Building a testing-free future

January 11 2021, by Steve Crang

A group of U-M researchers uses mathematical proofs to demonstrate that software meets specifications for safe, correct execution without the need for traditional testing. Credit: Karen Parker Moeller, MOEdesign

It's common sense: when you write software, you check your work. Testing is a carryover solution from all the other tools we've engineered throughout history, and it's a cornerstone of quality software design. And yet, as a standalone approach, testing is lacking: it's time consuming, it's labor and resource intensive, and most importantly it's extremely difficult to do exhaustively, leaving deployed software susceptible to unexpected inputs and behaviors.

In fact, most software in use today is so complex that testing it exhaustively is practically impossible.

"It would take billions and billions of years to run perfect test cases that cover all the behavior of reasonably large programs," explains assistant professor Baris Kasikci, "and if you leave certain things out, bugs can creep in."

But many critical systems rely in large part on testing to ensure that they're safe and reliable, leaving the question of bugs not an "if" but a "when."

"We're playing with bad software," says professor Karem Sakallah. "Right now, most of the software that we use is brittle and breaks at some point."

Researchers like Kasikci and Sakallah envision a smarter alternative. Rather than having humans, with all our imaginative limitations, come up with test cases, why not let math do the heavy lifting? Enter formal verification, a means to demonstrate that a program or algorithm is correct and reliable with all the elegance of a logical proof.

"Formal verification is an alternative here," assistant professor Manos Kapritsos explains. "I'm not going to try all the possible inputs and outputs; I'm going to prove, by converting my program into a series of logical formulas, that a property I specify holds at the end of my program."

That property, called a specification, is designed to describe how the program is allowed to behave. Essentially, a programmer has to be able to describe what the different good outputs of a program look like. The goal then is to prove that it's impossible for the program to behave otherwise.

"Having a foolproof system that says: you develop it, you check it automatically, and you get a certificate of correctness," Sakallah says, "that's what gives you confidence that you can deploy a program without issue."

Making it worth your while

A common question about formal verification is why it isn't just used on everything, if it's supposed to be so airtight. Unfortunately, while testing is certainly time consuming, writing proofs about a piece of code or an algorithm is usually even worse on that front—so far, anyway.

"Verification is associated with a great deal of time and energy," says Upamanyu Sharma, an undergraduate alum who did research with Kapritsos, "and for less critical applications, program testing and some static analysis is usually deemed sufficient."

That's why a primary early motivation for Kapritsos, Kasikci, and Sakallah, working with assistant professor Jean-Baptiste Jeannin in Aerospace Engineering, was automating the process.

To understand the difficulty, you have to look at what's involved in verifying a program. As mentioned above, the first step is to write a specification (spec), or a description of what the program's correct behavior should look like. It should be clear whether any specific program state, or the values a program has stored following each instruction, follows the spec or not.

The goal is to demonstrate that the spec is true for all program states that can be reached in all possible executions. For example, to prevent collisions at an intersection, a program for a traffic light controller must never end up in a state with a green light going all directions. Logically, a property that holds in all cases like this is called an invariant. Showing that this invariant holds without exhaustive testing requires a proof by induction. Coming up with this inductive invariant is the key step in these proofs.

"You prove properties based on the structure of the code, and you make an inductive proof," Kasikci explains. Induction is a core concept in mathematical and computational proofs. You demonstrate two things: first, that the property holds in the program's starting state; and then, that the property is maintained each time the program's state changes.

If the property holds for every possible state, you've got an invariant, proven by induction.

"Manually finding the inductive invariant is a tedious and error-prone process and is, in fact, one of the biggest challenges in formal verification," Sakallah explains.

That's what makes automation of this process so powerful—the researchers maintain that it will be a key to the practice's broader adoption.

Automatically deriving inductive invariants is achieved in most cases with tools called model checkers that construct and reason about mathematical formulas describing the code and the spec. Model checkers have been very successful in the hardware space for verifying the safety and correctness of chip designs; a model checker designed by Sakallah called AVR received the top award at last year's Hardware Model Checking Competition.

The lessons learned from these hardware verification problems have been quite helpful in designing software verifiers. Until very recently, however, the use of automation on software was limited to rudimentary problems.

"Adoption that we're looking at is mostly for very well-defined implementations," Kasikci says. "You're talking about quicksort algorithms that you can write in 20 lines of code. We have tools that can automatically prove that there is no bug in that code."

Basic data structures, basic algorithms—that was the level of complexity possible in discussions about automated verification when Kapritsos and Kasikci began their work. "They're still very important, because you rely on these things all the time when building software."

But the researchers are scaling things up. Their work over recent years has focused on adapting these tools to more complex classes of software that were previously out of reach.

An illusion of complexity

The first series of breakthroughs for the group was focused on the problem of automatically verifying the protocols that define how large, distributed computing systems communicate and cooperate. This problem brought together the challenging components of automatic verification and complex distributed systems that are very difficult to analyze.

To solve it, the researchers first made a key separation. Rather than package the abstract distributed protocol together with its actual implementation in code, they attacked the protocols alone.

"A protocol is pseudocode, or an algorithm," Kasikci explains. "It's a high-level description of how a system should behave." Taking that pseudocode and writing it into a real implementation introduces a lot of additional complexity.

By removing the complexity of implementation, they can formally verify a protocol and demonstrate its correctness in general before it's called upon for a particular use case.

"Even if you don't go into the implementation, reasoning about the protocol is very important," Kapritsos says. "If the protocol has a flaw, then of course the implementation would have a flaw. Before you pay developers to spend time implementing a protocol, you make sure that the protocol is actually correct."

A good example to understand what these distributed protocols do is the Paxos consensus protocol. This system is implemented widely to give the illusion of a single correct machine executing user requests, while in reality they're being handled by a large, complex system made up of many machines that can fail. Think of a bank with multiple branches and ATMs. For the user making a transaction, there might as well be a single bank server that gives them the same results no matter which ATM they use. But behind the scenes, there are many machines on that bank's network that have to remain in agreement about the transaction. Further, this illusion has to stay in place even if one or more of the bank's servers is out of commission at the time.

The issue with formally verifying protocols like this one is pretty straightforward: they're huge. The networks under consideration are widespread and can involve a theoretically limitless number of servers.

The group's solution came from a unique feature of these protocols inspired in part by Sakallah's background in hardware analysis. That feature was symmetry, and the insight paved the way for the group's most important breakthrough—what if distributed protocols aren't actually as complex as they look?

"Human-created artifacts seem to have some structure to them that make them tractable," Sakallah explains. "People do not create random things, they create very complex, but highly-structured, things: an airliner, chips, computers, software programs."

But something is lost between the period of creation and analysis, Sakallah continues. Making sense of a complex finished product seems overwhelming, but people had to actually make it somehow. "If you bring the insight that went into creating the artifact, you can use it to make the analysis of it tractable."

Formally verifying a distributed protocol like Paxos may involve a theoretically limitless number of "nodes" (individual servers or other entities connected by the protocol), but, the researchers argue, there isn't anything special about each one. They're all essentially the same, and all play the same role in the protocol. They back up data, agree on all of the transactions taking place, handle things a certain way in the event of a node failure—and they all do this uniformly.

As a consequence, solving the problem for a small number of nodes could be seamlessly extended to a huge number of nodes without meaningfully changing the analysis. This was the key to automation, and became the basis for the group's protocol verification framework called I4. Their method made use of the Averroes or AVR verification system originally designed for hardware in Sakallah's lab.

"The essence of our idea is simple," they wrote in their paper, presented to the ACM Symposium on Operating Systems Principles (SOSP) in October 2019, "the inductive invariant of a finite instance of the protocol can be used to infer a general inductive invariant for the infinite distributed protocol."

In the end, they were able to simplify the problem enough that they could use a model-checking tool to verify several distributed protocols with little to no human effort.

"Our goal in that project was to see if we could take the human out of the loop," says Kapritsos. With further development of I4, the researchers want any developer to be able to prove a distributed protocol that they don't necessarily understand very well themselves.

Sakallah and Aman Goel, a Ph.D. student in Sakallah's lab, undertook one such further development in 2020 with their project IC3PO. This tool, like I4 before it, is an extension of the AVR hardware verifier that takes fuller advantage of the symmetry and regularity observed in distributed protocols. It uses that symmetry to significantly scale their verification, make it more automated, and produce inductive invariants that are compact enough for a developer to read and understand afterwards.

The ultimate goal of IC3PO is to automatically prove complex protocols, such as Paxos, which would be a major advance in the verification of unbounded, or infinite, systems.

Kapritsos and Kasikci are also taking this arm of the effort further with a new NSF Formal Methods in the Field grant for their project "Automating the Verification of Distributed Systems." The proposed research will investigate a new approach for automating the verification of complex software running on multiple machines.

"This project is meant to push the idea of I4 further, into more complex protocols and implementations," Kapritsos says.

The nitty gritty

With the work on automatically verifying distributed protocols off to an ambitious start, that still left the actual messy implementations of these and other systems in need of exploration. Kasikci and Kapritsos took several steps in this direction throughout 2020.

Kapritsos and a team of collaborators published a tool called Armada at the 2020 ACM Conference on Programming Language Design and Implementation (PLDI) that targeted the semi-automatic verification of another complex class of programs, concurrent programs.

Concurrency has been a vital tool for increasing performance after processor speeds began to hit a plateau. Through a variety of different methods, the technique boils down to running multiple instructions in a program simultaneously. A common example of this is making use of multiple cores of a CPU at once.

Formal verification on these programs is notoriously difficult, even lagging by a decade behind other verification work, according to Kapritsos and undergraduate alumnus Upamanyu Sharma, both co-authors on the paper.

"The main challenge in concurrent programs comes from the need to coordinate many different threads of code together," Sharma says. "To verify that multi-threaded programs are correct, we have to reason about the huge number of possible interleavings that are possible when multiple methods run at the same time."

To date, a variety of proof methods have been designed to deal with different types of concurrency. In this project, the researchers set out to design a single framework that allows a user to apply many of these techniques to verify a single program.

With Armada, programmers use a C-like language to select a proof technique and provide some annotation describing how it should be applied to the program they're trying to verify. From there, the proof itself is generated automatically and ready to be run through a prover for verification. In the event the proof fails, the user changes their annotation or working code and generates a new one.

In the world of verifying concurrent programs, this is to date the most low-effort technique available. The authors hope that this shorter pipeline will encourage the broader use of verification outside of the most critical systems where the technique is already justified.

Kapritsos is also in the early stages of expanding the work from bug-proofing to another major pain point in software testing, performance. This as-yet untread territory could remove a major hurdle between eliminating all testing from the software development pipeline.

On another front, Kasikci was awarded a grant from DARPA to adapt formal methods to a unique new setting—small patches to compiled binary code. This project, called Ironpatch, will target complex systems already in deployment, like cars, ships, and rockets, that are so dense and heterogeneous that they're difficult to patch through traditional means.

"It's a little bit of a mess," said Kasikci. "Traditionally, you fix the bug in the source code, you rebuild the software and you redeploy it. But these moving environments are really hostile to that model because there's a lot of different software and lots of different kinds of computers."

Ironpatch takes a different approach, bypassing the source code and instead making tiny modifications called micropatches directly to the binary heart of the running software. This eliminates the need to recompile the software and because the changes it makes are so minute, they can fix problems without causing others. Ironpatch is designed to be self-contained—once a vulnerability is identified, the system will automatically generate, formally verify and apply a micropatch to eliminate it. This also eliminates the need to upload software patches to a remotely located system, a particularly handy feature when that system might be located on a spacecraft millions of miles away. Kasikci will be working on Ironpatch jointly with Kapritsos and associate professor Westley Weimer.

Verifying the future

These projects are just the beginning; the long-term ambitions of the researchers are much more thorough.

"For me there is a broader theme here—it's bringing formal verification closer to practice," Kapritsos says, "bringing it closer to real developers by making it more useful and easier to perform."

That means more than speeding up a testing alternative, it means a whole new paradigm in security and reliability. With formal verification as a universal stage in the development pipeline, it will enable entire libraries of complex, reusable code that is certified safe and ready to deploy.

"The key idea is composability," Kasikci says. "It's one of the fundamental ideas in computer science. We have little things, we put them together, and we build bigger things. We have functions, we put them together, we have programs. The beauty of formal verification is that when you prove a property about a function, you prove it for all input and output combinations. It doesn't matter what you compose with it, the property's still going to hold. If you have a bunch of these things where the property you're after is proven, then you can compose them to make something larger for which the property will hold automatically."

More information: Jacob R. Lorch et al. Replication Package for

Article "Armada: Low-Effort Verification of High-Performance Concurrent Programs", (2020). DOI: 10.1145/3395653

Haojun Ma et al. I4, *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (2019). DOI: 10.1145/3341301.3359651

Provided by University of Michigan

Citation: Building a testing-free future (2021, January 11) retrieved 18 April 2024 from https://techxplore.com/news/2021-01-testing-free-future.html