CSCI 599: Automated Reasoning and Verification

Units: 4

Fall 2017 – Monday, Wednesday—2:00pm – 3:50pm

Location: KDC 236

Instructor: Chao Wang

Office: SAL 334

Office Hours: Tuesday 2:00pm – 3:30pm (tentative)

Contact Info:

Http: http://www-bcf.usc.edu/~wang626/

Email: wang626@usc.edu

Course Description

The goal of this course is to provide an introduction to foundational techniques of automated reasoning and verification. In the past decade, tremendous progress has been made in developing these logic-based machine intelligence techniques and applying them to rigorous verification of systems (hardware, software, and embedded systems). The course will cover classic topics such as temporal logic and model checking, as well as efficient implementations of decision procedures such as binary decision diagrams, satisfiability solvers, and satisfiability modulo theory (SMT) solvers. The course will also discuss applications of these techniques in industrial settings.

The course is self-contained and does not require any prior knowledge in this domain. However, it assumes that students are comfortable with the basic concepts in discrete mathematics and computer programming.

The course will be graded on the basis of six homework assignments, a midterm exam, and a final project.

Learning Objectives

Students will gain an understanding of the theories and foundational techniques of logic-based automated reasoning and verification, and learn how to leverage related software tools in practical settings. Major learning objectives are:

1. Understand the theoretical foundation of automated reasoning,
2. Write properties and formal specifications in computational tree logic (CTL),
3. Write properties and formal specifications in linear-time temporal logic (LTL),
4. Verify CTL and LTL specifications using model checking,
5. Use abstract and refinement to scale up the verification,
6. Construct and use BDDs for symbolic model checking,
7. Write and use SAT and SMT solvers for bounded model checking,
8. Apply the aforementioned techniques to practical systems.

Prerequisites

1. General proficiency in discrete mathematics
2. Good programming skills

Required Readings and Supplementary Materials

There are no textbooks. Detailed lecture notes will be provided by the instructor.

The following book serves as a supplementary reading material (optional):

1. Model Checking, by Clarke, Grumberg, and Peled, The MIT Press, 1999.

Description and Assessment of Assignments

The grades will be based on the completion of six homework assignments, a midterm exam, and a final research project. A more detailed explanation of each category is provided below, followed by a table showing the breakdown for each of these categories.

Homework
Homework includes a programming related assignment (HW0, which accounts for 10% of the grade) and five standard assignments (HW1-5, each of which accounts for 5% of the grade). Homework assignments must be submitted electronically. Please take care to upload the correct files, because they are the ones that will be graded.

Midterm exam
Midterm exam will be open-book and open-notes, but electronic devices will not be allowed. There will be no make-up exams. If you have an important reason for missing the midterm, please make arrangements with the instructor in advance. Otherwise, a missed exam receives a grade of 0.

Research paper presentation
Research paper presentation asks each student to choose a paper from a list of recently published papers suggested by the instructor. The goal is to teach others about the topic, so everyone in the class will have a better understanding of the most recent development in the field.

To get a good grade in paper presentation, you need to excel in the following aspects:

- Clarity in presentation (how well you understand the paper and handle questions, how smooth your talk is, etc.)
- Quality of the slides (your slides must to be informative and thorough, with technical depths, figures, etc.)

Final project
Final project is a research project that asks students to develop new formal verification techniques or identify innovative uses of existing formal verification techniques. At the end of the project, each student must submit a project report.

Your grade on the final project depends on the following aspects:

- Novelty of the project design,
- Thoroughness in the execution, and
- Clarity in the project report.

Below are two sample final projects:

1. *Project Summary – Formal Verification*: In this project, students are required to find an interesting application, formulate it into a formal verification problem, and solve it using techniques learned from this course. For example, you may use any of the following verification tools, such as NuSMV (http://nusmv.fbk.eu/NuSMV/), CBMC (http://www.cprover.org/cbmc/) or VIS (http://vlsi.colorado.edu/~vis/).\
2. *Project Summary – Program Synthesis:* Syntax Guided Synthesis (SyGus) is an emerging technique for directly generating software code from a set of logical specifications. In this project, students are required to learn the SyGuS specification language and the basics of SyGuS tools. You need to identify an interesting application of your choice, formulate it into a SyGuS problem, and solve it using a SyGuS tool (e.g., http://www.sygus.org/).

Grading Breakdown

Assignment	**% of Grade**
Homework	35%
HW0 programming related (10% of the grade)	
HW1-5 standard (5% of the grade each)	
Midterm exam	25%
Research paper presentation	10%
Final project	30%
TOTAL	**100%**

Additional Policies

Late assignments will be accepted up to 24 hours after the announced deadline, with a penalty of 20%. Assignments received more than 24 hours late will receive a grade of 0.

If you feel that an error has been made in grading, please notify the grader within one week after the material is returned. For exams and final projects, please present a short written appeal to the instructor.

Course Schedule: A Weekly Breakdown

	Topic	Reading	Slides	Assignment
Week 1	Overview/Admin Kripke structure, CTL	Chapter 1 Ch.2_A	Lecture_0 Lecture_1	HW0 out
Week 2	CTL model checking	Ch.2_B	Lecture_2 Lecture_3	HW0 due
Week 3	Fairness constraints Counterexamples	Ch.2_C	Lecture_4 Lecture_5	HW1 out
Week 4	Simulation relations	Ch.3	Lecture_6 Lecture_7	HW1 due
Week 5	Abstraction refinement	Ch.3	Lecture_8 Lecture_9	HW2 out
Week 6	LTL model checking	Ch.4	Lecture_10 Lecture_11	HW2 due
Week 7	LTL and omega automata	Ch.4	Lecture_12 Lecture_13	HW3 out
Week 8	Binary Decision Diagrams **(Midterm)**	Ch.5	Lecture_14	(Project proposal due) HW3 due
Week 9	Symbolic model checking	Ch.6	Lecture_15 Lecture_16	HW4 out
Week 10	SAT solvers	Ch.7	Lecture_17 Lecture_18	HW4 due
Week 11	Bounded model checking	Ch.8	Lecture_19 Lecture_20	HW5 out
Week 12	SMT solvers		Lecture_21 Lecture_22	
Week 13	Advanced research topics (e.g., abstract interpretation)	Recent papers	Student presentations	HW5 due
Week 14	Advanced research topics (e.g., concurrent software verification)	Recent papers	Student presentations	
Week 15	Advanced research topics (e.g., crypto software verification)	Recent papers	Student presentations	
FINAL	**(Project report due)** no exam			**(Project report due)**

Academic Conduct

Plagiarism – presenting someone else's ideas as your own, either verbatim or recast in your own words – is a serious academic offense with serious consequences. Please familiarize yourself with the discussion of plagiarism in *SCampus* in Part B, Section 11, "Behavior Violating University Standards" https://policy.usc.edu/student/scampus/part-b. Other forms of academic dishonesty are equally unacceptable. See additional information in *SCampus* and university policies on scientific misconduct, http://policy.usc.edu/scientific-misconduct.

Discrimination, sexual assault, intimate partner violence, stalking, and harassment are prohibited by the university. You are encouraged to report all incidents to the *Office of Equity and Diversity/Title IX Office* http://equity.usc.edu and/or to the *Department of Public Safety* http://dps.usc.edu. This is important for the health and safety of the whole USC community. Faculty and staff must report any information regarding an incident to the Title IX Coordinator who will provide outreach and information to the affected party. The sexual assault resource center webpage http://sarc.usc.edu fully describes reporting options. Relationship and Sexual Violence Services https://engemannshc.usc.edu/rsvp provides 24/7 confidential support.

Support Systems

A number of USC's schools provide support for students who need help with scholarly writing. Check with your advisor or program staff to find out more. Students whose primary language is not English should check with the *American Language Institute* http://ali.usc.edu, which sponsors courses and workshops specifically for international graduate students. *The Office of Disability Services and Programs* http://dsp.usc.edu provides certification for students with disabilities and helps arrange the relevant accommodations. If an officially declared emergency makes travel to campus infeasible, *USC Emergency Information* http://emergency.usc.edu will provide safety and other updates, including ways in which instruction will be continued by means of Blackboard, teleconferencing, and other technology.

DOI: 10.1145/1364782.1364805 Leah Hoffman

Q&A
Talking Model-Checking Technology

A conversation with the 2007 ACM A.M. Turing Award winners.

EDMUND M. CLARKE, E. Allen Emerson, and Joseph Sifakis were honored for their role in developing Model-Checking into a highly effective verification technology, widely adopted in the hardware and software industries.

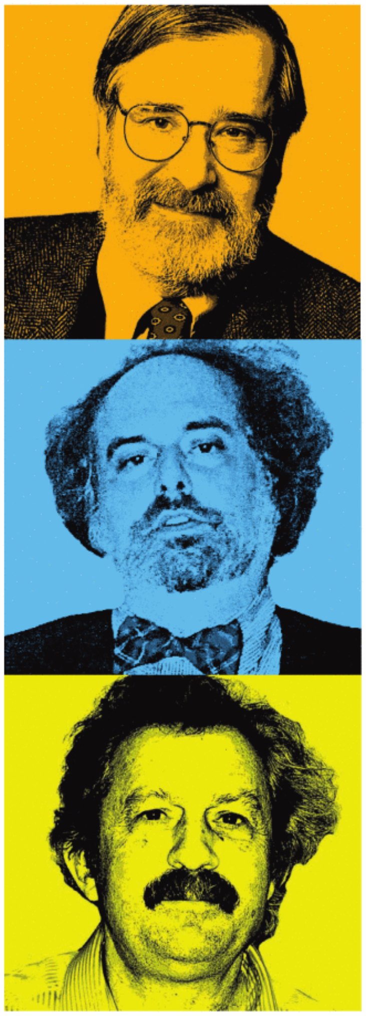
Let's talk about the history of formal software verification.

E. ALLEN EMERSON By the late 1960s, we recognized that a program should be viewed as a mathematical object. It has a syntax and semantics and formally defined behavior engendered by that syntax and semantics. The idea was to give a mathematical proof that a program met a certain correctness specification. So one would have some axioms characterizing the way the program worked for such-and-such an instruction and some inference rules, and one would construct a formal proof of the system, like philosophers do sometimes.

But it never really seemed to scale up to large programs. You ended up with something like 15-page papers proving that a half-page program was correct. It was a great idea but didn't seem to pan out in practice.

What about the history of model checking?

EDMUND M. CLARKE The birth of model checking was quite painful at times. Like most research on the boundary between theory and practice, theoreticians thought the idea was trivial, and system builders thought it was too theoretical. Researchers in formal methods were even less receptive. Research in the formal-methods community in the 1980s usually consisted of designing and verifying tricky programs with fewer than 50 lines using only pen and paper. If anyone asked how such a program worked in practice on a real computer, it would have been interpreted as an insult or perhaps simply as irrelevant.



EAE The idea behind model checking was to avoid having humans construct proofs. It turns out that many important programs, such as operating systems, have ongoing behavior and ideally run forever; they don't just start and stop. In 1977, Amir Pnueli suggested that temporal logic could be a good way to describe and reason about these programs. Now, if a program can be specified in temporal logic, then it can be realized as a finite state program—a program with just a finite number of different configurations. This suggested the idea of model checking—to check whether a finite state graph is a model of a temporal logic specification. Then one can develop efficient algorithms to check whether the temporal-logic specification is true of the state graph by searching through the state graph for certain patterns.

EMC Yes, Allen and I noticed that many concurrent programs had what we called "finite state synchronization skeletons." (Joseph Sifakis and J.P. Queille made the same observation, independently.) For example, the part of a mutual-exclusion program that handles synchronization does not depend

[CONTINUED ON P.110]

[CONTINUED FROM P.112]
on the data being exchanged in the critical sections. Many communication protocols had the same property. We decided to see if we could analyze finite-state programs by algorithmic means.

How exactly does that work?

EAE You have a program described by its text and its specification described by its text in some logic. It's either true or false that the program satisfies the specification, and one wants to determine that.

JOSEPH SIFAKIS Right. You build a mathematical model [of the program], and on this model, you check some properties, which are also mathematically specified. To check the property, you need a model-checking algorithm that takes as input the mathematical model you've constructed and then gives an answer: "yes," "no," or "I don't know." If the property is not verified, you get diagnostics.

And to formalize those specifications, those properties...

EAE What people really want is the program they desire, an inherently preformal notion. They have some vague idea about what sort of program they want, or perhaps they have some sort of committee that came up with an English prose description of what they want the program to do, but it's not a mathematical problem.

So one benefit of model checking is that it forces you to precisely specify your design requirements.

EMC Yes. But for many people, the most important benefit is that if the specification isn't satisfied, the model checker provides a counterexample execution trace. In other words, it provides a trace that shows you exactly how you get to an error that invalidates your specification, and often you can use that to find really subtle errors in design.

"The idea behind model checking was to avoid having humans construct proofs."

How have model-checking algorithms evolved over the years?

EMC Model-checking algorithms have evolved significantly over the past 27 years. The first algorithm for model checking, developed by Allen and myself, and independently by Queille and Sifakis, was a fixpoint algorithm, and running time increased with the square of the number of states. I doubt if it could have handled a system with a thousand states. The first implementation, the EMC Model Checker (EMC stands for "Extended Model Checker"), was based on efficient graph algorithms, developed together with Allen and Prasad Sistla, another student of mine, and achieved linear time complexity in the size of the state space. We were able to verify designs with about 40,000 states. Because of the state-explosion problem, this was not sufficient in many cases; we were still not able to handle industrial designs. My student Ken McMillan then proposed a much more powerful technique called symbolic model checking. We were able to check some examples with 10 to the one-hundredth power states (1 with a hundred zeros after it). This was a dramatic breakthrough but was still unable to handle the state-explosion problem in many cases. In the late 1990s, my group developed a technique called bounded model checking, which enabled us to find errors in many designs with 10 to the 10,000 power states.

EAE These advances document the basic contribution of model checking. For the first time, industrial designs are being verified on a routine basis. Organizations, such as IBM, Intel, Microsoft, and NASA, have key applications where model checking is useful. Moreover, there is now a large model-checking community, including model-checking users and researchers contributing to the advance of model-checking technology.

What are the limitations of model checking?

JS You have two basic problems: how to build a mathematical model of the system and then how to check a property, a requirement, on that mathematical model.

First of all, it can be very challenging to construct faithful mathematical models of complex systems. For hardware, it's relatively easy to extract mathematical models, and we've made a lot of progress. For software, the problem is quite a bit more difficult. It depends on how the software is written, but we can verify a lot of complex software. But for systems consisting of software running on hardware, we don't know how to construct faithful mathematical models for their verification.

The other limitation is in the complexity of the checking algorithm, and here we have a problem called the state-explosion problem (that Clarke referred to earlier), which means that the number of the states may go exponentially high with the number of components of the system.

EMC Software verification is a Grand Challenge. By combining model checking with static analysis techniques, it is possible to find errors but not give a correctness proof. As for the state-explosion problem, depending on the logic and model of computation, you can prove theoretically that it is inevitable. But we've developed a number of techniques to deal with it.

Such as?

EMC The most important technique is abstraction. The basic idea is that part of the program or the protocol you're verifying doesn't really have any effect on the particular properties that you're checking. So what you can do is simply eliminate those particular parts from the design.

You can also combine model checking with compositional reasoning, where you take a complex design and break it up into smaller components. Then you check those smaller components to deduce the correctness of the entire system.

How large are the programs we can currently verify with model checking?

EMC Well, first of all, there's not always a natural correspondence between a program's size and its complexity. But I would say we can often check circuits with around 10 to the 100th power states (1 with a hundred zeros after it).

JS Right. We know how to verify systems of medium complexity today—it's difficult to say but perhaps a program of around 10,000 lines. But we don't know how to verify very complex systems.

EMC We're always playing a catch-up game; we're always behind. We've developed more powerful techniques, but it's still difficult to keep up with the advance of technology and the complexity of new systems.

Can we use model checking to check concurrent programs?

EAE Arguably, model checking is a very natural fit for parallel programming. Typically, we treat parallelism as a nondeterministic—or, informally, random—choice, so, in a way a parallel program is a more complex sequential program, with many nondeterministic behaviors. Model checking is very well suited to describing and reasoning about the associated coordination and synchronization properties of parallel programs.

EMC Concurrent programs are much more difficult to debug because it's difficult for humans to keep track of a lot of things that are happening all at once. Model checking is ideal for that.

JS But if you have programs that interact with the physical environment, time becomes very important. For these systems, verification is much more complicated.

Do we have any algorithms that can operate directly on implementable code?

EMC To verify the process of translating a design to code, or to verify the code itself, is much more difficult. Some successful model checkers use this approach, however. The Java Pathfinder model checker developed at NASA Ames generates byte code for a Java program and simulates the byte code to find errors.

JS The best available technology is proprietary technology that was developed by U.S. companies. But most of the code-level model checkers are used to verify sequential software. If you want to verify concurrent software, then you need to be very careful.

EMC The SLAM model checker developed at Microsoft Research for finding errors in Windows device drivers is probably the most successful software model checker. It is now distributed to people who want to write device drivers for Windows. However, it is hardly a general-purpose software model checker.

"If you have programs that interact with the physical environment, time becomes very important. For these systems, verification is more complicated."

EAE In hardware verification, Verilog and VHDL are widely used design description languages. Many industrial model checkers typically accept designs described in these languages.

Is model checking something currently taught to undergraduates?

JS Formal verification is definitely taught in Europe. Europe has traditionally had a stronger community in formal methods, and I'd like to say it has also traditionally had a stronger community in semantics and languages.

EMC Yes, there's always been more interest in verification in Europe than in the U.S. Most of the major universities here—CMU, Stanford, UC Berkeley, U. Texas, and so on—do offer courses in model checking at both undergraduate and graduate levels, but it hasn't filtered down to schools where no one is doing research in the topic. Part of that has to do with the availability of appropriate textbooks; good books are just beginning to come out.

EAE Formal methods are being taught with some frequency [in the U.S.], but they are not broadly incorporated into the core undergraduate curriculum as required courses to the extent that operating systems and data structures are. It is probably more prevalent at the graduate level. But the distinction between undergraduate and graduate is not clear-cut. At many schools advanced undergrad and beginning grad overlap.

What's in store for model checking and formal verification?

EMC I intend to continue looking at ways of making model checking more powerful. The state explosion phenomenon is still a difficult problem. I have worked on it for 27 years and probably will continue to do so. Another thing I want to do is focus on embedded software systems in automotive and avionics applications. These programs are often safety-critical. For example, in a few years, cars will be "drive-by-wire"; there will be no mechanical linkage between the steering wheel and the tires. The software will definitely need to be verified. Fortunately, embedded software is usually somewhat simpler in structure, without complex pointers; I think it may be more amenable to model checking techniques than general software.

JS Personally, I believe we should look into techniques that allow some sort of compositional reasoning, where we infer global properties from local properties of the system, because of the inherent limitations of techniques based on the analysis of a global model. I'm working on this, as well as on theories of how to build systems out of components, component-based systems.

EAE Model checking has caused a sea change in the way we think about establishing program correctness, from proof-theoretic (deductive proof) to model-theoretic (graph search). I think we will continue to make more or less steady progress, but the pace of development of hardware and software is going to accelerate. Whether we ever catch up I don't know. Systems that are being designed are getting bigger and messier. The seat-of-the-pants approach will no longer work. We'll have to get better at doing things modularly, and we'll have to have better abstractions. **C**

Leah Hoffman writes about science and technology from Brooklyn, NY.