Teaching Formal Methods to Undergraduate Students Using Maude

Peter Csaba Ölveczky(✉)

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

Abstract. I have been teaching an introductory formal methods course based on Maude—first to third- and fourth-year students, and lately to second-year students—at the University of Oslo for a number of years. The first part of the course introduces functional modules in Maude and covers basic topics in term rewriting, whereas the second part of the course uses Maude to formally model and analyze a number of classic distributed systems, including: transport protocols such as the alternating bit and the sliding windows protocols, the two-phase commit protocol for distributed atomic commitment, distributed algorithms for mutual exclusion and leader election, and authentication protocols.

In this invited "experience report" I briefly motivate the use of Maude for an introductory formal methods course, outline the course content, and summarize student feedback and my own impressions about the course.

1 Introduction

Too many years ago I had to design an introductory formal methods course for third-year students at the University of Oslo. The main question was, and remains: *How* to teach an elective introductory formal methods course in an environment where students have never heard about formal methods, and where our colleagues are not overly receptive to the usefulness and beauty of a giving formal treatment to computer systems?

In this "invited experience report" I briefly describe the setting and some challenges when it comes to teaching introductory formal methods courses, and how these challenges might be overcome (Sect. 2). In Sect. 3 I discuss how some papers argue that formal methods should be taught. In Sect. 4 I argue that—based on the criteria for teaching formal methods—rewriting logic [14] and its accompanying Maude tool [10] should provide a suitable framework for introducing formal methods to undergraduate students.

I have taken my own medicine and have been teaching formal methods based on Maude for twenty years; first to third- and fourth-year students, and since 2019 to second-year students. When the course had reached a certain stability and maturity, I wrote a textbook, called "Designing Reliable Distributed Systems: A Formal Methods Approach Based on Executable Formal Modeling in Maude," which was published in 2018 in Springer's *Undergraduate Topics in Computer Science* series [21]. In Sect. 5 I give an overview of the content of the

K. Bae (Ed.): WRLA 2022, LNCS 13252, pp. 85–110, 2022.
https://doi.org/10.1007/978-3-031-12441-9_5

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 structure of the "Programming and Networks" bachelor degree at my university. The third year is devoted to freely selected courses and is not shown.

course and of this book. Finally, Sect. 6 summarizes my experiences and the results of the anonymous student evaluation throughout the years.

The longer paper [19] on the same topic gives more details, and presents a broader case for using Maude for teaching, since—in contrast to this paper—it is aimed at the formal methods community without expertise in Maude.

2 Setting and Challenges

In this section I discuss some challenges involved in trying to teach formal methods to undergraduate students at a place like the University of Oslo.

When Turing Award winner and department founder Ole-Johan Dahl was at the department, formal methods/verification was a mandatory course in the Bachelor program on "Programming," and hence around 80 students took the formal methods course every year. However, since then my esteemed colleagues have relegated formal methods to an elective course in the periphery of that Bachelor program, shown in Fig. 1, which shows the courses that the students should take in the first two years. (The program is in Norwegian, so there is no English version.) The formal methods course ("IN2100–Logikk for Systemanalyse") has to compete with a course introducing operating systems and computer networks and one on computability and complexity for the final 10 credits in this Bachelor program. In such a setting, would an 18–20-year-old student, who has no idea what formal methods are, choose to take the formal methods course instead of a (supposedly good, from what I hear) course on operating systems

and computer networks? I am pretty sure that as a 19-year-old student I would have taken the OS course instead, and would never have been exposed to formal methods during my studies. However, I was lucky enough to study while the above-mentioned Ole-Johan Dahl was still teaching, so we *had to* take the verification course, which led me to my current path.

This problem is compounded by the fact that students at the University of Oslo study "Informatics" to quickly get a good job, and therefore prefer to take more "practical," seemingly more work-relevant, courses. In a recent *Communications of the ACM* blog post [25], Daniel G. Schwartz at the Florida State University writes that this does not just apply to Norway: "Another issue is that most CS students are primarily only interested in acquiring the skills that will enable them to find jobs as software developers. Few have any interest in pursuing graduate studies and research. For this reason, they see no purpose in studying theoretical topics." If this were not enough, our students tend to have very limited background in mathematics, and tend not to study too much.

How can we overcome such "structural" challenges? Unless the Bachelor curriculum changes, or the course again becomes a third-year course, it would seem hard to attract students. Therefore, the main hope is to create *such a good course* that students recommend the course on an unknown subject to their peers. Indeed, most students taking the course this year do it because they heard it was a good course. The problem with this "word-of-mouth" strategy is that students mostly socialize with students at the same stage in their studies. Because of this, and because students "try out" many courses at the beginning of the semester, there is a need to *quickly* demonstrate the power and usefulness on relevant problems and applications. Furthermore, the lack of mathematical background[1] also means that the course should not be very hard or "theoretic."

Related to the above challenges, and maybe the reason why the formal methods course has been relegated to the purgatory of elective courses, is the following misconception, quoted from [17]:

> 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).

Fortunately, formal methods and their tools have matured quite a lot, and we also have a better understanding of what formal methods can and cannot do well. We need to advertise the success stories of formal methods; for example, in my course I discuss in some depth: the paper "How Amazon Web Services Uses Formal Methods" [17] by engineers developing the key cloud computing systems at Amazon Web Services; the work of Ralf Sasse and others to find previously unknown flaws in the Internet Explorer web browser using Maude [9]; and the work by David Basin and Ralf Sasse and others who use "Maude-related" methods to find serious flaws in the 5G standard [23] and, in particular, in the

[1] I once got complaints from the head of studies for supposedly having shown a quantifier in a lecture!

VISA and MasterCard payment systems [5,6]. The main Norwegian newspaper has even made a short video about the latter, which I show to my students.

Another misconception, that we ourselves quite often perpetuate, is that formal methods are aimed at *safety-critical* systems. It *is* true that society is increasingly dependent on such systems (from self-driving cars to airplanes and power distribution systems). However, in a country like Norway, I do not think that many students will end up developing safety-critical systems. Selling formal methods for safety-critical systems could therefore be self-defeating. Fortunately, in contrast to 20–30 years ago, when everybody developed their own systems for local use, these days cloud computing has led to world-wide services, where "the winner takes it all" in each kind of "service," with the profit for being that winner potentially enormous. Together with increasing system complexity, this need to develop the highest-quality system implies that an additional up-front investment in system quality really pays off in "mainstream" software development; this is also the main message of the above AWS paper [17].

Another challenge is the worse and worse mathematical background, and skepticism toward mathematics, among students. In [25] Schwartz writes that "most of CS undergrads don't like mathematics and so-called 'theory' courses, and would prefer to not take them," and quotes Leslie Lamport, who argues that "while good programming really requires mathematical precision, [Lamport] also acknowledges that 'basically, programmers and many (if not most) computer scientists are terrified by math.' " I guess that the solution to this problem is to use *accessible/intuitive formal methods* that do not require much mathematical background, and/or to make formal methods look more like "programming," which they like and master.

Another issue that sometimes pops up is that formal methods are not integrated with other courses. Therefore, showing the strength of, or at least exemplify the use of, formal methods to model and analyze systems encountered in other courses would show students—and maybe also our colleagues defining study plans—the usefulness of formal methods. This could include examples from security, networking/communication, databases and distributed transactions, operating systems, etc.

The paper [12] discusses the problem of addressing appropriate systems. The authors write that formal methods courses use examples and case studies that are either "constructed and thus do relate to practice" or are "based on projects of industry partners and are thus, too involved for students." Again, we need to address problems which look relevant, in fields such as social media, online shopping and other cloud applications (i.e., distributed transactions), and/or in authentication. To be able address relevant problems in different courses/domains we need an *expressive formalism*.

3 How to Teach Formal Methods?

Section 2 listed some challenges involved in making students take formal methods courses when they are not mandatory, and listed some possible "solutions" to

these challenges. In this section I first briefly discuss a few key papers on teaching formal methods (see, e.g., [19] and [8] for longer discussions on papers on the topic), and then try to distill some requirements for courses in formal methods.

3.1 A Few Papers on Teaching Formal Methods.

As its title suggests, in their paper "Teaching Formal Methods for Software Engineering: Ten Principles," Cerone, Roggenbach, Schlingloff, Schneider, and Shaikh list and elaborate on ten principles for teaching formal methods, which in my view boil down to the following "principles:"

- Formal methods are too large to gain encyclopedic knowledge; we should just use a few formal methods, since "there is loads to gain by intensively studying [a] few methods."
- Formal methods need tools, which "teach the method," and lab classes, which should imply that we need a high-quality and fairly stable tool.
- Formal methods are best taught by examples.

In their paper "Teaching Concurrency: Theory in Practice" [1], Aceto, Ingolfsdottir, Larsen, and Srba also share the view that "less is more," and that we should repeatedly convey key concepts, instead of providing a broad overview. They also advocate using *automatic* verification tools and very expressive and flexible, yet mathematically simple, executable modeling formalisms, as well as using modal and temporal logics to specify system requirements.

In the paper with the promising title "Teaching Formal Methods in the Context of Software Engineering," Liu, Takahashi, Hayashi, and Nakayama take a somewhat contrarian view [13]. They propose using VDM, refinement, and Hoare logic, but admit that "none of these techniques is easy to use by ordinary practitioners to deal with real software projects." In another divergence from teaching-formal-methods orthodoxy, they claim that "most effective for students [...] is to write formal specifications by hand, as they learn English as a foreign language." Like others, they also argue that "each course should not be too ambitious; instead it should be focused." Finally they admit that "there is little hope to apply refinement calculus in practice."

There is also a "white paper" on teaching formal methods, "Rooting Formal Methods Within Higher Education Curricula for Computer Science and Software Engineering: A White Paper" [8] by a number of participants, including me, at the *First International Workshop on "Formal Methods – Fun for Everybody"* in 2019. This paper advocates that a formal methods course must be mandatory for all Bachelor students in computer science. In addition to also being a proponent for using small "games" to teach formal methods, this paper emphasizes tool use, but not industrial tools, which "can cause frustration."

3.2 What to Teach?

We can try to summarize the various requirements for an introductory course in formal methods, where such a course is not mandatory, as follows:

1. It should repeatedly convey key formal methods concepts. But what are these key concepts? Certainly mathematical *modeling/formalization* of both *systems/designs* and of *requirements*, and of course *reasoning* about the models in terms of *model checking* and *verification*, and preferably also model-based *performance estimation*. Another key, slightly orthogonal use of formal methods, is the mathematical analysis of *code*. More generally, one might also want to introduce students to *logical reasoning* in general, dealing with logics, deduction rules, models, satisfaction, and may include key folklore results.
2. It should be fun for the students.
3. It should use relevant applications/examples, also related to other courses the student take, and should seem relevant to today's systems. To be able to this, the modeling formalism must be fairly general and expressive.
4. It should use *few*, but mature, tools, which should seem industry-relevant.
5. We must motivate with industrial success stories.
6. It should be simple and intuitive, and not require much mathematical background.
7. It should support automatic model checking methods.
8. The formalism must be executable, expressive, and general.

4 Why Teaching Formal Methods Using Maude?

In my view, rewriting logic and its Maude language and tool should be very well suited to introduce formal methods to undergraduates, as I think that Maude satisfies the "requirements" in the previous sections as follows:

1. (Repeatedly convey main formal methods concepts.) Maude primarily deals with modeling systems/designs in rewriting logic. It also supports formalizing systems requirements in the most elegant and intuitive temporal logic [27], linear temporal logic (LTL), and provides an LTL model checker. While Maude's primary focus is on modeling and model checking of said models, rewriting logic has been used to define the semantics of many programming languages [15,16], and is the foundation of the $\mathbb{K}$ programming language semantics and analysis framework [24]. $\mathbb{K}$ is a leading tool for formalizing the semantics of programming languages, and then analyzing programs, including Ethereum contracts [22]. For model-based performance analysis, rewriting logic has extensions to timed [18,20] and probabilistic [2] systems, such that the performance of the resulting probabilistic (and possibly timed) models can be analyzed by statistical model checking using, e.g., the PVeStA tool [3]. Finally, teaching Maude also provides an excuse to introduce simple logics (equational, rewriting, and temporal logics) and their deduction systems, satisfaction, models, and some folklore (un)decidability proofs.
2. (Fun for students.) What does "fun for students" mean? The students probably study computer science because they like programming. When I was a student, I loved functional programming. Maude modeling is essentially (first-order) functional programming in an object-oriented style.

3. (Relevant examples.) While simple, the Maude specification formalism is expressive and general. Therefore, relevant examples from different fields of computer science can easily be specified by undergraduates; as explained later, I use examples from security/cryptographic protocols, database courses, key communication protocols and distributed algorithms in my course.
4. (Few and mature tools.) My course only uses Maude, which is a mature and high-quality tool.
5. (Motivate with industrial success stories.) There are probably other tools with more industrial success stories. Maude has been used to find a number of previously unknown errors in Internet Explorer, as well as to model (aspects of) industrial systems such as Google's Megastore, Apache Cassandra, Apache ZooKeeper, and so on. However, the Tamarin tool [4] has been used to break the EMV card payment standard [6] and the 5G standard [23], and is based on multiset rewriting, and even includes some parts of the Maude implementation. Furthermore, the rewriting-logic-based $\mathbb{K}$ framework has been applied commercially to analyze electronic contracts on the blockchain.
6. (Simple and intuitive.) Maude is based on equational and rewriting logic. Equations—like $(x + y)^2 = x^2 + 2xy + y^2$—and their use to simplify an expression by replacing equals for equals, is something that all students are familiar with from school. Rewriting is fairly similar, so this simplicity and intuitive logic is one main strengths of rewriting logic, and should make it an ideal formalism for introducing formal methods to undergraduates.
7. (Model checking.) Maude provides a range of automatic analysis methods, including rewriting for quick simulation/prototyping and automatic model checking methods such as reachability analysis and LTL model checking.
8. (Executable expressive and intuitive formalism.) As elaborated above, the Maude formalism is both expressive, executable, and simple and intuitive.

Reasons for skepticism include a lack of good documentation for beginners; the manual is very nice and comprehensive, but is not well suited to learn the formalism for a formal methods novice. Furthermore, I use Full Maude for object-based modeling, but the lack of error messages in Full Maude is a significant problem. I understand that others, including Francisco Durán, teach object-based modeling by "encoding" classes and object rules directly in Maude.

5 Course and Textbook Content

In this section I give an overview of the content of the second-year introductory formal methods course I teach at the University of Oslo, and of its aforementioned textbook, "Designing Reliable Distributed Systems: A Formal Methods Approach Based on Executable Modeling in Maude" (Fig. 2). I also mention interesting exam problems I have given, and some exercises in the book, which might be useful for professors looking for exam problems.

The course consists of 14–15 90-minute lectures, and of the same number of 90-minute problem-solving sessions. I cannot assume much mathematical knowl-

edge, even though many (but far from all) students have taken a basic course in logic before taking my course.

The course is divided into two parts: *Part I* deals with equational specification in Maude, and covers basic theory of algebraic specifications and term rewrite systems, in addition to defining equational specifications using Maude. *Part II* deals with specifying and analyzing various distributed systems in Maude using rewriting logic. It is implicitly also meant to introduce some fundamental algorithms in distributed systems.

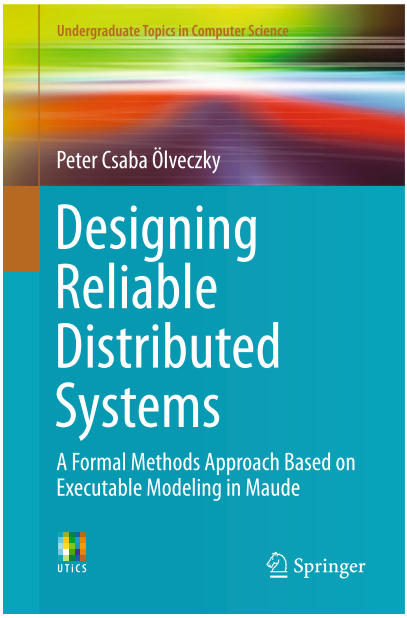

Fig. 2. Course textbook.

5.1 Part I: Equational Specification in Maude and Term Rewrite Theory

Equational Specification in Maude (3 lectures). These lectures introduce basic equational specifications in Maude, starting with many-sorted ones, followed by order-sorted and then membership specifications. We exemplify such specifications with Peano natural numbers, with a wide range of functions on such numbers, and then turn to Boolean values, lists, multisets, binary trees, and so on.

I then cover the built-in Maude modules `BOOL`, `NAT`, `INT`, `STRING`, `CONVERSION`, and `RANDOM`, and specification modulo structural axioms such as associativity, commutativity, and identity. This book includes a section on parametrized modules/programming in Maude, although I do not teach this in class.

Examples include sorting algorithms such as quicksort, merge-sort, insertion-sort, and bubble-sort.

Example 1. Lists of natural numbers can be defined as follows in Maude:

```
fmod LIST-NAT is protecting NAT .
  sorts List NeList .
  subsort Nat < NeList < List .

  op nil : -> List [ctor] .
  op _::_ : List List -> List [ctor assoc id: nil] .
  op _::_ : NeList NeList -> NeList [ctor assoc id: nil] .

  op length : List -> Nat .
```

```
  ops first last : NeList -> Nat .
  op reverse : List -> List .

  vars M N K : Nat .   var L : List .

  eq length(nil) = 0 .      eq length(N :: L) = 1 + length(L) .
  eq first(N :: L) = N .    eq last(L :: N) = N .
  eq reverse(nil) = nil .   eq reverse(N :: L) = reverse(L) :: N .
endfm
```

The module `LIST-NAT` defines a sort `List`, for lists of natural numbers, and a sort `NeList`, for non-empty such lists. Lists are constructed by the constructors `nil` and an infix associative "list concatenation" function `_::_`, so that a list $\langle 6, 2, 8, 4, 6 \rangle$ is represented as the term `6 :: 2 :: 8 :: 4 :: 6`. Since the concatenation function is declared to be associative, parentheses are not needed in this term. Furthermore, since the concatenation constructor is declared to have identity element `nil`, any list l is considered identical to the lists l `::` `nil` and `nil` `::` l, explaining why the equations above do not explicitly consider the case of singleton lists.

The well-known *merge-sort* algorithm can then be specified as follows, where the `merge` function is declared to be commutative:

```
fmod MERGE-SORT is protecting LIST-NAT .
  op mergeSort : List -> List .
  op merge : List List -> List [comm] .

  vars L1 L2 : List .  vars NEL1 NEL2 : NeList .  vars M N : Nat .

  eq mergeSort(nil) = nil .
  eq mergeSort(N) = N .

  ceq mergeSort(NEL1 :: NEL2) = merge(mergeSort(NEL1), mergeSort(NEL2))
      if length(NEL1) == length(NEL2)
         or length(NEL1) == length(NEL2) + 1 .

  eq merge(nil, L1) = L1 .
  ceq merge(M :: L1, N :: L2) = M :: merge(L1, N :: L2) if M <= N .
endfm
```

I also introduce some classic NP-complete problems (Knapsack, Subset Sum, Traveling Salesman, Hamiltonian Circuit, Clique, etc.) and show in the book how Subset Sum and Hamiltonian Circuit can be solved in Maude.

Example 2. In the Subset Sum problem the question is: Given a multiset MS of positive natural numbers and a number $K > 0$, is there a subset of MS whose elements have the sum K? This problem can be solved by the following function `subsetSum`, where the module also declares a data type `Mset` of multisets of nonzero natural numbers:[2]

[2] The function `sd` gives the difference between two natural numbers, since subtraction is not defined on natural numbers.

```
fmod SUBSET-SUM is protecting NAT .
  sort Mset .    --- multisets of non-zero natural numbers
  subsort NzNat < Mset .
  op none : -> Mset [ctor] .    --- empty multiset
  op _;_ : Mset Mset -> Mset [ctor assoc comm id: none] . --- mset union

  op subsetSum : Mset NzNat -> Bool .

  vars N K : NzNat .   var MS : Mset .

  eq subsetSum(none, K) = false .
  eq subsetSum(N ; MS, K) =
       if N == K then true
       else (if N > K then subsetSum(MS, K)
             else subsetSum(MS, sd(K, N)) or subsetSum(MS, K) fi)
       fi .
endfm
```

Termination (1+ lecture). This is one of my favorite topics. This part presents the basics of classic theory on termination of rewriting à la Dershowitz, in the simple, unsorted, and unconditional case without function attributes. The book shows one of the well-known proofs for the undecidability of termination based on reducing the uniform halting for Turing machines to a term rewrite system termination problem. It then discusses methods for proving termination using "weight functions" on well-founded domains, before presenting the elegant theory of simplification orders. Finally, it introduces two such simplification orders: the lexicographic and the multiset path order.

One exercise—used in two exams, to the chagrin of the students—is defining the lexicographic path order in Maude. This gives a taste of meta-programming: how an equational specification can be represented as a Maude term.

I loved to teach the theory of simplification orders, but since I started teaching the course to second-year students, I no longer deal with this nice theory, or with representing Turing machines as term rewrite systems. The grateful second-year undergraduate students are taught temporal logic instead.

Confluence (1- lecture). I continue the term rewriting basics by devoting a (short) lecture and book chapter to introducing students to confluence, again, in the most basic setting. I cover the expected bases: Newman's Lemma, unification, and checking (local) confluence using the Critical Pair's Lemma. I am not sure I convey this topic in a particularly interesting way, and I do not believe that confluence is the favorite topic of most students.

5.2 Equational Logic (1 lecture)

In one, probably quite heavy, lecture I cover equational logic: the deduction system (again in the unsorted and unconditional case), undecidability, and the usual equivalences between deduction in equational logic and equational simplification.

Then I discuss validity in *all* structures satisfying the equations E versus validity in the *"intended"* structure. Our Maude specifications (also) define the domains of our data types, so we are mostly interested in properties holding in models with those elements we have so painstakingly defined. This leads us to inductive theorems. I start by excusing myself that I cannot give a (finitary) sound and complete proof systems for inductive validity, since such a proof system for cannot exist due to the negative solution to Hilbert's Tenth Problem (thankfully for the students, the argument why that solution leads to the non-existence of the desired proof system for inductive theorems has been relegated to a long footnote). I present the general induction principle for data types—which can be seen as special case of the induction principle for natural numbers—and show the usual examples (binary trees, lists, natural numbers, etc.). I also show how Maude sometimes can be used to automatically prove induction theorems (or at least discharge the proof obligations):

Example 3. We can let Maude prove by induction that our addition function (defined in the module `NAT-ADD`) is associative:

```
fmod NAT-ADD is
  sort Nat .
  op 0 : -> Nat [ctor] .
  op s : Nat -> Nat [ctor] .
  op _+_ : Nat Nat -> Nat .

  vars M N : Nat .
  eq 0 + M = M .
  eq s(M) + N = s(M + N) .
endfm

fmod NAT-ASSOC-IND-PROOF is including NAT-ADD .
  ops t t2 t3 : -> Nat .
  eq (t + t2) + t3 = t + (t2 + t3) .        --- induction hypothesis
endfm

red (0 + t2) + t3 == 0 + (t2 + t3) .        --- base case
red (s(t) + t2) + t3 == s(t) + (t2 + t3) .  --- inductive case
```

Both `red` commands returns `true`, proving associativity of our addition function.

Models of Equational Specifications. We are doing "mathematical modeling" because we use an equational specification to precisely specify a mathematical model/structure. For equational specifications the models are algebras. Although I do not teach this to second-year students, I have devoted a chapter of the textbook to the classics of algebras in the context of algebraic specifications. This chapter covers many-sorted Σ-algebras, then (Σ, E)-algebras, leading to the algebras $\mathcal{T}_{\Sigma,E}$ and *normal form algebras*. I present the proof of Birkhoff's Completeness Theorem. Finally, I discuss initial algebras and why they are the ones we really wanted to specify, and that $\mathcal{T}_{\Sigma,E}$ and the normal form algebra both are this desired mathematical model specified by an equational specification.

5.3 Part II: Modeling and Analysis of Dynamic/Distributed Systems Using Rewriting Logic

In the second part of the course/book, we leave the static world of equations, and the classic theory of algebraic specification and (term) rewrite systems, and move to modeling and analyzing distributed, and dynamic systems in general. I am not aware of any other textbook (in English, at least) that gives an introduction to the modeling and analysis of distributed systems using Maude.

As mentioned elsewhere, additional goals include:

- giving a brief introduction to fundamental distributed algorithms and other folklore systems (such as the dining philosophers problem) that the students should know for "computer science literacy"; and
- looking at systems that are relevant to other courses that students take.

Rewriting Logic and Analysis in Maude (1 lecture). I start by explaining why equations are not suitable for modeling dynamic systems, and then introduce rewriting logic and its proof system, including the definition of concurrent steps. This can be introduced by small games (since some papers on teaching formal methods advocate that).

Example 4. In the following simple model of a soccer game, the term `"Malmo FF" - "Barcelona" 3 : 2` models a state in an (ongoing) game, whereas a state `"Malmo FF" - "Barcelona" finalScore: 4 : 2` represents a finished game.

```
mod GAME is protecting NAT . protecting STRING .
  sort Game .
  op _-_:_:_ : String String Nat Nat -> Game [ctor] .
  op _-_finalScore:_:_ : String String Nat Nat -> Game [ctor] .

  vars HOME AWAY : String .   vars M N : Nat .

  rl [homeTeamScores] : HOME - AWAY M : N => HOME - AWAY M + 1 : N .
  rl [awayTeamScores] : HOME - AWAY M : N => HOME - AWAY M : N + 1 .
  rl [finalWhistle] : HOME - AWAY M : N => HOME - AWAY finalScore: M : N .
endm
```

Example 5. In the *whiteboard game*, some natural numbers are written on a whiteboard. In each step of this exciting game, any two numbers n and m can be replaced by their arithmetic mean $n+m$ `quo` 2. Importing the data type `Mset` for multisets of numbers from Example 2, this game can be specified as follows:

```
mod WHITEBOARD is protecting SUBSET-SUM .
  vars M N : NzNat .
  rl [replace] : M ; N => (M + N) quo 2 .
endm
```

There is a treasure trove of small examples on this topic in the textbook and among the exam problems. As a running example, I use modeling the life of a

person—her age and civil status. This model is then extended to a population; i.e., a multiset of persons, who can communicate synchronously to get engaged, and use message passing to separate. Other examples include classics such the towers of Hanoi, tic-tac-toe, the coffee bean game, modeling all traveling salesman trips and using search to find short trips, packing "suitable" knapsacks (instead of just knowing that *there is* a suitable knapsack), and simulating the behaviors of Turing machines.

This lecture also covers the rewrite and search commands of Maude.

Example 6. We can use the rewrite command to simulate one behavior of the whiteboard game from a given initial state, and search to find all reachable *final* states where the resulting number is less than 13. We then exhibit the path to one such desired final state:[3]

```
Maude> rew 6 ; 33 ; 99 ;  1 ; 7 .
result NzNat: 59

Maude> search  6 ; 33 ; 99 ;  1 ; 7 =>! M such that M < 13 .

Solution 1 (state 151)
M --> 12

Solution 2 (state 153)
M --> 11

No more solutions.

Maude> show path 153 .
state 0, Mset: 1 ; 6 ; 7 ; 33 ; 99
===[ rl N ; M => (N + M) quo 2 [label replace] . ]===>
state 10, Mset: 1 ; 6 ; 7 ; 66
===[ rl N ; M => (N + M) quo 2 [label replace] . ]===>
state 51, Mset: 1 ; 7 ; 36
===[ rl N ; M => (N + M) quo 2 [label replace] . ]===>
state 125, Mset: 1 ; 21
===[ rl N ; M => (N + M) quo 2 [label replace] . ]===>
state 153, NzNat: 11
```

Object-Based Modeling of Distributed Systems (1 lecture). This lecture first shows that "objects" can be modeled as standard Maude terms, and explains that the state of a distributed system naturally can be seen as a multiset of objects and messages.

After showing that all this can be modeled in Maude, I make the possibly problematic decision to use Full Maude's support for very convenient object-oriented syntax, including for subclasses. I prefer a clean theory and un-cluttered models—at a high cost of lot of frustration when Full Maude.

[3] The command echo and some other Maude output are not shown.

The running example, populations of persons, is ideal for illustrating many notions of distributed object-based models: dynamic object creation (birth of children) and deletion (death of a person), rules only involving a single object (like birthdays), synchronous communication (getting engaged) and message-based communication (getting separated), and e subclasses to model that some people are Christians and others are Muslims.

This chapter/lecture also models the dining philosophers problem in an object-oriented style, and uses Maude's pseudo-random number generator to model different variations of *blackjack*, where the next card is drawn pseudo-randomly from the remaining cards. We can then perform randomized simulations to simulate how much money I have left after a day in the casino.

Modeling Communication and Transport Protocols (1 lecture). The goal of Part II is to model sophisticated distributed systems. To achieve this we need to model different forms and variations of communication: unicast, multicast, broadcast and "wireless" broadcast, message loss, ordered communication through links, and so on.

The first "larger" applications are a range of well-known transport protocols used to achieve ordered and reliable message communication on top of an unreliable and unordered communication infrastructure. we begin with a TCP-like sequence-number-based protocol. When the underlying infrastructure provides ordered (but lossy) communication between pairs of nodes, the sequence numbers in the TCP-like protocol can be reduced to 0 and 1, giving us the alternating bit protocol (ABP). Generalizing the TCP-like protocol and ABP so that a node can send any one of k different messages at any time, instead of only the same message, gives us the sliding windows protocol, supposedly the most used protocol in distributed systems. I like sliding windows for a homework exercise/project, since the search commands take some time to finish, which I think is useful for students who are used to programs always giving immediate feedback.

Distributed Algorithms (1 lecture). As mentioned, one of the goals of the course is to give a flavor of distributed systems, which we do using a number of fundamental distributed algorithms that are still used in state-of-the-art cloud-based transaction systems.

These algorithms are also easy to motivate using modern distributed transactions. For example, in today's cloud-based world the same *eBay* item could have been sold (at the dying moments of an auction) to two different persons; one through a server in Munich, and one through a server in Vanuatu. Or we can imagine an online travel agency with the following distributed transaction:

$$\begin{array}{l} \mathit{reserve}(X, \text{OSL-CDG}, \text{KLM}, \text{Dec 6 to 15}); \\ \mathit{reserve}(X, \text{Ritz}, \text{Imperial Suite}, \text{Dec 6 to 15}); \\ \mathit{reserve}(X, \text{Chez M}, \text{dinner}, \text{Dec 9}); \\ \mathit{pay}(X, \mathit{6000}, \text{MasterCard}, \mathit{1234567891234567}, \mathit{11/20}, ...); \end{array}$$

These examples can motivate the two-phase commit (2PC) protocol: In the *eBay* example, Vanuatu can veto Munich's commit request if it has also sold the item. In the Paris vacation example, if each single operation (at its own site) goes through, the entire transaction should be committed. If, however, one of the operations cannot be performed (there is no money on the credit card, or the Imperial Suite at the Ritz is not available those days), the entire distributed transaction must be aborted. 2PC ensures this.

Multiple distributed operations on the same data could lead to "lost updates." This can motivate the use of distributed mutual exclusion algorithms.

Example 7. In the *token-ring-based* distributed mutual exclusion algorithm, the nodes are organized in a ring structure. There is one *token*, and a node must hold the token to enter the critical section; when it exits the critical section, or when a node receives the token without wanting to enter the critical section, it sends the token to the next node in the ring.

This algorithm, where each node alternates forever between executing outside the critical section, and (if possible) executing inside the critical can be modeled using objects a class `Node`, whose attribute `status` can have the values `outsideCS` (the node is executing outside the critical section), `waitForCS` (the node is waiting to enter the critical section), and `insideCS` (the node is executing inside the critical section). The attribute `next` denotes the "next" node in the ring. The token is being sent around as a message:

```
load model-checker
load full-maude31

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

  class Node | next : Oid, status : Status .

  op token : -> MsgContent [ctor] .

  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)

(omod INITIAL is including TOKEN-RING-MUTEX .
  ops a b c d : -> Oid [ctor] .  --- object names
  op init : -> Configuration .   --- an 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 check whether it is possible to reach a state in which two nodes are executing in the critical section at the same time:

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

No solution.
```

Distributed mutual exclusion algorithms are ideal exam problems. The book presents the central server algorithm, the token ring algorithm, and Maekawa's voting algorithm, and I have used Lamport's bakery algorithm and the interesting Suzuki-Kasami algorithm as exam problems.

2PC solves the "same item sold twice problem" by aborting the whole transaction, since one site will veto another site's attempt to commit a (conflicting) transaction. A better idea is to sell the item to one of the buyers, which leads us to distributed leader election and distributed consensus. Leader election is a key part of distributed consensus algorithms, such as Paxos, which again are key components in many of today's cloud-based systems, like Google's Megastore. The textbook describes the Chang and Roberts ring-based distributed leader election algorithm and a spanning-tree based useful for wireless systems. I also introduce distributed consensus, but leave modeling Paxos as an exercise.

Staying on the cloud computing track, a very nice exam problem that illustrates how a cloud-based replicated data store can compromise between desired

levels of consistency, performance, and fault tolerance is inspired by Apache Cassandra: Your data are stored at n replicas; a read or a write request is sent to all replicas. A client gets the answer ("ok" for writes and the most recent value of the data item for reads) when k replicas have responded. A lower k gives improved performance (shorter waiting time for the client) and fault tolerance (since $n - k$ replicas can crash) improves, while consistency suffers (not even "read-your-writes" holds when k is low). The system should satisfy "eventual consistency," but other transaction guarantees depend on the value of k.

Finally, this chapter presents useful techniques for analyzing fault tolerance by modeling failures and repairs.

Cryptographic Protocols: Breaking NSPK (1 lecture + 1 guest lecture). One of my favorite chapters/lectures introduces public-key and shared-key cryptography. We then model the well-known Needham-Schroeder Public-Key (NSPK) authentication protocol. This is a great example to motivate formal analysis. The protocol is super small, only three lines, yet its flaws went undetected for 17 years before they were found by formal methods. This demonstrates that even very small distributed protocols are hard to understand, and that formal methods are useful to find subtle bugs in distributed systems.

It is very easy and natural to model NSPK with four intuitive rewrite rules. Another 13 or so simple rules model Dolev-Yao intruders. A plain Maude search for an unwanted "trusted" connection then breaks NSPK in around 100 minutes on my laptop; the standard search for compromised keys takes a few seconds. The students understand these models, and can modify them (e.g., to analyze Lowe's fix of NSPK) without problems.

NSPK is still a simpler older protocol, and Maude is not a cryptanalysis tool (although Maude-NPA [11] is a leading one). However, a tool like the Tamarin prover [4] is based on multiset rewriting, and has been in the news in Norway and elsewhere for breaking our card payment systems [5,6]. Ralf Sasse from ETH Zürich has generously given a guest the last two years where he talks about using Maude to find news flaws in the Internet Explorer web browser as a summer intern at Microsoft [9], and, especially, how they have used Tamarin to break the EMV protocol [6] and the 5G standard [23]. This guest lecture has been mentioned by some students as a highlight of the course.

System Requirements (1 lecture). I devote one lecture to introduce "system requirements" informally. What are invariants, eventually, until, and response properties? I explain how to analyze invariants by searching for bad states, and how to inductively prove (by hand) that something is an invariant for *all* initial states. I also discuss state-based versus action-based requirements, and various kinds of fairness assumptions needed to prove "eventually" properties.

Formalizing and Checking System Requirements Using Temporal Logic (1 lecture). I introduce linear temporal logic (LTL) and the use of Maude's

LTL model checker to formalize and then model check system requirements. We then have a wealth of examples to model check.

Example 8. Consider the token-ring mutual exclusion algorithm in Example 7. 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:[4]

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

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

The property does not hold: the model checker returns a counterexample where node `d` never wants 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
```

□

[4] '`[]`' and '`<>`' denote the temporal operators □ and ◇, respectively, and '`/\` and '`->`' denote logical conjunction and implication.

I have also proved by LTL model checking that all philosophers are guaranteed to eat infinitely often in one of the solutions to the dining philosophers problem. This required formalizing a number of fairness assumptions.

I briefly mention other logics like CTL and CTL*, LTL with past temporal operators, and Meseguer's *temporal logic of rewriting*, which allows us to reason about both state-based and action-based properties.

I included temporal logic for second-year students with trepidity. This is a completely new kind of logic for the students, which should require time and maturity to understand. I am pleasantly surprised that the students seem to master temporal logic with only one lecture: their exam solutions show that they understand temporal logic formulas and can judge whether such a formula holds in a model.

Real-Time and Probabilistic Systems (not taught). Up to this point, the models have been *untimed*. However, 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. The course textbook introduces how real-time systems can be modeled and analyzed in Maude, and also discusses timed extensions of temporal logics.

Randomized simulations, such that those performed simulating playing *black-jack* 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 [26], 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.

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 [7] 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 end 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. 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" [17] 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.

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

6 Evaluation

That I have worked hard on designing what I think *should* be a good and accessible introduction to formal methods by using Maude does not help much if the students disagree. The all-important question is therefore: What do the students think? Unfortunately, I have not solicited their feedback. Instead, the students have the possibility to provide feedback anonymously on courses signed up for. Most students do not bother to do this. Therefore, although I am trying to summarize the students' experiences the best I can, this evaluation is unscientific, anecdotal, and may suffer from selection bias.

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 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?" 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. Furthermore, since 2020 and 2021 were destroyed by/taught online due to Covid-19, I also separate out the results from those years. In particular, I believe, again without evidence, that the lack of (physical) lectures that make the curriculum understandable is a larger problem for harder-to-access theoretical courses than the more "practical" courses that students usually take. Or is Covid-19 just a convenient scapegoat for the 2020–2021 feedback?

How do you rate this course in general?			
	2007–2019	2019	2020–2021
Exceptionally good	15	4	8
Very good	23	3	4
Good	8	0	6
OK (neither good nor bad)	6	1	2
Not that good	1	0	3
Not good	0	0	0

Difficulty/level of the course			
	2007–2019	2019	2020–2021
Too difficult	1	0	2
Somewhat difficult	38	4	18
OK/Average	38	4	3
Easy	0	0	0
Too easy	0	0	0

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

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 2021. 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."
- "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."
- "One of the best of the ten courses I have taken at the department."

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."
- "Maude is very frustrating because of bad or (in Full Maude) missing error messages."
- "The theory part was more difficult than the rewrite rules part."
- "Lectures crashed with the lectures in a more "important" course."
- "Difficult. Unnecessary. Unnecessarily complicated language. Irrelevant."

Main complaints concern Full Maude and its "peculiarities" (lack of robustness and good error messages) and that there are too few resources about Maude. Finally, as expected, a number of students do not understand why they need to learn a programming language that is not widely used. When teaching, we have to emphasize again and again that we use a convenient language to teach and illustrate *general* formal methods principles, so that you could easily work with more "industrial" tools, like TLA+, after taking this course.

7 Concluding Remarks

In this paper I surveyed a few papers on, and distilled some requirements for, teaching formal methods. I claimed that rewriting logic and Maude provide an ideal framework that seems to satisfy these criteria. I have given an overview of the topics I cover in my second-year course and in its accompanying textbook. Finally, I summarized the feedback that students provide anonymously to the university. I end this paper by trying to address some obvious questions, and by making a suggestion to the organizers of WRLA 2024.

Is Maude really a suitable framework for introducing formal methods to second-year undergraduate students? This is really a two-pronged question: is Maude a good tool for teaching formal methods, and is the second year too early to introduce formal methods?

Concerning the first question, I still think that Maude is a great choice, as I argue in Sect. 4. It provides an intuitive functional programming style, which I think students enjoy. Furthermore, despite taking in a lot of well-established term

rewriting theory, we still manage to model and analyze fundamental distributed algorithms and protocols, such as sliding windows, all those distributed mutual exclusion, algorithms for distributed transaction systems, and also cryptographic protocols like NSPK, which is very easy to model and analyze, even for students. Maude also encourages us to teach temporal logic, which is quintessential for formalizing requirements of distributed systems. The by far main problem is that I teach object-oriented modeling using Full Maude. The lack of (useful) error messages in Full Maude understandably frustrates students, and takes away the pleasures of modeling in Maude. It might well be a mistake to use Full Maude for modeling object-based distributed systems; even Francisco Durán teaches object-based modeling using (core) Maude. (Core) Maude will supposedly provide support for object-based specification in the near future; that would make my course *much better* for the students, and cannot happen soon enough.

Regarding the second question, I have no answer or good methodology to answer it. Results from the first exam for second-year students were encouraging, and student feedback has been as positive as in previous years. However, it is hard to conclude anything from exam results and other feedback in 2020 and 2021, because of Covid-19. As usual, many students quit the course during the semester. However, I am not sure that one can gain much insight from this. It is common in Oslo, since signing up for classes is free, so students "try out" many courses. Furthermore, competing against a (supposedly good) introductory course on operating systems and networks for the only optional slot in the Bachelor program is challenging.

Is the course a success? This is the million-dollar question, and, again, the jury is still out on this one. I believe that it is fair to say that student evaluations generally are positive. Is this due to the topics covered and the textbook, or does the quality of the lectures and exercise seminars also play a role? Furthermore, just a small fraction of the students reply to these surveys, with a possible selection bias. Eventually, the proof is in the pudding, as they say: do the students take the course? In a related paper [19] from 2020, I wrote that the course—due to its precarious place in the Bachelor program—crucially relies on word-of-mouth recommendation by other students. Then the trend looked good, going from the usual 15–20 students to 42 students who took the exam in 2020. But with two years of Covid19-induced closure of the university, the interaction between students has essentially been non-existent, removing the potential for word-of-mouth recommendation. So we are back where we were before: between 15 and 20 student will take the exam this year.

Is my way the right way to teach Maude to undergraduates? If we want to teach Maude, is the way I do it the right way? Based on general evaluation and what I hear from students, it is tempting to significantly reduce the material on classic term rewriting and equational logic theory, which takes almost half the lectures. One could then add more Part II stuff: more fundamental distributed systems, and/or real-time and probabilistic systems. Maybe meta-programming? Strategies? Programming "web applications" with Maude's support for external

objects, e.g., via sockets and file systems? Or develop Maude semantics for simple multi-threaded imperative programming languages, which I think would be fun for the students.

Should I remove this theory? I like the theory on termination, but cannot convey as much enthusiasm for confluence; and students are not always enamored of equational logic and inductive theorems either. Can I drop the theory and make the course even more "practical"? If I drop confluence then also the equational logic part will suffer; furthermore, Maude requires your specifications to be confluent, so students should know about this. What should I do? If you, dear reader, teach Maude or related methods, I would love to hear your opinion and experiences. I would also love to know of interesting distributed systems that could be included in the course, or given as exam problems.

A suggestion to the organizers of WRLA 2024. With Maude now a mature tool with an impressive range of applications, it should be ripe for teaching formal methods. I think that multiple groups around the world are using Maude in teaching (mostly at the graduate level?). It would be enormously important for our community to know about each other's experiences, curricula, and ways of teaching Maude-based courses. I would therefore like to wrap up this WRLA 2022 "invited experience report" by proposing that the organizer of WRLA 2024 organize a special session on using Maude for teaching, where we can share our experiences on this important topic.

Acknowledgments. I would like to thank Kyungmin Bae for inviting me to give an invited talk at WRLA 2022, and for patiently waiting for this paper to be finished.

References

1. Aceto, L., Ingolfsdottir, 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. Basin, D.A., Cremers, C., Dreier, J., Sasse, R.: Tamarin: verification of large-scale, real-world, cryptographic protocols. IEEE Secur. Priv. **20**(3), 24–32 (2022)
5. Basin, D.A., Sasse, R., Toro-Pozo, J.: Card brand mixup attack: bypassing the PIN in non-Visa cards by using them for Visa transactions. In: 30th USENIX Security Symposium, USENIX Security 2021, pp. 179–194. USENIX Association (2021)
6. Basin, D.A., Sasse, R., Toro-Pozo, J.: The EMV standard: break, fix, verify. In: 42nd IEEE Symposium on Security and Privacy, SP 2021. IEEE (2021)

7. 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)
8. Cerone, A., et al.: Rooting formal methods within higher education curricula for computer science and software engineering: a white paper. In: Cerone, A., Roggenbach, M. (eds.) FMFun 2019. CCIS, vol. 1301, pp. 1–26. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-71374-4_1
9. Chen, S., Meseguer, J., Sasse, R., Wang, H.J., Wang, Y.M.: A systematic approach to uncover security flaws in GUI logic. In: IEEE Symposium on Security and Privacy, pp. 71–85. IEEE Computer Society (2007)
10. Clavel, M., et al.: All About Maude. LNCS, vol. 4350. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-71999-1
11. Escobar, S., Meadows, C.A., Meseguer, J.: Maude-NPA: cryptographic protocol analysis modulo equational properties. In: Aldini, A., Barthe, G., Gorrieri, R. (eds.) FOSAD 2007/2008/2009. LNCS, vol. 5705, pp. 1–50. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-03829-7_1
12. Krings, S., Körner, P.: Prototyping games using formal methods. In: Cerone, A., Roggenbach, M. (eds.) FMFun 2019. CCIS, vol. 1301, pp. 124–142. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-71374-4_6
13. 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)
14. Meseguer, J.: Conditional rewriting logic as a unified model of concurrency. Theor. Comput. Sci. **96**, 73–155 (1992)
15. Meseguer, J., Rosu, G.: The rewriting logic semantics project. Theor. Comput. Sci. **373**(3), 213–237 (2007)
16. Meseguer, J., Roşu, G.: The rewriting logic semantics project: a progress report. Inf. Comput. **231**, 38–69 (2013)
17. 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)
18. 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
19. Ölveczky, P.C.: Teaching formal methods for fun using Maude. In: Cerone, A., Roggenbach, M. (eds.) FMFun 2019. CCIS, vol. 1301, pp. 58–91. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-71374-4_3
20. Ölveczky, P.C., Meseguer, J.: The Real-Time Maude tool. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 332–336. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_23
21. Ölveczky, P.C.: Designing Reliable Distributed Systems: A Formal Methods Approach Based on Executable Modeling in Maude. UTCS, Springer, London (2017). https://doi.org/10.1007/978-1-4471-6687-0
22. Park, D., Zhang, Y., Saxena, M., Daian, P., Roşu, G.: A formal verification tool for Ethereum VM bytecode. In: Proceedings of the ESEC/FSE 2018, pp. 912–915. ACM (2018)
23. Peltonen, A., Sasse, R., Basin, D.A.: A comprehensive formal analysis of 5G handover. In: 14th ACM Conference on Security and Privacy in Wireless and Mobile Networks, WiSec 2021, pp. 1–12. ACM (2021)
24. Roşu, G., Şerbănuţă, T.F.: An overview of the K semantic framework. J. Logic Algebraic Program. **79**(6), 397–434 (2010)

25. Schwartz, D.G.: Rethinking the CS curriculum. Blog at the Communications of the ACM, May 2022. https://cacm.acm.org/blogs/blog-cacm/261380-rethinking-the-cs-curriculum/fulltext
26. Sebastio, S., Vandin, A.: MultiVeStA: statistical model checking for discrete event simulators. In: ValueTools, pp. 310–315. ICST/ACM (2013)
27. Vardi, M.Y.: Branching vs. linear time: final showdown. In: Margaria, T., Yi, W. (eds.) TACAS 2001. LNCS, vol. 2031, pp. 1–22. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-45319-9_1