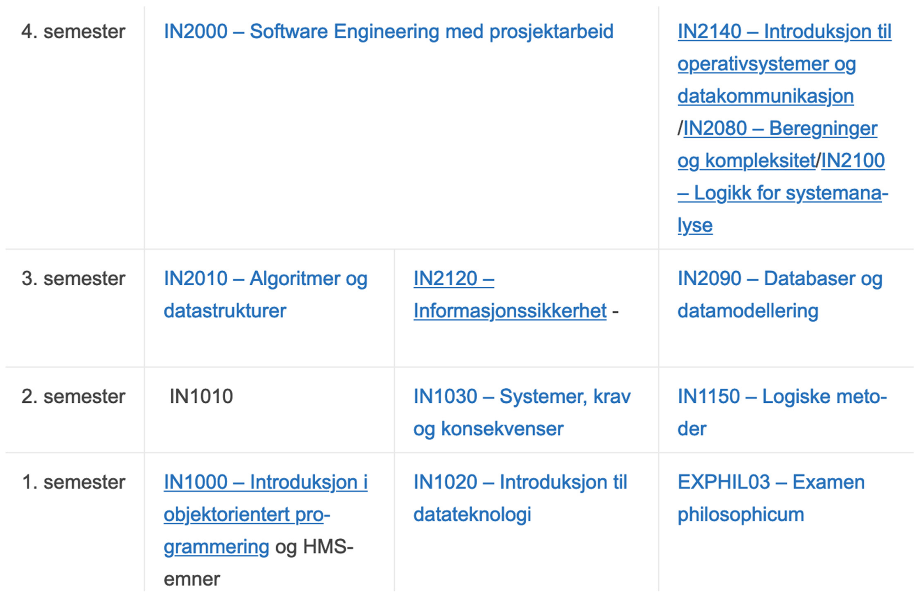
> In industry, formal methods have a reputation for requiring a huge amount of training and effort to verify a tiny piece of relatively straightforward code, so the return on investment is justified only in safety-critical domains (such as medical systems and avionics).

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.

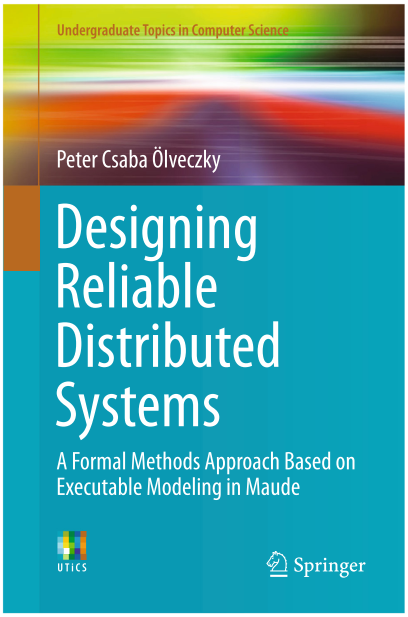

Undergraduate Topics in Computer Science
Peter Csaba Ölveczky
Designing Reliable Distributed Systems
A Formal Methods Approach Based on Executable Modeling in Maude
UTiCS
Springer

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