Debugging system for complex analysis of programs that import huge swaths of commonly used code

May 25 2016, by Larry Hardesty

“[T]oday, if you want to write a program, you go and bring in these huge frameworks and these huge pieces of functionality that you then glue together, and you write a little code to get them to interact with each other. If you don’t understand what that big framework is doing, you’re not even going to know where your program is going to start executing,” says Armando Solar-Lezama.

Symbolic execution is a powerful software-analysis tool that can be used to automatically locate and even repair programming bugs. Essentially, it traces out every path that a program's execution might take.

But it tends not to work well with applications written using today's programming frameworks. An application might consist of only 1,000 lines of new code, but it will generally import functions—such as those that handle virtual buttons—from a programming framework, which includes huge libraries of frequently reused code. The additional burden of evaluating the imported code makes symbolic execution prohibitively time consuming.

Computer scientists address this problem by creating simple models of the imported libraries, which describe their interactions with new programs but don't require line-by-line evaluation of their code. Building the models, however, is labor-intensive and error prone, and the models require regular updates, as programming frameworks are constantly evolving.

Researchers at MIT's Computer Science and Artificial Intelligence Laboratory, working with colleagues at the University of Maryland, have taken an important step toward enabling symbolic execution of applications written using programming frameworks, with a system that automatically constructs models of framework libraries.

The researchers compared a model generated by their system with a widely used model of Java's standard library of graphical-user-interface components, which had been laboriously constructed over a period of years. They found that their new model plugged several holes in the hand-coded one.

They described their results in a paper they presented last week at the International Conference on Software Engineering. Their work was

funded by the National Science Foundation's Expeditions Program.

"Forty years ago, if you wanted to write a program, you went in, you wrote the code, and basically all the code you wrote was the code that executed," says Armando Solar-Lezama, an associate professor of electrical engineering and computer science at MIT, whose group led the new work. "But today, if you want to write a program, you go and bring in these huge frameworks and these huge pieces of functionality that you then glue together, and you write a little code to get them to interact with each other. If you don't understand what that big framework is doing, you're not even going to know where your program is going to start executing."

Consequently, a program analyzer can't just dispense with the framework code and concentrate on the newly written code. But symbolic execution works by stepping through every instruction that a program executes for a wide range of input values. That process becomes untenable if the analyzer has to evaluate every instruction involved in adding a virtual button to a window—the positioning of the button on the screen, the movement of the button when the user scrolls down the window, the button's change of appearance when it's pressed, and so on.

For purposes of analysis, all that matters is what happens when the button is pressed, so that's the only aspect of the button's functionality that a framework model needs to capture. More precisely, the model describes only what happens when code imported from a standard programming framework returns control of a program to newly written code.

"The only thing we care about is what crosses the boundary between the application and the framework," says Xiaokang Qiu, a postdoc in Solar-Lezama's lab and a co-author on the new paper. "The framework itself is

like a black box that we want to abstract away."

To generate their model, the researchers ran a suite of tutorials designed to teach novices how to program in Java. Their system automatically tracked the interactions between the tutorial code and the framework code that the tutorials imported.

"The nice thing about tutorials is that they're designed to help people understand how the framework works, so they're also a good way to teach the synthesizer how the framework works," Solar-Lezama says. "The problem is that if I just show you a trace of what my program did, there's an infinite set of programs that could behave like that trace."

To winnow down that set of possibilities, the researchers' system tries to fit the program traces to a set of standard software "design patterns." First proposed in the late 1970s and popularized in a 1995 book called "Design Patterns," design patterns are based on the idea that most problems in software engineering fit into just a few categories, and their solutions have just a few general shapes.

Computer scientists have identified roughly 20 design patterns that describe communication between different components of a computer program. Solar-Lezama, Qiu, and their Maryland colleagues—Jinseong Jeon, Jonathan Fetter-Degges, and Jeffrey Foster—built four such patterns into their new system, which they call Pasket, for "pattern sketcher." For any given group of program traces, Pasket tries to fit it to each of the design patterns, selecting only the one that works best.

Because a given design pattern needs to describe solutions to a huge range of problems that vary in their particulars, in the computer science literature, they're described in very general terms. Fortunately, Solar-Lezama has spent much of his career developing a system, called Sketch, that takes general descriptions of program functionality and fills in the

low-level computational details. Sketch is the basis of most of his group's original research, and it's what reconciles design patterns and program traces in Pasket.

"The availability of models for frameworks such as Swing [Java's library of graphical-user-interface components] and Android is critical for enabling symbolic execution of applications built using these frameworks," says Rajiv Gupta, a professor of computer science and engineering at the University of California at Riverside. "At present, framework models are developed and maintained manually. This work offers a compelling demonstration of how far synthesis technology has advanced. The scalability of Pasket is impressive—in a few minutes, it synthesized nearly 2,700 lines of code. Moreover, the generated models compare favorably with manually created ones."

More information: Synthesizing framework models for symbolic execution. DOI: 10.1145/2884781.2884856 , people.csail.mit.edu/xkqiu/icse2016-final.pdf

This story is republished courtesy of MIT News (web.mit.edu/newsoffice/), a popular site that covers news about MIT research, innovation and teaching.

Provided by Massachusetts Institute of Technology