

**A Brief Dismissal of the Fundamental Truths of Computability; Or,  
Why Formal Methods Hold Great Promise For Pre-Silicon Verification of RTL**

*by Ken Sailor*

*Verification Manager, Service Provider Division, PMC-Sierra, Inc.*

The analysis of computability is an intellectual achievement of the first rank as well as an historical curiosity. Many years before working computers were built, brilliant mathematicians discovered certain properties describing the capability and limitations of computing. These results are typically taken as a statement about what kinds of problems can be solved by computation, but the standard interpretation is mistaken. While computation about computation is limited in theory, in practice these limitations do not prevent useful and important computational analysis.

The Halting Problem is perhaps the fundamental theorem in the study of computability, a magnificent intellectual gem and a rite of passage for every student of computer science. It is a simple statement with a devilishly simple proof.

The Halting Problem states that there is no way to calculate whether a program with a specific input will loop forever or will eventually halt with a result. If you've ever sat and stared at a blank computer screen wondering if the computer is going to start responding to you again, then you know how useful such a computation would be -- you'd break the current processing and ask the computer if it were ever going to be done. If it said no, then you could cancel the calculation and move on to something more useful. A solution to the Halting Problem would be handy, indeed.

The Halting Problem has no solution, however, and this is easily proven by contradiction. Proof by contradiction, if you've forgotten, follows this form: if you assume a proposition and that assumption leads to a contradiction, then that proposition must be false. A consistent logical system cannot allow ambiguity. Every statement is true or false, not both.

So to prove the Halting Problem can't be solved let's assume it can: assume that for every program I can calculate whether a program with a given input will eventually halt, completing its computation and returning some result rather than looping forever.

Since programs can make use of other programs, I now write a curiosity of a program: this new program loops forever if its input is a program that halts on its input.

Now consider my new program when applied to itself. If it halts, then it must loop forever. If it loops forever, then it must halt. This is a contradiction because no program can both loop forever and halt.

Therefore the assumption that the Halting Problem can be solved is false: the Halting Problem is not computable.

If the proof confuses you, it's probably because you're thinking too hard -- or perhaps you were scarred by a particularly nasty lecture on computability. In reality, the proof is just as simple as it is brief. Assuming the ability to solve the Halting Problem leads quickly to a contradiction, therefore we must not have that ability.

And it gets worse. The Halting Problem is just the beginning. Rice's Theorem proves we cannot compute *any* semantic property programs. By now, anyone hoping for computational help with the creation or analysis of programs must be in despair. Surely these theorems demonstrate the impossibility of computationally-determining any useful semantic property of any program -- but that is a naïve misinterpretation of these results.

Every programmer knows from experience about infinite loops. Occasionally, even the best of us write a program that gets stuck in a loop, and finding that loop is usually not a big deal -- certainly not after the first few programs you write. In reality, we solve the Halting Problem with every program we write.

Wait, you say, we're not computers so the result doesn't apply to us. Ah, but if we use any method to find or prevent the looping, then we *are* computing the answer. My surefire method is to check that I update my loop counters.

What's going on? We're not supposed to be able to compute the Halting Problem, but somehow we're doing it all the time. How can that be? Some insight can be given by another property that is regularly computed.

A semantic property of programs widely used is type-safety, a property determined by automated type checking. A type-safe program will never suffer a runtime type error such as the garbage that results from multiplying a character string by a library record.

What is type checking? A typed programming language is one that requires each variable and function to be restricted to a specific type. For example, if I use variable  $x$  as a number in a program, then every occurrence of that  $x$  must be a number: not necessarily the same value, just the same type. Programmers of VHDL and other typed languages are familiar with typing rules and know that the compiler will not translate code until all the pesky type errors have been hunted down and eliminated. Type checking eliminates a whole class of errors since a program written in a typed language will never suffer any runtime type errors.

Users of un-typed languages like Verilog and Lisp know that it is often convenient to use variables for whatever is needed at the moment, and if they have had to debug enough code, they will know to look for variables interpreted in unexpected ways since subtle bugs can originate with such unintentional use. Type errors are often more difficult to debug than infinite loops.

But to get back to the point, how is it possible to type check given the inability to solve the Halting Problem and Rice's Theorem? The mystery has a surprising answer.

There are in the case of type checking three kinds of programs:

- 1 The programs that are recognizably type safe
- 1 The programs that will recognizably suffer a runtime type error, and
- 1 The programs that may or may not suffer a runtime error -- we just can't tell

The third kind of program is the interesting kind. Some of these programs may suffer a type error depending on their input, but many never will. The only way to tell is to run them and see if an error occurs. This is why Rice's Theorem is true while at the same time we can determine type safety: type checking can recognize some programs that exhibit the desired property, but not all that share the property. A program vetted by type checking is guaranteed to be type-safe, while a program rejected will not necessarily suffer a type error.

Type checking, as every Verilog programmer knows, is overly pessimistic. Many useful and valid programs that would never suffer a runtime error are disallowed by type checking. For this reason, many programmers reject type checking since they find it too constraining. Programmers of typed languages, however, know that type-safe programs are so numerous and easily constructed that they don't care that type checking excludes other useful programs. It's an exchange of a modest amount of freedom for safety.

The same reasoning can be applied to the Halting Problem: you *can* solve the Halting Problem for many real programs -- but among the rejected for not being guaranteed to halt there will be programs that would indeed halt. In general, nobody really cares: you can construct enough of the guaranteed programs so that you will never miss the ones that would halt but can't be recognized by the computation.

Why hasn't anyone constructed a halting inference engine? There are some obvious reasons:

- 1 As stated above, solving the Halting Problem is no big deal. Most programmers have no trouble with it
- 1 Most programming languages are semantically complex (the polite way of saying downright messy) and make such analysis difficult if not impossible
- 1 Generations of computer scientists have been trained that they can't solve the Halting Problem so why would they try?

Even though the Halting Problem is as solvable as type checking, no one has provided a solution, but things are different in the realm of semiconductor design.

Since the transformation of one chip's worth of RTL into working silicon is a long and expensive process, we want to be sure that the RTL is at least logically correct before we fabricate it. I'd love to be able to eliminate all bugs magically -- but short of that, I'm happy to have guaranteed the elimination of the bugs that can be automatically recognized. These are classes of bugs like meta-stability crossing clock domains and FIFO errors that can be hunted down using formal methods and tools.

Why have formal methods appeared for RTL analysis, but not for the Halting Problem? Compare these to my three reasons above:

- 1 Testing RTL is a big deal. It's hard to prove correct in an *ad hoc* fashion with one-off test environments. Unlike the Halting Problem, people are willing to pay to have help testing for known problems
- 1 RTL has to support deterministic and predictable synthesis, so RTL is more restricted than a general purpose programming language and semantically cleaner
- 1 Nobody told engineers that it couldn't be done! Nobody bothered to teach engineers computability. Since they didn't know it couldn't be done, it *is* being done. Useful properties are being computed on RTL

This is why I believe the formal analyses appearing as EDA tools represent a significant achievement. These methods are formally sound and proving their value in practice. Formal methods have important, possibly spectacular potential for improving the development and reliability of chips.

And who knows, computer scientists may someday wake up and see the same thing is possible for programming languages.

### **About The Author**

Ken Sailor is a verification manager in PMC-Sierra's Service Provider Division. Since joining PMC-Sierra in 1998, Mr. Sailor has been responsible for managing pre-silicon verification of semiconductor designs as well as investigating emerging methods of verification. He has over 15 years experience in the telecommunications and high-technology industries. Prior to joining PMC-Sierra, he was a founding employee of Hypercore Technologies where he helped develop a novel ATM switch. Mr. Sailor earned a doctorate in Computer Science from the University of Saskatchewan.

