QRA
critical thinking

It's Time to Start Using Formal Methods

FOR ENGINEERING EMBEDDED SYSTEMS

INTRODUCTION

Between 1985 and 1987, a radiation therapy device called the Therac-25 was involved in at least six incidents in which the device delivered massive overdoses of radiation. The patients involved suffered radiation burns and symptoms of radiation poisoning. Three of those patients eventually died. All because of a latent software bug. A race condition that had gone undetected. A test case no one had thought to define.

Thirty-five years have now passed since the Therac-25 was brought to market in 1982. In that time, the volume and complexity of software in embedded systems has grown enormously. More and more of that software has become mission-critical and safety-critical. If embedded systems are to function effectively and safely, that software must be extremely reliable.

To meet ever-increasing reliability demands, new methodologies for specifying, designing and coding the software in embedded systems – methods like model-based design – have evolved. Yet software verification, for the most part, has remained rooted in the same methods that were used to test the Therac-25. We're still defining test cases and monitoring test coverage. In other words, our procedures for verification of software have not kept pace with our advances in designing and implementing it.

As the complexity of embedded systems and their reliance on software for mission-critical and safety-critical functions continue to grow, the organizations that develop these systems will eventually be forced to adopt more robust methodologies for their verification.

Fortunately, recent advances have made verification techniques known as formal methods a viable alternative to traditional testing.

We believe the use of formal methods for model-based design verification will offer systems and software engineers – and the companies they work for – a much higher level of confidence in the accuracy and robustness of the embedded systems they design and produce.
We believe the time to begin transitioning to formal methods for model-based design verification is now. In this article, we'll explain why. But first, let's look at what we mean by formal methods.

A BRIEF HISTORY OF FORMAL METHODS

In computer science, "formal methods" are techniques that use mathematical logic to reason about the behaviour of computer programs.

To apply formal methods in system verification, you (or a tool built for the purpose) must translate your system into a mathematical structure – a set of equations. You then apply logic, in the form of mathematical "rules," to ask questions about the system and obtain answers about whether particular outcomes occur.

Formal methods go all the way back to Euclid. So, almost all of us thus have some experience with them from a secondary school geometry class. As you'll undoubtedly remember, we start with an axiom or postulate, which we take as self-evident, and we use logic to reason toward our theorem using "rules" which had previously been proven true. If we always apply only the logical transformations allowed, then the conclusion we reach at the end – our theorem – must be true. QED.

Formal methods for engineering computer systems work in much the same way.

In computer science, formal methods really kicked off – on a theoretical basis – in the late 1960s and early '70s, when widespread use of computing was still in its infancy. Theoretical mathematicians were observing computer programming, still relatively simple at the time, and saying, *Hey, that's a mathematical structure! I can apply set theory to that!*

Tony Hoare is generally credited with introducing formal methods to computer science with his paper An Axiomatic Basis for Computer Programming and his invention of Hoare logic. Hoare logic[i] and similar formal methods work much like algebra. They even make use of algebraic laws, like the associative, commutative and distributive properties. You apply the same transformation on both sides of the equal sign, and both sides of the equation remain equal.

Let's say you want to prove a specific output of your system never goes above a certain value. Using formal methods, you would apply your chosen set of rules to prove your assumption – your requirement – is true. In the end, if you've applied your algorithms correctly, and if you find that, indeed, your selected output never exceeds that specified value, then, as in an Euclidean proof, there is no question your theorem is true. You're absolutely certain of it. You've proven beyond a doubt that your system meets that requirement.

Image 1: Tony Hoare introducer of formal methods to computer science.

In contrast, if you were to apply a representative set of inputs to your system to test your assumption empirically, you could never really be sure your assumption was true. Unless, of course, your set of test cases exercised all possible combinations of input values and stored states which affect the selected output. A daunting and exponential task in today's embedded software environment.

To illustrate this point, let's look at another, very basic example. Suppose you wanted to find the zeros of the polynomial x2 + 5x +6. Now, you could try plugging in values for x until you were satisfied you had found all the zeros. "Or, you could simply solve the quadratic equation (of the from ax^2+bx+c=0) with the quadratic formula:

$$x=(-b +-sqrt(b^2-4ac))/(2a)$$

which in this example gives the solution...

$$0 = x^2 + 5x +6 = (x + 2)(x + 3).$$

Now, you've *proven* that the zeros of the equation are -2 and -3. That's how formal methods work.

EARLY USE OF FORMAL METHODS FOR ENGINEERING APPLICATIONS

Formal methods didn't gain much traction with industry until the 1990s. Before then, computers and computer programs were relatively simple, while formal methods were primitive and difficult to apply. Testing remained the most efficient means of system verification.

Then, programming errors began getting companies into serious trouble.

Not long after the Therac-25 catastrophe, disaster struck AT&T's global long-distance phone network. On January 15, 1990, a bug in a new release of switching software caused a cascade of failures that brought down the entire network for more than nine hours. By the time the company's engineers had resolved the problem – by reloading the previous software rel – AT&T had lost more than $60 million in unconnected calls. Plus, they'd suffered a severe blow to their reputation – especially amongst customers whose businesses depended on reliable long-distance service.
Four years later, a bug was discovered in the floating-point arithmetic circuitry of Intel's highly-publicized Pentium processor. This error caused inaccuracies when the chip divided floating-point numbers within a specific range. Intel's initial offer – to replace the chips only for customers who could prove they needed high accuracy – met with such outrage that the company was eventually forced to recall the earliest versions. Ultimately, the Pentium FDIV bug cost Intel some $475 million.

The Therac-25, the AT&T switching control software and the Intel Pentium chip were all tested extensively. Still, that testing failed to find the catastrophic bugs in those systems. Today, due in large part to the Pentium bug, formal methods verification is now a standard practice at Intel [ii], and is used routinely by other manufacturers to verify IC chip designs. Yet software developers lag far behind hardware makers in the use of formal methods for embedded system verification.

This discrepancy is due primarily to the difference

between IC logic and modern software logic. The logic in a CPU reduces to arrays of logic gates: ANDs, NANDs, ORs, etc. It's all Boolean. The formal methods engines used for Boolean logic, such as *satisfiability solvers,* or SAT solvers, are now very well understood (thanks, again, to the Pentium bug, and to companies who picked up the ball and ran with it). Formal verification of ICs requires very fast computers, but only because the logic arrays are so vast.

Software is a whole different problem. Modern software logic is more complicated than IC logic. It requires more sophisticated mathematics. The solvers used in formal methods verification of software, known as *satisfiability modulo theories* SMT solvers, add mathematical constructs beyond Boolean logic.

SMT solvers have taken longer to mature. In fact, they're still evolving. For now, it is quite difficult to apply formal methods to the full source code of large-scale embedded applications. Converting large, complex source files – like a flight-control program, for example – into formal methods language is still a daunting, arduous and extremely time-consuming task.

But that doesn't make formal methods software verification impossible.

To apply formal methods to a large software program today, you need to do one of two things. You can apply them to small portions of the program, critical parts that must work without fail, for example. Or you can apply them to an *abstraction* of the actual implementation.

Model-based design is just such an abstraction. It simplifies the representation of the system and breaks it into interconnected blocks. This abstraction, in turn, simplifies both the task of translating the design into formal methods language, and the task of querying the system.

Recent breakthroughs, which we'll discuss shortly, as well as complete coverage of the design now make this second approach the preferred one for formal verification of embedded systems.

But before we discuss this approach further, let's look more closely at the reasons for applying it.

The amount of software in cyber-physical embedded systems continues to grow. Systems like automobiles, purely mechanical thirty or forty years ago, are now bristling with processors running millions of lines of code. More and more of that code is mission-critical and safety-critical. Embedded programs are getting so big, they're becoming too difficult to test.

Traditional testing methods involving test cases and coverage – methods that worked fine twenty or thirty years ago, on simpler systems – don't really work anymore. The sheer volume and complexity of today's embedded software make testing a losing proposition. It keeps getting harder and harder to prove that nothing disastrous will go wrong.

Lack of confidence in testing is beginning to impede innovation. Take the integration of self-driving cars with computer controlled intersections. Scientists claim this concept would eventually eliminate the need for traffic lights, ease urban road congestion and save millions of lives. Unfortunately, engineers we spoke with at the Embedded Software Integrity for Automotive conference in Detroit last year told us that - while they have the capability to build such a system – they literally cannot solve the problem of how to verify it to a high enough level of confidence. They wouldn't be able to trust it. It would just be too great a liability.

In other words, our engineering ideas and design capacities are outpacing our ability to test the software that controls them.

WHY THE TIME IS RIGHT FOR FORMAL METHODS FOR ENGINEERING EMBEDDED SYSTEMS

Formal methods represent a big shift away from how most systems are being verified today. Making that shift will require a significant expenditure, and for now, it's tough to make an economic justification for it. An accountant might ask, "Couldn't we just increase our testing and still spend less?" And it would be hard to argue with him. It's difficult to calculate ROI... until a catastrophe occurs.

On the other hand, companies who doggedly continue with traditional testing risk getting left behind. Organizations like NASA, Lockheed Martin and Honeywell are gradually making the shift to formal methods. Those who delay could find themselves struggling to catch up, while losing competitive advantage.

There is no real alternative in sight. Traditional testing is simply not a viable method for verification of tomorrow's complex embedded systems. Disasters like the Therac-25, the AT&T network collapse and the Pentium FDIV bug will become more frequent in the future, unless we shift toward formal verification in embedded systems. Companies need to start looking at formal methods on small projects or parts of projects, and begin charting their migration to formal methods verification.

Fortunately, three major breakthroughs are making it far easier to adopt formal methods today.

The first of these breakthroughs is an exponential improvement in SAT and SMT solvers and theorem provers. These tools are now thousands of times faster than they were just a few years ago. And new solvers and theorem provers, like Microsoft's Z3, amalgamate different types of solvers to solve different types of problems. They're bringing together the best research from around the world and putting it at user's fingertips.

Second, dramatic reductions in the cost of distributed computing now let us throw much more computing power at a problem for much less money. As a result, a problem that may have taken an SMT solver eight minutes to solve in 2012 takes only about two seconds today.

And finally, the more widespread adoption of model-based design is making it easier to apply formal methods to a wider range of problems. This developing market has given rise to the development of a growing number of **formal methods verification tools**, which are built for use with model-based design applications like MathWorks' Simulink.

THE ADVANTAGES OF FORMAL METHODS VERIFICATION TOOLS FOR MODEL-BASED DESIGN

Without the new tools just mentioned, translating a Simulink model into solver language would be slow, tedious work, and the result would likely not be very robust. Plus, solver output tends to be difficult to interpret for someone without a practiced eye.

With these new tools, on the other hand, the process of translation is automated and accelerated, while interpretation is greatly simplified and far more intuitive.

Take QRA's new tool, QVtrace, for example. To start the process, you simply input your Simulink model into QVtrace. QVtrace automatically translates the model into solver language. Then, to verify your model, you pose questions to QVtrace based on your system's requirements. This is the largest task of the process, as it involves translating each requirement into QVtrace's mathematical requirements language, QCT. Once you've input a requirement, you just click the Analyse button and QVtrace solves the model for the requirement.

When QVtrace finishes solving for the input requirement, it will return one of two things. It will either provide confirmation (and thus, verification) that the model meets the requirement, or it will provide a counterexample.

If the design fails to meet the requirement, QVtrace will supply a counterexample in the form or a set of inputs that would cause the system to violate the requirement. It also visually highlights the parts of the model involved in the violation of the requirement. This is the "trace" in QVtrace. This last feature focuses your search, helps you find bugs faster, and expedites correction and verification of your design.

One of the biggest advantages of formal methods verification tools like QVtrace, is that they find those "odd" cases that testing often misses. Cases that take many time steps to trigger. Cases that testers wouldn't think of. Cases that cause disasters like those we mentioned earlier. That's because an SMT solver doesn't formulate test cases or reason about whether something is reasonable to test. It simply solves the equation. It examines everything that could affect the output.

And because they solve equations, formal methods verification tools provide better coverage than even automated test case generation. Test cases generated by a modelling tool may test every path, but they won't cover every condition. Formal methods verification tools offer complete coverage, because they convert the model into a single (albeit enormous) equation and solve the equation for each requirement.

Formal methods may also facilitate a new model-based design process called correct by construction. To use this process, you first model a small portion of your system and then verify it using formal methods. You then correct and reverify, until you're one hundred percent certain that part of the system functions perfectly. Then, you add a bit more to the model and run a complete verification again. Since you've already verified your baseline model, you know that any errors you find will be in the latest addition. You then correct, reverify, and keep repeating the process until your design is complete. Correct by construction should result in systems that are better designed and more reliable.

Finally, by querying their models with formal methods tools, engineers gain greater understanding of their designs. Greater understanding will give them greater confidence in their current design, and – in terms of lessons learned – a stronger foundation to build on in future projects.

CONCLUSION

State-of-the-art embedded systems have become too big and too complex to be reliably verified using traditional testing methods.

Traditional testing has simply become too risky from a liability standpoint. **The only viable alternative in sight is formal methods verification.**

To learn more about how QVtrace can help you make the shift to formal verification, visit https://qracorp.com/qvtrace/.

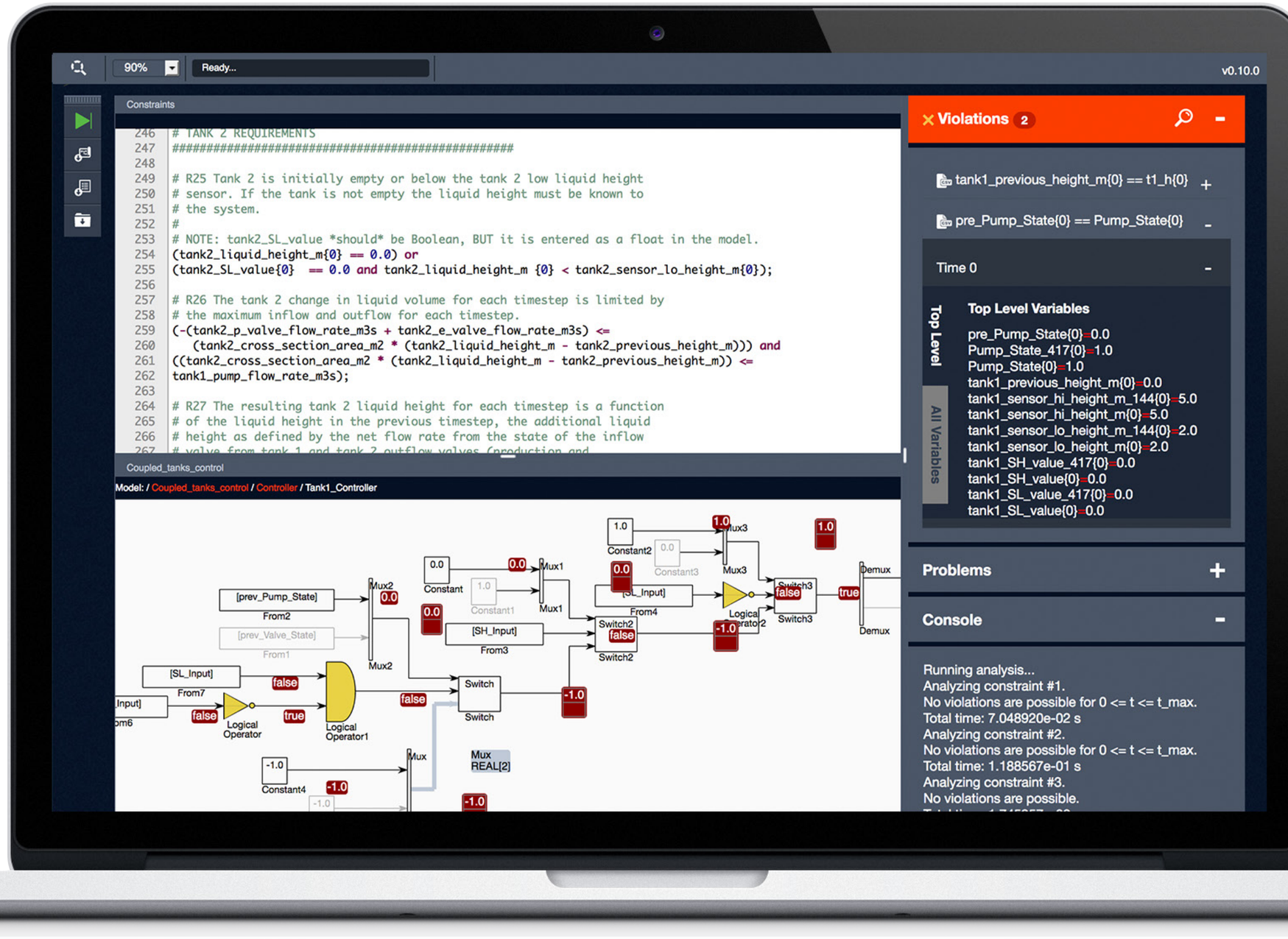
qvtrace

Build with Confidence

Try QVscribe at qracorp.com/qvtrace

[i] Hoare, C. A. R., *An Axiomatic Basis for Computer Programming*, Communications of the ACM, October 1969.

[ii] Kaivola, R. et al, *Replacing Testing with Formal Verification in Intel Core i7 Processor Execution Engine Validation*, Intel, Computer Aided Verification Conference, June 2009.



To learn more about QVtrace, visit qracorp.com/qvtrace

qracorp.com