

Teaching Concurrency: Theory in Practice[★]

Luca Aceto¹, Anna Ingólfssdóttir¹, Kim G. Larsen², and Jiří Srba²

¹ School of Computer Science, Reykjavik University, Kringlan 1, 103 Reykjavik, Iceland

² Department of Computer Science, Aalborg University, Selma Lagerlöfs Vej 300, 9220 Aalborg Ø, Denmark

Abstract. Teaching courses that rely on sound mathematical principles is nowadays a challenging task at many universities. On the one hand there is an increased demand for educating students in these areas, on the other hand there are more and more students being accepted with less adequate skills in mathematics. We report here on our experiences in teaching concurrency theory over the last twenty years or so to students ranging from mathophobic bachelor students to sophisticated doctoral students. The contents of the courses, the material on which they are based and the pedagogical philosophy underlying them are described, as well as some of the lessons that we have learned over the years.

1 Introduction and background

We report on our experiences in teaching concurrency theory, as well as principles of modelling and verification for reactive systems [18]. Some of us have been teaching such courses for about twenty years now and the underlying philosophy in our teaching has not changed much over this time span. However, the structure of the courses we shall discuss in what follows has naturally evolved over time to reflect the scientific developments in our subject matter and has been adopted in lecture series that have mostly been held at Aalborg University and Reykjavík University over the last seven-eight years. Our teaching experience and the structure, contents and pedagogical philosophy of our courses form the basis for a textbook we published in 2007 [1] and for the supplementary material that accompanies the book (see <http://www.cs.aau.dk/rsbook> for an on-line collection of further sources). So, the experiences we report in this article are based on the teaching material accessible to a wide variety of students at various stages of their academic development and with differing levels of mathematical background and maturity.

The aim of the above-mentioned semester-long courses was to introduce students at the early stage of their M.Sc. degrees, or late in their B.Sc. degree

[★] The work of Aceto and Ingólfssdóttir has been partially supported by the projects “The Equational Logic of Parallel Processes” (nr. 060013021) and “New Developments in Operational Semantics” (nr. 080039021) of the Icelandic Research Fund. Srba was partially supported by Ministry of Education of the Czech Republic, project No. MSM 0021622419.

studies, in computer science to the theory of concurrency, and to its applications in the modelling and analysis of reactive systems. This is an area of formal methods that is finding increasing application outside academic circles and allows the students to appreciate how techniques and software tools based on sound theoretical principles are very useful in the design and analysis of non-trivial reactive computing systems. (As we shall discuss later in Section 2, we have also taught intensive three-week courses based on our course material as well as given lectures series intended for Ph.D. students.)

In order to carry this message across to the students in the most effective way, the courses present

- some of the prime models used in the theory of concurrency (with special emphasis on state-transition models of computation like labelled transition systems [23] and timed automata [2]),
- languages for describing actual systems and their specifications (with focus on classic algebraic process calculi like Milner’s Calculus of Communicating Systems [30] and logics like modal and temporal logics [11, 20, 34]), and
- their embodiment in tools for the automatic verification of computing systems.

The use of the theory and the associated software tools in the modelling and analysis of computing systems is a rather important component in our courses since it gives the students hands-on experience in the application of what they have learned, and reinforces their belief that the theory they are studying is indeed useful and worth mastering. Once we have succeeded in awakening an interest in the theory of concurrency and its applications amongst our students, it will be more likely that at least some of them will decide to pursue a more in-depth study of the more advanced, and mathematically sophisticated, aspects of our field—for instance, during their M.Sc. thesis work or at a doctoral level.

It has been very satisfying for us to witness a change of attitudes in the students taking our courses over the years. Indeed, we have gone from a state in which most of the students saw very little point in taking the course on which this material is based, to one in which the relevance of the material we cover is uncontroversial to most of them! At the time when an early version of our course was elective at Aalborg University, and taken only by a few mathematically inclined individuals, one of our students remarked in his course evaluation form that ‘This course ought to be mandatory for computer science students.’ Now the course is mandatory, it is attended by all of the M.Sc. students in computer science at Aalborg University, and most of them happily play with the theory and tools we introduce in the course.

How did this change in attitude come about? And why do we believe that this is an important change? In order to answer these questions, it might be best to describe first the general area of computer science to which our courses and textbook aim at contributing. This description is also based on what we tell our students at the beginning, and during, our courses to provide them with the context within which to place the material they learn in the course and with the initial motivation to work with the theory we set about teaching them.

The correctness problem and its importance. Computer scientists build artifacts (implemented in hardware, software or, as is the case in the fast-growing area of embedded and interactive systems, using a combination of both) that are supposed to offer some well defined services to their users. Since these computing systems are deployed in very large numbers, and often control crucial, if not safety critical, industrial processes, it is vital that they correctly implement the specification of their intended behaviour. The problem of ascertaining whether a computing system does indeed offer the behaviour described by its specification is called the *correctness problem*, and is one of the most fundamental problems in computer science. The field of computer science that studies languages for the description of (models of) computer systems and their specifications, and (possibly automated) methods for establishing the correctness of systems with respect to their specifications is called *algorithmic verification*.

Despite their fundamental scientific and practical importance, however, twentieth century computer and communication technology has not paid sufficient attention to issues related to correctness and dependability of systems in its drive toward faster and cheaper products. (See the editorial [33] by David Patterson, former president of the ACM, for forceful arguments to this effect.) As a result, system crashes are commonplace, sometimes leading to very costly, when not altogether spectacular, system failures like Intel's Pentium-II bug in the floating-point division unit [35] and the crash of the Ariane-5 rocket due to a conversion of a 64-bit real number to a 16-bit integer [27].

Classic engineering disciplines have a time-honoured and effective approach to building artifacts that meet their intended specifications: before actually constructing the artifacts, engineers develop models of the design to be built and subject them to a thorough analysis. Surprisingly, such an approach has only recently been used extensively in the development of computing systems.

Our textbook, and the courses we have given over the years based on the material it presents, stem from our deep conviction that each well educated twenty-first century computer scientist should be well versed in the technology of algorithmic, model-based verification. Indeed, as recent advances in algorithmic verification and applications of model checking [12] have shown, the tools and ideas developed within these fields can be used to analyze designs of considerable complexity that, until a few years ago, were thought to be intractable using formal analysis and modelling tools. (Companies such as AT&T, Cadence, Fujitsu, HP, IBM, Intel, Microsoft, Motorola, NEC, Siemens and Sun—to mention but a few—are using these tools increasingly on their own designs to reduce time to market and ensure product quality.)

We believe that the availability of automatic software tools for model-based analysis of systems is one of the two main factors behind the increasing interest amongst students and practitioners alike in model-based verification technology. Another is the realization that even small reactive systems—for instance, relatively short concurrent algorithms—exhibit very complex behaviours due to their interactive nature. Unlike in the setting of sequential software, it is therefore not hard for the students to realize that systematic and formal analysis techniques

are useful, when not altogether necessary, to obtain some level of confidence in the correctness of our designs. The tool support that is now available to explore the behaviour of models of systems expressed as collections of interacting state machines of some sort makes the theory presented in our courses appealing for many students at several levels of their studies.

It is our firmly held belief that only by teaching the theory of concurrent systems, together with its applications and associated verification tools, to our students, we shall be able to transfer the available technology to industry, and improve the reliability of embedded software and other reactive systems. We hope that our textbook and the teaching resources that accompany it, as well as the experience report provided in this article will offer a small contribution to this pedagogical endeavour.

Historical remarks and acknowledgments. As already stated earlier, we have used the material covered in [1] and discussed in this article in the present form for courses given at several institutions during the last seven-eight years. However, the story of its developments is much older, and goes back at least to 1986. During that year, the third author (Kim G. Larsen, then a freshly minted Ph.D. graduate from Edinburgh University) took up an academic position at Aalborg University. He immediately began designing a course on the theory of concurrency—the branch of theoretical computer science that he had worked on during his doctoral studies under the supervision of Robin Milner. His aim was to use the course, and the accompanying set of notes and slides, to attract students to his research area by conveying his enthusiasm for it, as well as his belief that the theory of concurrency is important in applications. That material and the pedagogical style he adopted have stood the ‘lecture room test’ well, and still form the basis for the way we organize the teaching of the part on classic reactive systems in our courses.

The development of those early courses was strongly influenced by Robin Milner’s teaching and supervision that Kim G. Larsen enjoyed during his doctoral studies in Edinburgh, and would not have been possible without them. Even though the other three authors were not students of Milner’s themselves, the strong intellectual influence of his work and writings on their view of concurrency theory will be evident to the readers of this book. Indeed, the ‘Edinburgh concurrency theory school’ features prominently in the academic genealogy of each of the authors. For example, Rocco De Nicola and Matthew Hennessy had a strong influence on the view of concurrency theory and the work of Luca Aceto and/or Anna Ingólfssdóttir, and Jiri Srba enjoyed the liberal supervision of Mogens Nielsen.

The material upon which the courses we have held at Aalborg University and elsewhere since the late 1980s were based has undergone gradual changes before reaching the present form. Over the years, the part of the course devoted to Milner’s Calculus of Communicating Systems and its underlying theory has decreased, and so has the emphasis on some topics of mostly theoretical interest. At the same time, the course material has grown to include models and specification languages for real-time systems. The courses we deliver now aim at

offering a good balance between classic and real-time systems, and between the theory and its applications.

Overall, as already stated above, the students' appreciation of the theoretical material covered here has been greatly increased by the availability of software tools based on it. We thank all of the developers of the tools we use in our teaching; their work has made our subject matter come alive for our students, and has been instrumental in achieving whatever level of success we might have in our teaching.

Road map of the paper. The paper is organized as follows. We begin by describing the material we teach students in our courses as well as the types of students we have taught over the years (Section 2). We then present the organization of our courses and what role each component plays in the understanding of students (Section 3). Section 4 is devoted to the role that software tools play in our teaching. Next we introduce a possible syllabus and exam form for a one-semester course (Section 5). The proposed course and exam skeletons have served us well in our teaching over the years. Some concluding remarks are offered in Section 6.

2 What do we teach and to whom?

When planning a course on a topic that is closely related to one's research interests, one is greatly tempted to cover a substantial body of fairly advanced material. Indeed, earlier editions of our courses presented rather challenging results and mathematically sophisticated techniques from books such as Milner's classic monograph [30]. Our teaching experience over the years, however, has taught us that, when presenting topics from concurrency theory to today's students at bachelor and master level, we achieve best results by following the golden rule that

Less is more!

This is particularly true when we address bachelor students or teach intensive versions of our courses. In those situations, it becomes imperative to present a tightly knit body of theoretical material, exercises and projects that are designed in order to convey repeatedly a few main messages. Therefore, our courses do *not* aim at giving broad overviews of many of the available formalisms for describing and reasoning about reactive systems. We have instead selected a basic line of narrative that is repeated for classic reactive systems and for real-time systems. Our story line has been developed over the years with the specific aim to introduce in an accessible, yet suitably formal way, three notions that we use to describe, specify and analyze reactive systems, namely

- languages for the description of reactive systems as well as their underlying semantic models,
- behavioural relations giving the formal yardstick for arguing about correctness in the single-language approach, and

- logics for expressing properties of reactive systems following the model-checking approach to the correctness problem and their connections with behavioural relations.

Of course, when planning a course based on the above-mentioned narrative, we are faced with a large array of possible choices of languages, models and logics. Our personal choice, which is based on our own background, personal tastes and research interests, is reflected in the topics covered in our courses and in the textbook [1].

As mentioned above, we typically divide our courses into two closely knit parts. The first part of the course deals with classic models for reactive systems, while the second presents a theory of real-time systems. In both parts we focus on the three main themes mentioned above and stress the importance of formal models of computing devices, the different approaches that one can use to specify their intended behaviour and the techniques and software tools that are available for the (automatic) verification of their correctness.

In the setting of classic reactive systems, we typically present

- Milner’s Calculus of Communicating Systems (CCS) [30] and its operational semantics in terms of the model of Labelled Transition Systems (LTSs) [23],
- the crucial concept of bisimilarity [32, 30] and
- Hennessy-Milner Logic (HML) [20] and its extension with recursive definitions of formulae [26].

In the second part of the course, we usually introduce a similar trinity of basic notions that allows us to describe, specify and analyze real-time systems—that is, systems whose behaviour depends crucially on timing constraints. There we present

- the formalisms of timed automata [2] and Timed CCS [44–46] to describe real-time systems and their semantics in terms of the model of timed labelled transition systems,
- notions of timed and untimed bisimilarity, and
- a real-time version of Hennessy-Milner Logic [25].

After having worked through the material in our courses, our students will be able to describe non-trivial reactive systems and their specifications using the aforementioned models, and verify the correctness of a model of a system with respect to given specifications either manually or by using automatic verification tools like the Edinburgh Concurrency Workbench (CWB)³ [13], the Concurrency Workbench of the New Century (CWB-NC)⁴ [14] and the model checker for real-time systems UPPAAL⁵ [8].

Our, somewhat ambitious, aim is therefore to present a model of reactive systems that supports their design, specification and verification. Moreover, one

³ <http://www.dcs.ed.ac.uk/home/cwb/>

⁴ <http://www.cs.sunysb.edu/~cwb/>

⁵ <http://www.uppaal.com/>

of the messages that we reiterate throughout the course is that, since many real-life systems are hard to analyze manually, we should like to have computer support for our verification tasks. This means that all the models and languages that we introduce in our courses need to have a *formal* syntax and semantics.

The level of detail and formality that we adopt in teaching courses based on the above-mentioned material depends both on the level of mathematical maturity of the students and on the length of the course. However, we feel that our experience strongly indicates that the contents of our courses lends itself to presentations at widely different levels of ‘mathematical depth’. The resulting types of courses are all based on the following main messages that serve as refrains in our narrative.

1. Formal models of computing systems can be developed using very expressive and flexible, but mathematically rather simple, formalisms.
2. These models are executable and can serve both as descriptions of actual implementations of systems and as their specifications. Therefore a fundamental component of their theory and practice is a notion of equivalence or approximation between objects in the model. Such a notion of equivalence or approximation may be used as a formal yardstick for establishing the correctness of systems.
3. Modal and temporal logics play a fundamental role in the specification of (un)desirable properties of reactive systems and are the cornerstone of the model-checking approach to the correctness problem.
4. All of the above ingredients are embodied in software tools for the automatic verification of computing systems.

In an intensive three-week course taken jointly by bachelor and master students in computer science and software engineering, we cannot hope to cover much of the material presented in [1]. We therefore focus on the very basic topics that we believe the students must understand in order to use the available models, techniques and tools in a conscious way when working on the projects we set them. In such courses, when introducing, for instance, the theory of classic reactive systems covered in the first part of [1], we eschew many of the mathematical details presented in the book and limit ourselves to describing the model of labelled transition systems, CCS and its operational semantics, trace equivalence, the definition of strong and weak bisimilarity and the proof technique it supports, HML and its extension with very simple recursive definitions as well as its connection with bisimilarity and with branching-time temporal logics. Knowing the syntax and basic semantics of CCS allows the students to describe their models in a way that can be used as input for tools such as the CWB and the CWB-NC. Familiarity with trace equivalence and bisimilarity gives the students enough knowledge to formulate and establish correctness requirements using equivalence checking. Moreover, the basic knowledge of HML and its recursive extension they develop is sufficient to interpret the debugging information they receive from the tools when equivalence checking tests fail and to formulate fairly sophisticated correctness requirements in logical terms.

At the other end of the spectrum are courses delivered to M.Sc. or Ph.D. students who specialize in topics related to concurrency theory. Those students can instead be taught the unified theory underlying the material presented in [1] by complementing the applications of the theory and practical assignments with a careful presentation of the main theorems and their proofs, as well as of the mathematics that underlies the algorithmics of concurrency. When teaching a course to such students, we typically cover the theory of fixed-points of endofunctions over complete lattices, culminating in a proof of Tarski's fixed-point theorem [41], the view of bisimilarity as a largest fixed-point and characteristic formula constructions for bisimilarity [22, 38].

The teaching of the semantics of recursive extensions of HML to either kind of students is a challenging pedagogical exercise. Rather than presenting the general theory underlying the semantics of fixed-point logics in all its glory and details, we prefer to pay heed to the following advice, quoted in [24]:

Only wimps do the general case. Real teachers tackle examples.

Following the prime role played by examples in our teaching, we essentially teach students how to compute the collection of states in a finite labelled transition system that satisfy least or largest fixed-point formulae on a variety of examples. At the same time, we make the students define recursive formulae that intuitively express some properties of interest and have them check whether the formulae are correct by performing the iterative computations to calculate their semantics over well chosen labelled transition systems. This is essentially an 'experimental' approach to teaching students how to make sense of fixed points in the definition of temporal properties of concurrent systems. In our experience, after having worked through a good number of examples, and having solved and discussed the solutions to the exercises we give them, the students are able to express basic, but important, temporal properties such as

the system can deadlock

or

it is always the case that a message that is sent will eventually be delivered.

Note that a specification of the latter property involves the use of both largest and least fixed-point formulae. We consider it a great success that students are able to define such formulae and to convince themselves that they have the expected meaning.

Overall, the story line in our courses lends itself to delivery at different levels of formality, provided we stress throughout the course the connections between the mathematical theories we present and their practical applications and meaning. In our experience, if the lecturers provide sufficient context for the formal material, the students do respond by accepting the necessary mathematics within the limits of their abilities. Having said so, teaching formally based techniques to present-day bachelor and master students in computer science does present challenges that any working university lecturer knows very well. The material

we cover does have a strong relevance for the practice of computing, but it involves the ‘M’ word, viz. Mathematics. This means that many students have the preconceived idea that the material we plan to teach them is beyond their abilities. In order to make the material more palatable to students and overcome this psychological obstacle to its acceptance by our audience, we have developed a narrative that uses anthropomorphic examples and, e.g., fairly recent theoretical results on process equivalences based on games that, at least according to our own extensive but admittedly biased experience, does help the students in understanding difficult notions like bisimilarity. Our lecturing style and the structure of our courses will be described in slightly more detail in Section 3. Here we limit ourselves to mentioning that, in our experience, the game characterization of bisimilarity (see, e.g., [39, 42]) helps students at all levels understand the fundamental concept of bisimilarity and its difference from the simulation preorder and the equivalence induced by the latter. All our students like to play games, be they computer games or board games, and they all seem to understand the basic rules of the bisimulation game and to play the game without too many problems. Using the game, the students can often argue convincingly when two processes are not bisimilar, also in cases when they would have trouble using the relational definition of bisimilarity. In fact, we have even had students who can use the game characterization of bisimilarity, but who do not know precisely what a relation, an equivalence relation or relation composition are. We are not sure about what this lack of background says about the teaching of discrete mathematics, but it does seem to call for providing more context in those courses for the introduction of these fundamental mathematical concepts and their uses in computer science.

A similar game characterization is also used for arguing about the meaning of recursive HML formulae. This offers an alternative approach to the example-based explanation of recursive formulae and provides a different view on understanding largest and least fixed-point properties.

3 How do we teach concurrency theory?

As we mentioned in the previous section, we have used much of the material presented in our textbook [1] in several one semester courses at Aalborg University and at Reykjavík University, amongst others. These courses usually consist of about thirty hours of lectures and a similar number of hours of exercise sessions, where the students solve exercises and work on projects related to the material in the course. As we stated earlier, we strongly believe that these practical sessions play a very important role in making the students appreciate the importance of the theory they are learning, and understand it in depth. The importance that we attach to practical sessions is also reflected by the fact that we devote just as much time to them as to the lectures themselves. Indeed, we usually devote *more* time to hands-on tutorial sessions than to the lectures since two or more lecture slots are typically devoted to work on the mini-projects we set the students.

During the exercise sessions and the mini-projects that we usually set during each installment of our courses, the students work in groups that typically consist of two or three members. The groups of students are supposed to work independently on the solutions to the exercises and to engage in peer instruction [29], at least ‘in the small’. The teaching assistants and we discuss the solutions with the students, ask them further ‘what if’ questions that arise from their purported answers to the exercises, spur the students to question their proposed answers to the exercises and use the results of the exercise sessions to find out what topics need to be further clarified in the lecture room. We always post solutions to selected exercises after each exercise session. This allows the students to check whether the answers they proposed are in agreement with ours and whether they are convinced by the model solutions we provide. Bone fide instructors can obtain the material we typically use for the exercise sessions, as well as the model solutions we distribute, by emailing us at `rsbook@cs.aau.dk`.

As mentioned above, apart from the standard exercise sessions, students taking our courses usually work on two group projects. For each such project, the students receive six supervised work hours. The aim of the projects is to give the students experience in modelling a reasonably non-trivial scenario and in analyzing their models using one of the software tools for computer-aided verification introduced during the course.

The first project usually deals with a modelling and verification task in ‘classic concurrency theory’; the students use Milner’s CCS as a modelling language and Hennessy-Milner logic with recursive definitions and/or CCS itself as a specification language. They then verify their models employing either the CWB or the CWB-NC to perform equivalence checking and/or model checking as appropriate. In fact, we often ask our students to verify properties of their model using *both* equivalence checking and model checking. The rationale for this choice is that we believe that students should be familiar with both approaches to verification since this trains them in selecting the approach that is best suited for the task at hand.

Examples of projects that we have repeatedly used over the years include modelling and analysis of

- basic communication protocols such as the classic Alternating Bit Protocol [7] or the CSMA (Carrier Sense Multi Access) Protocol,
- various mutual exclusion algorithms using CCS and Hennessy-Milner logic with recursion [43],
- the solitaire game presented in [5, Chapter 6].

A useful source for many interesting student projects is the book [15], which presents semaphore-based solutions to many concurrency problems, ranging from classic ones (like the barbershop) to more exotic problems (like the Sushi bar). Indeed, as cogently argued in [17], not only topics in classic concurrency control from courses in, say, operating systems can be fruitfully used as student projects to provide context for the material covered in concurrency-theory courses, but model checkers and other software tools developed within the concurrency-theory

community can be employed to make the material typically taught in operating systems course come alive for the students.

The second project usually deals with a modelling and verification task involving real-time aspects, at least in part; the students use networks of timed automata [2] as a modelling language and the query language supported by the tool Uppaal as a specification language. They then verify their models employing Uppaal to perform model checking, synthesize schedules or strategies to win puzzles or games as appropriate. Examples of projects that we have repeatedly used over the years include modelling and analysis of

- the board game Rush Hour,
- the gossiping girls puzzle,
- real-time mutual exclusion algorithms like those presented in, e.g., [3], and
- more problems listed on the web page at <http://rsbook.cs.aau.dk/index.php/Projects>.

Examples of recent courses based on the book [1] may be found at the URL

<http://www.cs.aau.dk/rsbook/>.

There the instructor will find suggested schedules for his/her courses, exercises that can be used to supplement those in our textbook [1], links to other useful teaching resources available on the web, further suggestions for student projects and electronic slides that can be used for the lectures. (As an example, we usually supplement lectures covering the material in [1] with a series of four-six 45 minute lectures on Binary Decision Diagrams [10] and their use in verification based on Henrik Reif Andersen's excellent lecture notes [4] that are freely available on the web and on Randel Bryant's survey paper [10].)

Our pedagogical style in teaching concurrency theory can be gleaned by looking at our own recent textbook on the subject [1]. This book is by no means the first one devoted to aspects of the theory of reactive systems. Some of the books that have been published in this area over the last twenty years or so are the references [6, 16, 19, 21, 28, 30, 36, 37, 40] to mention but a few. However, unlike all the aforementioned books but [16, 28, 37], ours was explicitly written to serve as a *textbook*, and offers a distinctive pedagogical approach to its subject matter that derives from our extensive use of the material presented there in book form in the classroom. In writing that textbook we have striven to transfer on paper the spirit of the lectures on which that text is based. Our readers will find that the style in which that book is written is often colloquial, and attempts to mimic the Socratic dialogue with which we try to entice our student audience to take active part in the lectures and associated exercise sessions. Explanations of the material presented in the textbook are interspersed with questions to our readers and exercises that invite the readers to check straight away whether they understand the material as it is being presented. This is precisely how we present the material in the classroom both during the lectures and the tutorial sessions. We engage the students in continuous intellectual table tennis so that they are enticed to work through the course material as it unfolds during the course

sessions, in some cases feeling that they are ‘discovering things themselves’ as the course progresses.

As we mentioned earlier, we have developed a collection of anthropomorphic examples that, we believe, make the introduction of key notions come alive for many of our students. By way of example, we mention here that we introduce the whole syntax of Milner’s CCS by telling the story of a computer scientist who wants to maximize her chances of obtaining tenure at a research university by gaining exclusive access to a coffee machine, which she needs to continue producing publications after having published her first one straight out of her thesis.

As another example of this approach, we describe the iterative algorithm for computing the set of processes satisfying largest fixed-point formulae in Hennessy-Milner logic with recursion by drawing a parallel with the workings of a court of law. Each process satisfies the formula (or ‘is innocent’) unless we can find a reason why it should not (that is, ‘unless it is proven to be guilty’). Apart from leading to memorable scientific theatre, we think that this analogy helps students appreciate and remember the main ideas behind the iterative algorithms better. Dually, when introducing the iterative algorithm for computing the set of processes satisfying least fixed-point formulae in Hennessy-Milner logic with recursion, we say that the key intuition behind the algorithms is that no process is ‘good’ (satisfies the formula) unless it is proven to be so.

These are simple, but we believe telling, examples of the efficiency of story telling in the teaching of computer science and mathematics, as put forward by Papadimitriou in [31]. The readers of our book [1] and of the further material available from the book’s web site will find other examples of the use of this pedagogical approach in our teaching and educational writings.

4 The role of software tools in our teaching

We strongly recommend that the teaching of concurrency theory be accompanied by the use of software tools for verification and validation. In our courses, we usually employ the Edinburgh Concurrency Workbench [13] and/or the Concurrency Workbench of the New Century [14] for the part of the course devoted to classic reactive systems and, not surprisingly, Uppaal [8] for the lectures on real-time systems. All these tools are freely available, and their use makes the theoretical material covered during the lectures come alive for the students. Using the tools, the students will be able to analyze systems of considerable complexity, and we suggest that courses based upon our book and the teaching philosophy described in this article be accompanied by two practical projects involving the use of these, or similar, tools for verification and validation.

We moreover recommend that the aforementioned tools be introduced as early as possible during the courses, preferably already at the moment when the students hear for the first time about the language CCS, and that the use of tools be integrated into the exercise sessions. For example, the students might be asked about the existence of a particular transition between two CCS expressions

and then proceed with verifying their answers by using a tool. This will build their confidence in their understanding of the theory as well as motivate them to learn more about the principles behind these tools.

In several of our recent courses, we have also used blogs as a way to entice the students to put their thoughts about the course material and their solutions to the exercises in writing. The rationale for this choice is that we feel that many of our students underestimate the importance of writing down their thoughts clearly and concisely, and of explaining them to others in writing. The use of blogs allows students to comment on each other's posts, thoughts and solutions. Moreover, in so doing, they learn how to exercise restraint in their criticisms and how to address their peers in a proper scientific debate. In order to encourage students to make use of course blogs, we have sometimes reserved a tiny part of the final mark of the course, say 5%, for activity on the course blog. Overall, students have made good use of the course blogs whenever we have asked them to do so. Despite the lack of conclusive data, we believe that the use of a blog or of similar software is beneficial in university level courses.

Finally, let us remark that we encourage the students taking our courses to experiment with 'The Bisimulation-Game Game'⁶. This tool, which has been developed by Martin Mosegaard and Claus Brabrand, allows our students to play the bisimulation game on their laptops and to experiment with the behaviour of processes written in Milner's CCS using the included graphical CCS visualizer and simulator. It would be interesting to obtain hard data measuring whether the use of such a graphical tool increases the students' understanding of the bisimulation game. We leave this topic for future investigations.

5 A possible syllabus and exam form

In this section we shall present a possible syllabus of the course which integrates two mini-projects. The assumption is that the course is given in 15 lectures, each lecture consisting of two 45 minute blocks.

- **Lecture 1: Labelled Transition Systems.** Introduction to the course; reactive systems and their modelling via labelled transition systems; CCS informally.
- **Lecture 2: CCS.** Formal definition of CCS syntax; SOS rules; number of examples; value passing CCS.
- **Lecture 3: Strong Bisimilarity.** Trace equivalence; motivation for behavioral equivalences; definition of strong bisimilarity; game characterization; examples and further properties of bisimilarity.
- **Lecture 4: Weak Bisimilarity.** Internal action τ ; definition of weak bisimilarity; game characterization; properties of weak bisimilarity; a small example of a communication protocol modelled in CCS and verified in CWB.
- **Lecture 5: Hennessy-Milner Logic.** Motivation; syntax and semantics of HML; examples in CWB; correspondence between strong bisimilarity and HML on image-finite transition systems.

⁶ <http://www.brics.dk/bisim/>

- **Lecture 6: Tarski’s Fixed Point Theorem.** Motivation via showing the need for introducing temporal properties into HML; complete lattices; Tarski’s fixed point theorem and its proof; computing fixed points on finite lattices.
- **Lecture 7: Hennessy-Milner Logic with Recursion.** Bisimulation as a fixed-point; one recursively defined variable in HML; game characterization; several recursively defined variables in HML.
- **Lecture 8: First Mini-Project.** Modelling and verification of Alternating Bit Protocol in CWB.
- **Lecture 9: Timed CCS.** Timed labelled transition systems; syntax and semantics of timed CCS; introduction to timed automata.
- **Lecture 10: Timed Automata.** Timed automata formally; networks of timed automata; timed and untimed bisimilarity; region construction.
- **Lecture 11: Timed Automata in UPPAAL.** UPPAAL essentials; practical examples; algorithms behind UPPAAL; zones.
- **Lecture 12: Second Mini-Project.** Modelling and verification of Rush Hour puzzle.
- **Lecture 13: Binary Decision Diagrams.** Boolean expressions; normal forms; Shannon’s expansion law; ordered and reduction binary decision diagrams; canonicity lemma; algorithms for manipulating binary decision diagrams.
- **Lecture 14: Applications of Binary Decision Diagrams.** Constraint solving; Boolean encoding of transition systems; bisimulation model checking; tool IBEN.
- **Lecture 15: Round-Up of the Course.** Overview of key concepts covered during the course; exam information.

We recommend that the lectures be accompanied by two hours of exercises, with a possible placement before the lectures so that exercises related to Lecture 1 are solved right before Lecture 2 and so on. We find the students’ active involvement in the exercise sessions crucial for the success of the course. To further motivate the students we use the technique of *constructive alignment* [9] so that the course objectives, the information communicated with the students during the lectures and exercises, as well as the exam content and form are aligned. This practically means that in each lecture we explicitly identify two or three main points that are essential for the understanding of the particular course topic, we exercise those points during the exercise sessions and then examine them at the exam. In order to be more explicit about the essential elements of the course, we mark in each exercise session one or two problems with a star to indicate their importance. During the exam (which is typically oral in our case, but can be easily adapted to a written one) the students pick up one of the star exercises (or their minor variants) and are then given 20 minutes to find a solution, while at the same time another student is being examined. At the exam start we ask the respective student to first present the solution to the chosen exercise and we proceed with the actual examination only if the exercise was answered at a satisfactory level.

After introducing the alignment technique described above we can report on a remarkable increase in students' involvement in the exercise sessions as well as in their understanding of the most essential notions covered during the course.

6 Concluding remarks

In this article, we have reported on our experiences in teaching concurrency theory over the last twenty years or so to a wide variety of students, ranging from mathophobic bachelor students to sophisticated doctoral students. We have described the contents of the courses, the material on which they are based and the pedagogical philosophy underlying them, as well as some of the lessons that we have learned over the years. Our main message is that concurrency theory *can* be taught with a, perhaps surprisingly, high degree of success to many different types of students provided that courses present a closely knit body of theoretical material, exercise and practical sessions, coupled with the introduction of software tools that the students use in modelling and verification projects. Our teaching experience over the years forms the basis for our textbook [1] and for the material that is available from its associated web site.

Of course, it is not up to us to determine how successful our pedagogical approach to teaching concurrency theory to many different types of students is. We can say, however, that several of the students taking our courses, who will not specialize within formal methods, appreciate the course as they often have the opportunity of applying the methods and tools we introduce in their projects on the practical construction of various kinds of distributed systems—for instance, in order to better understand a particular communication network or embedded controller.

Also, several neighbouring departments—in particular, the Control Theory Department and the research group on hardware-software co-design at Aalborg University—have adopted the model-based approach and tool support advocated by our course, both in research and in teaching at several levels (including doctoral education).

Finally, let us mention that, to the best of our knowledge, our textbook and the teaching approach we have described in this paper have so far been adopted in at least 15 courses at universities in several European countries and in Israel. (See <http://rsbook.cs.aau.dk/index.php/Lectures>.)

Overall, the positive feedback that we have received from the colleagues of ours who have used our material and pedagogical approach in their teaching and from the students following our courses gives us some hope that we may be offering a small contribution to making students appreciate the beauty and usefulness of concurrency theory and of algorithmic, model-based verification.

References

1. L. Aceto, A. Ingólfssdóttir, K. G. Larsen, and J. Srba. *Reactive Systems: Modelling, Specification and Verification*. Cambridge University Press, August 2007.

2. R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Comput. Sci.*, 126(2):183–235, 25 Apr. 1994. Fundamental Study.
3. R. Alur and G. Taubenfeld. Fast timing-based algorithms. *Distributed Computing*, 10(1):1–10, 1996.
4. H. R. Andersen. An introduction to binary decision diagrams, 1998. Version of October 1997 with minor revisions April 1998. 36 pp. Available at <http://www.itu.dk/people/hra/notes-index.html>.
5. A. Arnold, D. Bégay, and P. Crubillé. *Construction and Analysis of Transition Systems Using MEC*, volume 3 of *AMAST Series in Computing*. World Scientific, 1994.
6. J. C. Baeten and P. Weijland. *Process Algebra*. Cambridge Tracts in Theoretical Computer Science 18. Cambridge University Press, 1990.
7. K. Bartlett, R. Scantlebury, and P. Wilkinson. A note on reliable full-duplex transmission over half-duplex links. *Commun. ACM*, 12:260–261, 1969.
8. G. Behrmann, A. David, and K. G. Larsen. A tutorial on UPPAAL. In M. Bernardo and F. Corradini, editors, *Formal Methods for the Design of Real-Time Systems, International School on Formal Methods for the Design of Computer, Communication and Software Systems, SFM-RT 2004, Bertinoro, Italy, September 13-18, 2004, Revised Lectures*, volume 3185 of *Lecture Notes in Computer Science*, pages 200–236. Springer-Verlag, 2004.
9. J. Biggs. *Teaching for quality learning at University*. Open University Press, 1999.
10. R. E. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Comput. Surv.*, 24(3):293–318, 1992.
11. E. Clarke and E. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In D. Kozen, editor, *Proceedings of the Workshop on Logics of Programs*, volume 131 of *Lecture Notes in Computer Science*, pages 52–71. Springer-Verlag, 1981.
12. E. Clarke, O. Gruenberg, and D. Peled. *Model Checking*. MIT Press, December 1999.
13. R. Cleaveland, J. Parrow, and B. Steffen. The concurrency workbench: A semantics-based tool for the verification of concurrent systems. *ACM Trans. Prog. Lang. Syst.*, 15(1):36–72, 1993.
14. R. Cleaveland and S. Sims. The NCSU Concurrency Workbench. In R. Alur and T. Henzinger, editors, *Proceedings of the 8th International Conference Computer Aided Verification*, New Brunswick, NJ, U.S.A., July/August 1996, volume 1102 of *Lecture Notes in Computer Science*, pages 394–397. Springer-Verlag, 1996.
15. A. B. Downey. *The Little Book of Semaphores*. Green Tea Press, second edition, 2008. Available at <http://www.greenteapress.com/semaphores/>.
16. W. Fokkink. *Introduction to Process Algebra*. Texts in Theoretical Computer Science. An EATCS Series. Springer-Verlag, Berlin, 2000.
17. R. Hamberg and F. Vaandrager. Using model checkers in an introductory course on operating systems. *Operating Systems Review*, 42(6):101–111, 2008.
18. D. Harel and A. Pnueli. On the development of reactive systems. In *Logics and models of concurrent systems (La Colle-sur-Loup, 1984)*, volume 13 of *NATO Adv. Sci. Inst. Ser. F Comput. Systems Sci.*, pages 477–498. Springer-Verlag, Berlin, 1985.
19. M. Hennessy. *Algebraic Theory of Processes*. MIT Press, Cambridge, Massachusetts, 1988.
20. M. Hennessy and R. Milner. Algebraic laws for nondeterminism and concurrency. *J. ACM*, 32(1):137–161, 1985.

21. C. Hoare. *Communicating Sequential Processes*. Prentice-Hall International, Englewood Cliffs, 1985.
22. A. Ingólfssdóttir, J. C. Godskesen, and M. Zeeberg. Fra Hennessy-Milner logik til CCS-processer. Master's thesis, Department of Computer Science, Aalborg University, 1987. In Danish.
23. R. Keller. Formal verification of parallel programs. *Commun. ACM*, 19(7):371–384, 1976.
24. S. G. Krantz. *How to Teach Mathematics (a personal perspective)*. American Mathematical Society, 1993.
25. F. Laroussinie, K. G. Larsen, and C. Weise. From timed automata to logic - and back. In J. Wiedermann and P. Hájek, editors, *Mathematical Foundations of Computer Science 1995, 20th International Symposium*, volume 969 of *Lecture Notes in Computer Science*, pages 529–539, Prague, Czech Republic, 28 Aug.–1 Sept. 1995. Springer.
26. K. G. Larsen. Proof systems for satisfiability in Hennessy–Milner logic with recursion. *Theoretical Comput. Sci.*, 72(2–3):265–288, 23 May 1990.
27. J. L. Lions. ARIANE 5 flight 501 failure: Report by the inquiry board, July 1996. Available on-line at the URL <http://www.cs.aau.dk/~luca/SV/ariane.pdf>.
28. J. Magee and J. Kramer. *Concurrency: State Models and Java Programs*. John Wiley, 1999.
29. E. Mazur. *Peer Instruction: A User's Manual*. Series in Educational Innovation. Prentice-Hall International, Upper Saddle River, NJ, 1997.
30. R. Milner. *Communication and Concurrency*. Prentice-Hall International, Englewood Cliffs, 1989.
31. C. H. Papadimitriou. *Mythematics*: storytelling in the teaching of computer science and mathematics. In V. Dagdilelis, M. Satratzemi, D. Finkel, R. D. Boyle, and G. Evangelidis, editors, *Proceedings of the 8th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education, ITiCSE 2003, Thessaloniki, Greece, June 30–July 2, 2003*, page 1. ACM, 2003.
32. D. Park. Concurrency and automata on infinite sequences. In P. Deussen, editor, *5th GI Conference*, Karlsruhe, Germany, volume 104 of *Lecture Notes in Computer Science*, pages 167–183. Springer-Verlag, 1981.
33. D. A. Patterson. 20th century vs. 21st century C&C: The SPUR manifesto. *Commun. ACM*, 48(3):15–16, 2005.
34. A. Pnueli. The temporal logic of programs. In *Proceedings 18th Annual Symposium on Foundations of Computer Science*, pages 46–57. IEEE, 1977.
35. V. R. Pratt. Anatomy of the Pentium bug. In P. D. Mosses, M. Nielsen, and M. I. Schwartzbach, editors, *TAPSOFT'95: Theory and Practice of Software Development, 6th International Joint Conference CAAP/FASE, Aarhus, Denmark, May 22–26, 1995, Proceedings*, volume 915 of *Lecture Notes in Computer Science*, pages 97–107. Springer-Verlag, 1995.
36. B. Roscoe. *The Theory and Practice of Concurrency*. Prentice-Hall International, Englewood Cliffs, 1999.
37. S. Schneider. *Concurrent and Real-time Systems: the CSP Approach*. John Wiley, 1999.
38. B. Steffen and A. Ingólfssdóttir. Characteristic formulae for processes with divergence. *Information and Computation*, 110(1):149–163, 1994.
39. C. Stirling. Local model checking games. In *Proceedings of the 6th International Conference on Concurrency Theory (CONCUR'95)*, volume 962 of *LNCS*, pages 1–11. Springer-Verlag, 1995.

40. C. Stirling. *Modal and Temporal Properties of Processes*. Springer-Verlag, 2001.
41. A. Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5:285–309, 1955.
42. W. Thomas. On the Ehrenfeucht-Fraïssé game in theoretical computer science (extended abstract). In *Proceedings of the 4th International Joint Conference CAAP/FASE, Theory and Practice of Software Development (TAPSOFT'93)*, volume 668 of *Lecture Notes in Computer Science*, pages 559–568. Springer-Verlag, 1993.
43. D. Walker. Automated analysis of mutual exclusion algorithms using CCS. *Journal of Formal Aspects of Computing Science*, 1:273–292, 1989.
44. W. Yi. Real-time behaviour of asynchronous agents. In J. C. Baeten and J. W. Klop, editors, *Proceedings CONCUR 90*, Amsterdam, volume 458 of *Lecture Notes in Computer Science*, pages 502–520. Springer-Verlag, 1990.
45. W. Yi. *A Calculus of Real Time Systems*. PhD thesis, Chalmers University of Technology, Göteborg, Sweden, 1991.
46. W. Yi. CCS + time = an interleaving model for real time systems. In J. Leach Albert, B. Monien, and M. Rodríguez, editors, *Proceedings 18th ICALP*, Madrid, volume 510 of *Lecture Notes in Computer Science*, pages 217–228. Springer-Verlag, 1991.