Software Verification with Towers of Abstractions

Bruce W. Weide
The Ohio State University

ABSTRACT. Reasoning about software system behavior is much like reasoning about physical system behavior—except in one critical respect. Empirical observation combined with the required disclaimer that "past experience is no guarantee of future performance" is inherently part of reasoning about whether a physical system will behave as hoped: whether the bridge will stand, the airplane will fly, or the hardware will properly execute programs. On the other hand, in principle one might hope to decide by purely mathematical reasoning and formal symbolic proof whether a software system will always behave as intended, *assuming* the physical computer running it will properly execute programs. As a practical matter, though, it has been far from clear until recently that proofs of correctness of software behavior can be produced consistently except for trivial software. There is now evidence suggesting that it might soon be routinely possible to verify—fully automatically—the correct behavior of significant pieces of software. An essential feature for software verification to scale up in this way is the use of an approach that modularizes proofs of software correctness by focusing attention on no more than two adjacent levels in a "tower of abstractions". An example illustrates how software verification proceeds when the specifications of desired behavior and the programs to be executed are written in a language that explicitly incorporates such towers of abstractions.

1 Introduction

Who hasn't used a computer program that crashed at an inopportune moment? Wouldn't it be great if that would never happen again? Many people believe this is a pipe dream. They argue that it is literally impossible to rid non-trivial software of all errors, and that even if reliable and defect-free software were possible in principle it would be too expensive to produce in practice. After all, it has been argued [10] that "[s]oftware entities are more complex for their size than perhaps any other human construct". If other

engineers cannot guarantee that their artifacts will always work, then why should we expect software engineers to be able to make such guarantees for software?

Those who claim bug-free software is impossible draw two important conclusions from a single hidden assumption: that the way industrial-strength software is developed today is essentially the *only way* it could be done. They routinely assert, based on this assumption, that:

- Software must be tested in order that one should gain confidence in its correctness; but of course, there is no such thing as enough testing to show that software is entirely bug-free.
- The cost of reasoning carefully about all possible software behaviors must grow combinatorially (exponentially or probably worse) in the size of the software, e.g., the number of lines of code, and hence is infeasible for large programs.

1.1 On Testing

The shortcomings of testing—not just for software but across all of engineering—are well known. One of the pioneers of computing, Edsger W. Dijkstra, famously observed [15] that "[p]rogram testing can be used to show the presence of bugs, but never to show their absence!" The proper conclusion from this quite correct observation is that no amount of testing can hope to show software is bug-free. The improper (but popular and oft-repeated) conclusion is that it is impossible to show that software is bug-free.

The relevant question is whether testing is the only possible means of establishing software correctness. Rather than looking for software errors in the manner a civil engineer looks for weak spots that might become potholes in pavement, is it possible to reason about the behavior of a program in the manner of a mathematician proving a theorem? The theorem to be proved says, in effect, "for all permissible values of the inputs, this program produces correct outputs." A kind of mathematical reasoning like this is actually what many software professionals are taught to do while *developing* a program, though the process is rarely explained to them this way.[1] The practical difficulty is that humans are asked to carry out this reasoning only informally and in their heads. Humans make mistakes, especially when trying to process massive amounts of information such as the possible states of and paths through program code. Therefore, as a practical matter most large programs have some defects. Moreover, as noted by critics

[1] Other software professionals subscribe to "test-driven design" which, as its name implies, simply assumes testing will be used to decide whether software might be correct.

of mathematical approaches to establishing software correctness, residual errors in software professionals' reasoning inevitably escape detection now, and would continue to do so even if they were subjected to the usual social processes that historically have governed "proof" in the mathematics literature [37].

Might it be possible, though, to *formalize* the reasoning software professionals perform while developing programs, thereby permitting the use of computerized symbolic reasoning tools to catch these admittedly unavoidable human errors before software is released?

1.2 On The Combinatorial Explosion

Suppose it were possible, i.e., that we could to turn the establishment of the correct behavior of software into a mathematical rather than an empirical matter. There would remain a huge practical problem: real industrial-strength software consists of thousands or millions of lines of code. For a variety of technical reasons inherent in how most software is currently written, there are inevitably "unanticipated interactions between structural components" [57] in today's software systems. Surely, keeping track of all of them would be infeasible even for automated tools.

The relevant question then becomes whether the current languages and methods for software development are the only viable approaches to creating large software systems. Rather than designing software systems that are merely *structurally* modular (which certainly has already been achieved), might it be possible to design them to be *behaviorally* modular? This basic idea is hardly foreign to software professionals. Indeed, most would agree they try to achieve this, and some would argue they actually do. For otherwise how could mere humans possibly understand the large software systems they create? These massive and complex creations sometimes fail—but to be honest, the best of them also work correctly nearly all the time and do amazingly sophisticated things. Here is a particularly cogent explanation of the idea that underlies how software professionals try to develop software in order to maintain intellectual control over it. In this passage, "module" is taken to be "procedure" [43]:

> We reason separately about the correctness of a procedure's implementation and about parts of the program that call the procedure. To prove the correctness of a procedure definition, we show that the procedure's body satisfies its specification. When reasoning about invocations of a procedure, we use only the specification.

If the behavioral correctness of each software module truly could be reasoned about in isolation, then the cost of reasoning about an entire system

would need to grow only linearly in the number of lines of code [57]. The problem today is that even those who think they know how to achieve behavioral modularity generally cannot *always* achieve it when working within the confines of state-of-the-practice software development methods and programming languages. And even if they were able to achieve it, they generally would be unable to make a concise convincing argument that they have done so, which means their dramatically simplified reasoning about the overall software system behavior as a result of assuming behavioral modularity might not be sound.

While the combinatorial explosion involved in reasoning about overall system behavior has not been tamed by present methods of software development, then, at least it has been caged. The challenge it poses might not be nearly as imposing as originally imagined. Might we just need to tweak programming languages and software design techniques to conquer the residual threats to modular reasoning about behavior?

1.3 Prospects for Practical Automated Software Verification

This article explores the thesis that automated **software verification**—mechanized fully formalized reasoning about software behavior relative to some formalized specification of correct behavior—might be not only possible but on the verge of becoming practical. What can make this claim meaningful and believable?

First, one needs a language in which to write formal specifications of the "permissible inputs" and "correct outputs" of a program—to describe *what* the software is supposed to do, i.e., its desired behavior. Fortunately, the language of mathematics and logic is available as the foundation for describing desired behavior. Unfortunately, it is currently unrealistic to expect software engineers to write not only ordinary programming codes in a formal language, but also to write specifications in a (related but distinct) formal language. The problem lies in education, though; it is not a problem in principle. Most current software professionals have not been taught to write formal specifications using mathematical and symbolic logic notations and have not practiced this skill. Yet those who are trained and experienced at doing so find it no more difficult (and usually far less so) than writing code to meet those specifications. In fact, the more complex the software, the more likely it seems that a formal specification of behavior is more concise and simpler to write and understand than code to achieve it. So, while the challenge of writing formal specifications cannot be ignored, it is far from fatal to the vision of software verification, given that software engineers evidently are able to write very complex program code in equally formal languages. Moreover, without precise and unambiguous specifica-

tions written in some language—either informal but adequately rigorous, or formal—even testing has no hope of providing confidence that software is correct. Without such specifications, there is simply no firm standard against which to judge correctness either for testing or for verification. Finally, experience shows that the very process of formalizing specifications leads to better software designs and results in mistakes being caught earlier in the design process [44]. The vision of software verification does not exacerbate the problems of capturing informal software requirements into rigorous specifications. If anything, it helps solve them by offering well-understood conventional notations from standard mathematics in which to express those requirements as formal specifications.

Second, one needs a formal statement of the proposed algorithm (code) that transforms inputs into outputs. Fortunately, there are many computer programming languages that seem to be available as the basis for this task—to describe *how* to achieve the desired behavior. Unfortunately, programming languages in widespread use for developing industrial-strength software are not the right ones for this job. One needs a way to combine the above symbolic entities—formal specifications of behavior and code purported to achieve it—into a set of mathematical statements called **verification conditions** (VCs) whose total size grows slowly in the total size of the above entities. There are now specification-plus-programming languages that make this possible. Although they are not the languages currently used for industrial-strength software or taught in most classrooms, they are not different in kind but rather in a few essential details.

Finally, one needs a way to mechanically prove each of the VCs in order to complete the proof-of-correctness of the software. Fortunately, there has been considerable progress on this front in the past 10-15 years, partly as a result of improvements in automated theorem-proving technology [21, 22] and partly as a result of the availability of faster computers. Unfortunately, these automated provers themselves are very complex pieces of software that lead to a bootstrapping problem: who proves the correctness of the provers, and can we trust them if they are not themselves verified? It turns out one really need not trust these most complex links in the verification tool chain. Rather than worrying about possibly faulty automated theorem-proving code to *discover* a proof, one can insist that an automated theorem-prover that purports to have discovered a proof emit a proof certificate to substantiate that claim. A proof certificate is essentially a step-by-step proof in symbolic logic that can be automatically *checked* by a separate tool. Checking a putative proof is far easier than discovering a proof in the first place, so much so that verifying such a proof-checker once seems quite feasible—far easier than tasks already claimed to be accomplished,

such as verifying (with a one-time herculean effort) a compiler [39, 42] or an operating system kernel [34, 59].

One key point in the above paragraph should give reason for pause: how have faster computers (faster by a mere factor of perhaps 100) been able to make a serious difference in the ability to find proofs of mathematical statements, a problem considered computationally intractable under the best of circumstances? The situation is analogous to that facing chess-playing programs until fairly recently. It was long considered impossible that a computer could defeat a human chess champion, but chess-playing programs underwent some improvements and the computers running them became faster, even as the difficulty of playing championship chess *remained essentially fixed*. Similarly over the years, the mathematical statements that one needs to prove in order to establish software correctness have not grown fundamentally more difficult, yet the tools to prove them have advanced and the computers that run those tools have become substantially more powerful.

The conclusion is that automated software verification no longer can be dismissed as a real possibility for practical use in the near future.

1.4 Basics of Software Verification: An Example

Before moving on to the details of the above thesis, here is an example that illustrates the basic idea of software verification—for a trivial program, yes, but it is intended as an illustration of the idea. Imagine a simple imperative programming language with a programming type `Natural`, whose values are considered to be non-negative mathematical integers (initially 0) without upper bounds. There are already operations to `Increment`, `Decrement`, and compare the values of `Natural` variables, but not to add them. Figure 1 shows how one might write a new (recursive) programming operation `Add` to fill this gap. It is intended to compute the sum of the values of the parameters `n` and `m` and place the result in `n`, without changing the value of `m`. Assume there are formal symbolic specifications written using the language of mathematics to describe the behavior of `Natural` variables in the program, the behavior of the existing programming operations, and the intended behavior of `Add`—all of which will be elaborated later in this article for a more interesting example.

Software verification for such code consists of two steps:

1. Combine the above specifications and code to produce a number of putative theorems, the verification conditions, whose validity entails the correctness of the program.

2. Prove each of the VCs.

```
procedure Add (updates n: Natural, restores m: Natural)
    if IsPositive (m) then
        Decrement (m)
        Add (n, m)
        Increment (n)
        Increment (m)
    end if
end Add
```

Figure 1: Code for `Add` Operation

Prove

$m_0 - 1 < m_0$

Given

1. $0 \leq n_0$
2. $0 \leq m_0$
3. $0 \leq 0$
4. $0 < m_0$
5. $0 \leq m_0 - 1$

(a) VC #1

Prove

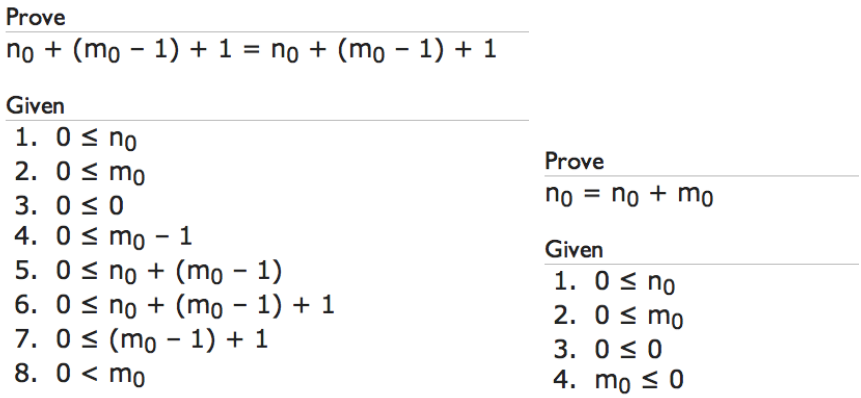
$(m_0 - 1) + 1 = m_0$

Given

1. $0 \leq n_0$
2. $0 \leq m_0$
3. $0 \leq 0$
4. $0 \leq m_0 - 1$
5. $0 \leq n_0 + (m_0 - 1)$
6. $0 \leq n_0 + (m_0 - 1) + 1$
7. $0 \leq (m_0 - 1) + 1$
8. $0 < m_0$

(b) VC #2

Figure 2



(a) VC #3 (b) VC #4

Figure 3

Figures 2 and 3 show screen-shots of the results of step 1 of software verification using our group's software verification tools. These are the result of rote syntactic translation of the code of Figure 1 and the relevant specifications into four VCs by following a set of sound and relatively complete proof rules for the specification-and-programming language used, followed by routine simplification by substitution of equals for equals. Each VC has the form of a universal statement about the variables appearing in it, which in this case are all mathematical integers. The proofs of these four VCs are left to the reader—who might notice that in some important respects they do not resemble theorems that any self-respecting mathematician would write. While VCs are technically just "little theorems" as far as a human or automated prover is concerned, the techniques needed to prove them might be somewhat different than those that would be applied to prove deep mathematical theorems posited by professional mathematicians—or even homework exercises in a textbook. A typical VC includes a number of assumptions that are not at all relevant to the conclusion to be proved and, in fact, might even overwhelm a human with so many useless details that it is difficult to separate the wheat from the chaff. It is therefore important to remember that VCs like these are expected to be proved by automated theorem-provers in step 2 and generally are not meant to be seen by humans.

The remainder of this article:

- reviews the history of software verification by identifying important

milestones, both positive and negative;

- elaborates on why software verification—not to mention fully automated software verification—has not been widely believed to be workable (e.g., see the implicit and explicit criticisms above);
- discusses in some detail why these criticisms are unfounded (e.g., see the brief responses above), including some arguments based on the conceptual differences between drawing conclusions about physical systems and about software systems; and
- demonstrates via a more interesting example what is required to make software verification scalable to non-trivial software.

There is no review of contemporary related work. This is a concession to brevity, particularly as many research groups around the world have tackled Tony Hoare's **verifying compiler grand challenge** [26, 27]. The focus instead is more tutorial in nature with respect to our group's approach to software verification, in order to match the needs of the intended audience: mathematicians, logicians, philosophers, and computer scientists for whom software verification might otherwise be viewed as a somewhat mysterious undertaking. Some of the discussion involves arguments that seem closely related to ones that arise in the philosophy of science, but there are no references to such related work, either, and no digression into secondary objections that might be raised from that direction.

The reader is assumed to have some basic grounding in computer programming, mathematics, science, and logic—and an open mind.

2 Historical and Philosophical Perspective

Before examining software verification in more technical detail, it is worth reviewing its short history as well as its connections with concepts that are somewhat more familiar to most scientists, engineers, and mathematicians.

2.1 Sources of Difficulty and Hope

The idea of formally, symbolically, and automatically proving the correctness of a program relative to a full behavioral specification can be traced to Floyd [18], Hoare [24] and King [31] only about 40 years ago. Just a few years later, a paper critical of this vision [13] produced a profoundly chilling effect on both community and financial support for software verification research, at least in the U.S. That paper argued against the practical prospects for verifying software, as follows: "[T]he theorems that arise in trying to *prove* real programs are not simple... [T]otally automated *proof* systems are out of the question..." The italics are in the original: *prove/proof*

in italics is defined by the authors as formal proof, in contrast to proof by fallible social processes. Interestingly, the claimed complexity of proving programs correct is neither as superficially compelling nor as true as the obvious objection that software verification researchers still encounter far more often: "Software correctness is undecidable." This observation presumably is intended to lead the reader/listener to the (misguided) conclusion that, therefore, software verification is a practical impossibility and any vision of it ought to be dismissed out of hand.

It is interesting to revisit [13] now in light of recent successes of automated verification of hardware and software, including among many others [30, 41, 4, 50, 60, 61, 42, 40, 59]. Whatever doubts one might have had about the feasibility or potential impact of automated verification a third of a century ago, the dream of automatically verified software can hardly be dismissed as "out of the question" today. Indeed, evidence now suggests that, contrary to the claim of [13], VCs are generally quite simple mathematically. In most situations, software verification is an enormous bookkeeping task involving VCs that modern theorem-provers often—yet still not always—can prove mechanically. Practical difficulties in verifying correct software seem to have little to do with VCs being fundamentally difficult to prove [32, 61] and nothing to do with undecidability. An important conclusion from recent software verification research is that automatically proving valid VCs that are *obvious* to mathematicians, and devising specifications that lead to such VCs also being obvious to automated provers, present the real challenges.

Why should it be the case that VCs for correct programs are "obvious"? In short, VCs capture exactly the properties that a software developer would have to be convinced are true in order to write correct code in the first place. Software professionals, typically being at best amateur mathematicians, generally do not write code that depends on new theorems that would be publishable in the mathematics literature. The formal reasoning underlying software verification merely captures the informal reasoning of the software developer and carefully checks it. Most of the time, the developer's reasoning checks out; occasionally it does not.

As might be expected, then, careful study of *failed* attempts to prove VCs resulting from typical code (e.g., [32, 55]) has been enlightening. Empirical observation to date suggests that:

- When software fails to always behave correctly when it is executed—hence software verification leads to the inability to prove a VC—it is usually traceable to mistakes arising from the fact that humans are unable to keep completely straight in their heads the overwhelming number of bookkeeping details that impact the correctness of the code they write. It is plausible that automated software verification should

be feasible and helpful in rooting out human error in software development precisely because keeping track of an enormous number of details is the sort of thing computers can do well.

- When software always behaves correctly when it is executed—yet software verification leads to the inability to prove a VC—it is usually traceable to human failure to adequately *justify* correctness [33, 61] with appropriate annotations in the code, not to insidious underlying properties such as undecidability or even inherent mathematical complexity. The unproven VC is true, is obvious to a mathematician, and yet is not sufficiently obvious to the automated prover, so its proof is not discharged mechanically. Annotations (e.g., invariants) to justify correctness might be too weak or wrong, or the mathematics on which those justifications are based might contain insufficient developments and/or results (e.g., the prover does not know about certain established theorems from the underlying mathematical domains). Other times, specifications, annotations, and supporting mathematics might be technically adequate but not engineered to match the capabilities of automated provers. In yet other cases, automated provers might simply lack the fine-tuning necessary to reach obvious conclusions.

In other words, the practical barriers to automated software verification are quite a bit less daunting than the ones that typically have been advanced by critics as insurmountable.

2.2 Mathematical Models: Physical System Behavior *vs* Software System Behavior

At its heart, software verification deals with a fundamental and somewhat philosophical question that arises across the physical sciences and engineering:

> Is a proposed mathematical model of some behavior a valid "cover story" for that phenomenon, in the sense that it permits one to analyze and predict relevant features of the phenomenon?

In physical science, a mathematical model can be a useful cover story even when it contributes nothing to causal understanding and helps explain no causal connections, but instead merely predicts certain observable features. A mathematical model of higher-level phenomena (say, the behaviors of measured properties of electrical circuits) might be related by purported explanatory or causal links to putative mathematical models of underlying lower-level phenomena (say, electron mobility and charge flux). On the other hand, that mathematical model might be treated simply as a law.

Consider Ohm's Law: $V = IR$. Here, V is the voltage across two terminals of a component in an electrical circuit, I is the current flowing through that component, and R is the resistance of the component. Without making any reductionist commitment to an explanation of what voltage and current are or how they arise from other physical phenomena, one might be willing to assume that resistance—whatever that is—is a constant for a given circuit component, and conclude from Ohm's Law that V and I for that component are proportional to one another. This is a mathematical conclusion drawn entirely from the mathematical model. That model describes no causal physical relationships to explain the origins of the measured voltage, current, or resistance, but it is consistent with empirical observations of electrical circuits and (experience has shown) can be used to predict certain features of previously unexamined cases. The scientific "truth" of Ohm's Law is based on this utility rather than on its ability to help explain the underlying physical causes of the measured quantities involved in it.

A law of this sort is, of course, not something that either a god or a legislature has decreed. It is a no more and no less than a mathematical model of the physical world that allows one to predict some measurable behaviors of the physical world. It abstracts away details that are not needed in order to make these predictions.

In (a Popperian view of) science, someone postulates such a mathematical model of a phenomenon that is consistent with observations already made. Then the broader science community searches for counter-examples to the claim that the proposed mathematical model is a valid cover story for the phenomenon. Finding no counter-examples after a long and presumably careful search, the science community tends to accept as a matter of social process that the mathematical model is a valid cover story for the behavior of interest. Confidence grows in the validity of the mathematical model in this sense so long as only corroborating evidence is found; if certain observations seem anomalous, then sometimes the model can be adjusted and thereby saved from the scrap heap of history.

If mathematical relationships between models at distinct levels can be postulated and checked, so much the better. Then, such mathematical models have the potential for leading to explanation and understanding of causal connections between physical phenomena at different levels rather than merely for making predictions at a single level [11]. But even when no such connection is made, a mathematical model can be useful in practice for its predictive power alone.

The above view pertains to the role of mathematical modeling in nearly all of science and engineering. The exception is software engineering, where there is a crucial difference: a mathematical model of software-system be-

havior may be specified entirely without regard to natural phenomena. A software system can be treated separately from the computer on which it runs, i.e., as a purely symbolic entity whose meaning (an abstract dynamic behavior) is determined by the semantics of the programming language in which it is written. Its desired behavior is similarly a purely symbolic entity whose meaning is determined by the semantics of the specification language, which is largely that of mathematics. A piece of software thus treated is, in principle, a closed system whose behavior can be completely analyzed and predicted in its own terms: the description of a mathematical model of software behavior is essentially the software itself.

Of course, proving that a software system in this sense is "correct" most certainly does not prove that, when executed on a physical computer rather than on the conceptual computational model underlying the programming language semantics, the software actually behaves as advertised. The leverage one gains in treating a software system purely conceptually is one of separation of concerns and potentially assignment of blame for defects. If software that has been proved correct does not execute correctly on a real computer, then the defect lies outside the lines of code that have been verified. It might be in the compiler, which could have generated improper code for execution by not matching the semantics of the programming language source code; or in the operating system, which could have loaded the executable code generated by the compiler into the wrong place in memory; or in the computer itself, which could have been designed or implemented or manufactured incorrectly relative to its own specification. Fortunately, it recently has been shown possible to prove (using interactive proof methods, i.e., not fully automatically) that system software such as a compiler [39, 42] or an operating system [34, 59] is correct in the same sense of not containing lines of code that are in error when executed on an idealized non-physical computer. Computer hardware designers have been using related methods to verify that symbolic representations of computer operation in their designs faithfully capture specified behavior. In other words, in principle, the entire software milieu (extending even to symbolic representations of computer hardware designs) is subject to the same sort of formal verification processes as those described in this article. Once the software verification vision is fully realized, defects in computer-based systems will have been isolated to the physical realizations of computers themselves and the physical systems they interact with.

The observation that a software system can be treated as a purely symbolic entity and reasoned about as such is behind the notion popularly called **virtual reality (VR)**. VR is a misnomer: nothing about the phenomena explained by the software engineer's mathematical models need be

even remotely "real" or "realistic". A software engineer is free to design a mathematical model of truly imaginary and/or entirely abstract behavior that would never be observable in the natural physical world, and then to try to create the desired behavior in a software system by building it on top of existing lower-level software components that have their own mathematical models of their own behaviors. Because a software professional not only *designs* the desired mathematical model of new behavior but also *designs* one or more realizations of such behavior—rather than relying on nature to have done that—understanding and explaining the precise causal connections between the specified higher-level behavior and the stipulated lower-level behaviors are vital aspects of a software engineer's job.

2.3 Towers of Abstractions

It is interesting to note what happens when students in an early computer science course are asked to think about what might be meant by "levels" in the discussion above. Consider any popular video game, e.g., Nintendo®Wii bowling. Are there bowling balls and bowling pins inside the computer? Of course not! What is inside the computer, then, that makes this bowling-like behavior? The answers typically offered are enlightening. Vectors? Numbers? Bits? None of these "exists" inside the computer in the usual sense that one could directly observe them as physical objects or entities. Instead, each of these *concepts* is nothing but a figment of a software professional's imagination. Each is but a mathematical model that can be presented to a software-system user and/or to one learning computer science, as though it did exist inside the computer, in such a compelling and logically irrefutable way that one is hard-pressed *not* to believe that each one enjoys a physical reality embodied in the hardware. We can easily forget that someone—again, a person, not a god—designed in gory detail exactly how to **represent** "higher-level" concepts in terms of "lower-level" concepts already in place: bowling balls and pins using vectors; vectors using numbers; numbers using bits; bits using ... voltages, right?

This is what one learns in a digital electronics course. Surely voltages physically exist inside the computer, don't they? Actually, there are no voltages inside the computer, either, in the sense that someone has ever directly observed a physical "voltage object" in a computer or anywhere else. Voltage is an abstract mathematical quantity that can be measured by what looks at first like a magic device and that can be related to other measurable quantities by, e.g., Ohm's Law. What really exists inside the computer that makes these voltages is ... electrons, right? No one has ever directly observed one of these, either. If one could see an electron, would

it look more like Pluto orbiting the sun or more like a wave on water? Probably not much like either; after all, these are metaphors arising from similarities between mathematical models that help predict certain aspects of the behavior of the theoretical objects physicists call electrons and the mathematical models they use for predicting certain behaviors of the solar system or of water waves. Maybe what's *really* inside the computer are even smaller entities that make up the electrons.

The above scenario illustrates a **tower of abstractions**. Strictly speaking, "tower" is too restrictive a metaphorical term selected more for the image it evokes (e.g., Figure 4) than for technical merit. In fact, each new abstraction might be built "on top of" *multiple* abstractions from "lower" layers.[2]

Figure 4: A Famous Tower

Do software professionals need to think about the answers to the physics questions listed above? Only to pass physics exams. What physically exists inside the computer is of little consequence to software engineers. All that

[2]The idea that direction in a tower of abstractions should be viewed as up/down (though only approximately in the case of Figure 4) is also arbitrary and metaphorical, but the senses of "higher" and "lower" are easily remembered: consider the term "underlying" applied to explanation and causality among physical phenomena.

matters is that the bits we *imagine* to exist inside it behave as bits are supposed to behave; similarly, the numbers, vectors, etc., that we *imagine* to exist inside it. Metaphysical questions, such as whether there might be an infinite regress at play here, are also of little consequence to software engineers. All that matters right now is the layer of immediate interest and (sometimes) the one immediately below it. The layers above and below these are simply out of the picture when reasoning about the behavior—hence the behavioral correctness—of a particular software component.

Still, it is interesting to think for another moment about the point where we relate bits to voltages as we move across those two adjacent levels in this tower of abstractions. For voltages and below, the mathematical models have been developed *a posteriori* to be consistent with the observed behavior of natural physical systems and (for the surviving models) to predict the behavior of those systems. For bits and above, the mathematical models have been developed *a priori* to create new synthetic or artificial behaviors that need not adhere to any particular laws other than those of the mathematical theories used in these mathematical models—and the laws of logic. Wii bowling pins act very much like real bowling pins, but there is no reason whatsoever they could not have been designed by software professionals to exhibit behavior that could never be seen in a real bowling alley. Some video games, in fact, are popular precisely because they do not depict a virtual reality that mimics the natural world in which we live.

In summary, then, mathematical modeling of software-system behavior is different from mathematical modeling of natural physical-system behavior in two key ways:

- The mathematical models of software-system behavior are *not limited* to describing behavior that might be observed in the natural physical world.
- The connections between mathematical models that describe behavior at two neighboring levels of a software system's tower of abstractions are always critically important: an explanation that *relates* two such levels in the software-models region of a tower of abstractions is never optional, as it might be in the physical-models region of a tower of abstractions.

2.4 Testing *vs* Verification for Towers of Abstractions

The remainder of this article deals with a special version of the earlier question in the context of towers of abstractions:

> Is a proposed mathematical model of software component/system behavior a valid "cover story" for that phenomenon, in the sense

> that it permits one to analyze and predict relevant features of the phenomenon?

This question can be approached in two distinct ways. One can search for counter-examples to the claim that the higher-level model is a valid cover story, as when using a science paradigm, by observing the behavior of software as it executes. This is known, in software engineering as in the rest of science and engineering, as testing. Alternatively, one has—for software—the option of trying to prove mathematically the claim that the higher-level mathematical model provides a valid cover story for the software's behavior, using a mathematics paradigm. This approach is known in software engineering as verification. While the term might be used elsewhere with different meanings, there is no comparable notion to software verification in natural science and engineering but only in mathematics and logic. For example, there is no way to prove mathematically that a bridge being designed by a civil engineer will not fall down. One can hope to prove that a mathematical statement of such a property holds for a particular mathematical model of the bridge, but this *proves* nothing about the physical bridge itself.

Yet in principle there might be a way to prove that software will not crash or give wrong answers, because the mathematical models involved in software behavioral specifications at two adjacent levels in a tower of abstractions are purely symbolic entities, as are the computer programs that bring together lower-level mathematical models to create behaviors consistent with a higher-level one. Mathematics and logic can directly address such questions. If it ever becomes practical to prove that a higher-level mathematical model provides a valid cover story for software-system behavior implemented by carefully relating the new mathematical model to existing lower-level mathematical models in a tower of abstractions, it inevitably will become a professional responsibility for software engineers to carry out such verification—at first for the most critical software systems, eventually for all commercial or otherwise important software.

A significant question, then, is whether automated software verification that accounts for *connections between levels in a tower of abstractions* rather than working only at *a single level of abstraction within a tower* (as in the simple example of Section 1.4) will ever become practical. The next section explains why we believe it will, first with an overview of how our research group's current software verification tools do the job, and then by using another (still reasonably simple) example as an illustration of proof-of-concept.

3 Automated Software Verification in RESOLVE

Our group's research in automated software verification directly addresses the verifying compiler grand challenge [26, 27]. This challenge effectively has been reissued from the similar vision mentioned earlier, and first expressed over 40 years ago [18, 24, 31]. Briefly paraphrased, the challenge is:

> Given a mathematical model specifying the desired behavior of new software, and like-kind specifications of existing software that can be combined to achieve that behavior, prove automatically that given program code correctly combines the latter components/systems to implement the desired behavior.

A verifying compiler should, of course, respect the crucial role of towers of abstractions. For programs written in the specification-and-programming language being compiled, it should be able to work for code that operates within a single level of a tower. It also should be able to work for code that bridges two adjacent levels in a tower—indeed, it must support this in order that a new level in the tower should be constructed correctly in the first place. If it can do both, then in principle there are no limits to the complexity of software with which it can deal. If one has the mathematical theories to model any desired software behavior, then the verifying compiler should be able to produce VCs (with content involving those theories) whose validity is tantamount to correctness of the software that is claimed to exhibit that behavior. Afterward, it should be able to prove those VCs.

3.1 The Vision

The long-term vision guiding our group's research is that of a future in which no production software is considered properly engineered unless it has been fully specified and automatically verified as satisfying these specifications. The verifying compiler grand challenge, when met, will offer significant benefits. Of course, it will not imply that verified application software will always operate perfectly: other software with which it communicates might not work properly, the hardware on which it executes might not work properly, etc. However, full behavioral specification plus modular verification that software meets its specification will imply a clear separation of concerns. For verified software, residual errors will be limited to whether the specification captures the requirements and to whether the supporting software and underlying hardware behave as advertised. Questions about the correctness of verified software components relative to their specifications will be effectively moot.

The approach used in our RESOLVE framework for software verification [53, 51] begins with respect for modular reasoning. We use the term **soft-**

ware component because nearly all software is intended to be used as a piece (or module) of another, larger, software system—which very likely is itself a software component in the same sense. Scalability of reasoning about software behavior to large software systems demands that each software component be verifiable in isolation, so its proof of correctness need not be revisited, revised, and/or redone every time the component is used in the context of a larger software system. This is the essence of modular reasoning. A much weaker notion is modular construction, in which syntactic modularity is ensured but semantic modularity is not: all the plugs and sockets "fit together", but there is no way to reason about whether an entire assemblage behaves as expected without flattening it out into its atomic pieces. Most programming languages support modular construction of software but not modular reasoning about its behavior. RESOLVE is unique among modern imperative programming languages in that it has been designed from the beginning to support modularity in both these senses.

Modular verification of a RESOLVE software component is a two-step process. First, we combine the specifications describing the component's intended behavior, the specifications of the components used in a proposed realization of that behavior, and the code that combines the latter to implement the former, generating as output a set of verification conditions (VCs, as introduced earlier). The VCs are assertions in the formal language of mathematical logic with variables, functions, and predicates ranging over the mathematical theories appearing in the specifications. The validity of these VCs implies the correctness of the code to be verified. So, the second step is to prove that the VCs are valid. The first step is done by a tool called a **verification-condition generator**, which is comparable in complexity to a normal compiler except that it generates VCs rather than executable code. The second step is done by a separate tool, an automated theorem-prover, as discussed earlier.[3] We have built our own special-purpose theorem-prover called SplitDecision [1], but we also use Isabelle/HOL [46], Z3 [14], and Dafny/Z3 [40], which have larger and more diverse user communities and are more fully developed tools. Fortunately, VCs often tend to be rather easy to prove when compared to the sorts of theorems a sophisticated tool like Isabelle can prove interactively with a professional mathematician or logician at the helm (e.g., [2]). Rather than being mathematically deep, typical VCs are generally bookkeeping jungles in which conclusions are to be proved from many (mostly irrelevant) assumptions, as seen in Figures 2 and 3. The reason a practical verifying compiler is envisioned as fully automated is that computers are better at bookkeeping tasks like this, while people are better at creative tasks. The division of labor between humans

[3]Figure 5 also shows a proof-checker tool for reasons raised in Section 1.3.

and computers should acknowledge this. Hence, for us, a verifying compiler should require no input from a software professional that looks like a "proof command".

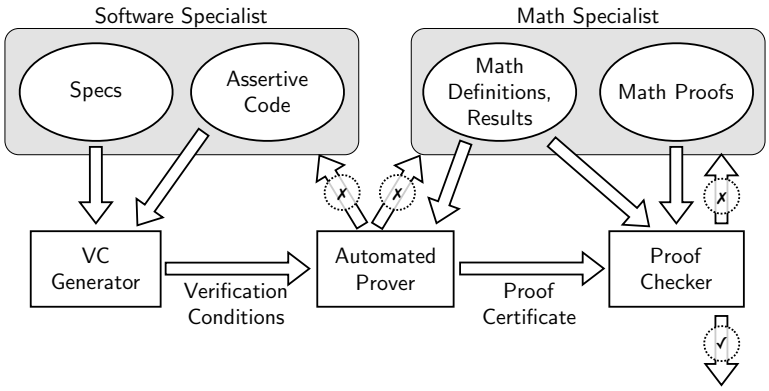


Figure 5: System Overview of a RESOLVE Verification System

A number of research groups recently have achieved considerable progress toward addressing various aspects of the verifying compiler grand challenge, e.g., [4, 8, 6, 5, 28, 48, 7, 9, 17, 38, 60, 12, 61, 29, 40]. Meanwhile, others have achieved impressive results in verifying—albeit at great expense in terms of human time spent with interactive theorem proving tools—significant pieces of potentially important software systems such as compilers [39, 42] and operating system kernels [34, 59]. We do not survey such tools or results, but instead concentrate on specific aspects of software verification using RESOLVE to illustrate how towers of abstractions play a crucial role in the process.

3.2 Proof of Correctness of Data Representation

As noted earlier, a mathematical model of higher-level phenomena in the physical world might be treated simply as a law, or it might be related to putative mathematical models of underlying lower-level phenomena. Both situations also arise in software verification. The former is the case when a new software component is an enhancement, or extension, of an existing component: it provides incremental additional functionality within a single level in a tower of abstractions. Here, mathematical modeling is homogeneous in the sense that the mathematical vocabulary involved in specifying the incremental behavior is the same as that used in specifying the behavior of the existing component it enhances. The key technical feature of this

subproblem is that the new software component introduces no new **types** (which we consider to be what are traditionally called abstract data types, or ADTs) and hence does not entail a change of mathematical theories between the specifications of an existing software component and the desired behavior of the new component. One assumes that the mathematical model of the component being enhanced is a valid cover story for its behavior, and must prove that a mathematical model involving the same mathematical theories is a valid cover story for the behavior of program code that adds a piece of new functionality to the underlying component. We have previously published a variety of verification results on this subproblem [54, 52, 58, 51].

The rest of this article outlines the RESOLVE approach to the other situation: the new component introduces a new ADT with a mathematical model that (in general) involves different mathematical theories from those used in the mathematical models of the behaviors of the underlying software components. Here, mathematical modeling is heterogeneous in the sense that the mathematical vocabulary involved in specifying the incremental behavior might not be the same as that used in specifying the behavior of the existing components. This problem is known in computer science as **proof of correctness of data representation** [25]. It is analogous to the situation faced by physical scientists and engineers when connecting two related levels in a tower of abstractions: a mathematical model proposed as a cover story for some behavioral phenomena must be related to lower-level mathematical models of underlying phenomena that are claimed to be responsible for producing the higher-level behavior to be explained.

The purported connection between a higher level and a lower level in a tower of abstractions involves some sort of "bridge law" that relates them. Focusing on how one is to interpret values of variables of a new higher-level ADT from its data representation in terms of the values of lower-level ADTs, Figure 6 uses the terminology of computer science for this situation: the **abstract state** (or abstract value) of a higher-level variable is related by an **abstraction relation** to the **concrete state**, which consists of a collection of one or more variables of lower-level ADTs.

When the mathematical models of neighboring levels in the tower involve different mathematical theories, it can be difficult to reason fully automatically about VCs that make connections between them. It is one thing to prove VCs that involve only, say, linear arithmetic over the integers (as in Figures 2 and 3). It is one thing to prove VCs that involve only, say, finite sets of objects. It is one thing to prove VCs that involve only, say, finite strings of objects. It is another thing entirely to prove VCs that involve all three of these theories. There are techniques to combine decision procedures for individual theories [45, 3, 36] under certain technical conditions.

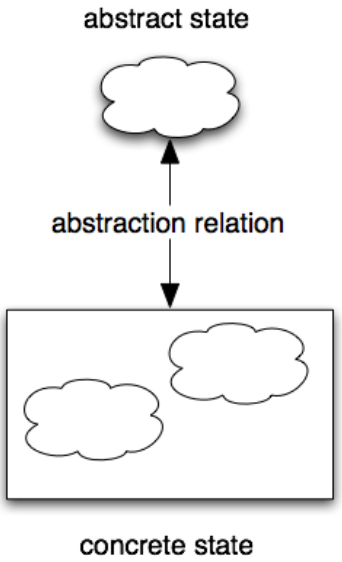


Figure 6: How a Higher-Level ADT is Related to Lower-Level ADTs

However, VCs involving lengths of strings, cardinalities of sets, mappings between strings and sets, etc., stir together all the above theories in ways that defy immediate mechanical combination of individual decision procedures (where they exist). Moreover, some of the individual theories used in software specifications are, of course, undecidable. Yet inevitably such theories will be needed in software specifications. It is simply unacceptable to limit expressiveness of specifications by insisting that all descriptions of behavior involving integers should use only linear arithmetic, on the grounds that this would make verification easier.

The approach used by Harvey Friedman [19, 20] as part of our group's software verification research has been to identify interesting fragments of theories involving integers, sets, strings, as well as partial and total functions and relations. These are the theories that commonly arise in RESOLVE software specifications. What mathematical predicates and functions are involved in actual VCs? Where quantifiers appear, which quantifier structures arise in actual VCs? How many variables do actual VCs have? Can one devise decision procedures for theory fragments that satisfy technical restrictions suggested by these empirical observations? Such questions lead to plenty of challenges for logicians. The focus turns away from the classical question of whether a full theory is decidable—because it simply isn't decidable if the theory is generally useful for modeling full functional behavior of software. Instead, the focus turns to whether the restricted forms of actual VCs make it decidable: structural restrictions on the sentences to be proved, size or complexity restrictions on those sentences, etc.

Even within our own group, there is no consensus that new decision procedures for such theory fragments are the best way to go in order to improve automated theorem-provers for software verification. Would it be better to seek powerful but necessarily heuristic automated general-purpose proof techniques that work well on actual VCs? No one knows yet. It is worthwhile to explore both approaches.

3.3 Example: A `Set` ADT Represented Using a `Queue` ADT

We now consider another relatively simple example. The earlier example (Figure 1) showed an enhancement to add new functionality to an existing ADT. This one illustrates how proofs of correctness of data representation arise and are carried out in the RESOLVE framework. RESOLVE has the standard imperative programming control constructs and basic constructs supporting component-based software design. The first major difference between RESOLVE and a typical industrial language like Java is that, in RESOLVE, we program over arbitrary mathematical spaces: the type of each variable in the program is an ADT, also called its **program type**,

and each program type has an associated mathematical model that determines the variable's **mathematical type** for purposes of reasoning about the values the variable might have during program execution.[4] Software engineers barely realize they are following exactly this approach when they program with a built-in **value type** in a language like Java. The value of a variable of program type `int` in Java, say, is treated for reasoning purposes as if it were a mathematical integer between some bounds, not as a block of 32 bits in computer memory—and certainly not as a vector of 32 voltages or as a particular configuration of electrons inside a memory chip. The reason, of course, is that it would be far more difficult to reason about the behavior of programs that use `int`s if their (incidental) data representation in terms of bits were not hidden behind the cover story that `int`s *behave* like mathematical integers whose values are constrained to lie within certain bounds. RESOLVE applies this approach to all variables of all program types, so *every* variable ranges over the values of its program type's mathematical model. This distinguishes it from languages like Java, where a variable of a programmer-defined type—a so-called **reference type**—is thought of as ranging over references to, or addresses of, memory locations where data representations are stored during program execution.

The required modularity of reasoning for scalable software verification is actually quite difficult to achieve because of a number of common programming practices that routinely appear in today's industrial software by virtue of programming languages with features that are too permissive. The interested reader may consult [56] for details about a chief impediment to modularity: interference via aliased references. A language with reference types inevitably permits two variables to have the same (reference) value, which is known as **aliasing**. The most popular approach to reasoning about references in current verification systems is separation logic [49, 47]. Separation logic has been shown viable for dealing with references that are aliased *within* a component, but not for dealing with references that are aliased *across component boundaries*. A recent paper focusing on reasoning about such cross-boundary interference [16] concludes: "Fundamentally, because it concerns aliased pointers, freedom from interference is extremely difficult to protect against using programming language restrictions, and too expensive to protect against with runtime checking. It is better to say that *if* there is no interference *then* refinement reasoning is sound, rather than to say that it is unconditionally sound." Suffice to say that, despite protestations

[4] The mathematical model, or mathematical type, associated with each program type justifies the term "abstract data type". Rather than being based on how a program type is represented, reasoning is raised up one level in the tower of abstractions to something that is presumably easier to reason about than the underlying data representation would be.

that it is "difficult to protect against" such interference, the RESOLVE language nevertheless does just that. In summary, the RESOLVE approach to verification is modular because the RESOLVE language has been carefully designed to support modular reasoning about software component behavior in this respect (as well as others).

The specification of the example higher-level software component to be realized and verified is shown in Figure 7. This component provides client programmers with a new ADT, `Set`, and operations by which to manipulate `Set` variables. The mathematical model of a `Set` is a finite mathematical set of objects of some type `Item` (whose own mathematical model is undetermined at this point, but fixed, and is known by the name `Item` in mathematical contexts as well). So, for example, the **`initialization ensures`** clause means that the value of any newly declared programming variable of type `Set` is initially the empty set. Operations are specified in the generally accepted software design style known as design-by-contract: a **`requires`** clause introduces a precondition that must be true in the client (calling) program in order that the operation call is legal, and an **`ensures`** clause introduces a postcondition that the component implementer guarantees to be true in the client program upon return from the call. In a postcondition, a variable decorated with a prefix # denotes the value of that variable at the time of a call to the operation, where the undecorated version denotes its value upon return from the operation. So, for example, the `Add` operation has two parameters: a `Set s` (whose values $\#s$ and s are considered to be finite mathematical sets of `Item`s) and an `Item x` (whose values $\#x$ and x are considered to be whatever the specification for the actual type `Item` declares its mathematical model to be). In code that uses this component, the `Add` operation may be called whenever $x \notin s$. If this is true at the time of the call, then when the call returns, it will be the case that $s = \#s \bigcup \{\#x\}$.

The code to be verified uses a single `Queue` to represent a `Set`. The specification for the existing lower-level component, `QueueTemplate`, is shown in Figure 8; the code that uses it to implement the higher-level `SetTemplate` specification is shown (split into two parts for typesetting purposes) in Figures 9 and 10. It can be seen that the specification in Figure 8 is similar in kind to that in Figure 7—but the theory involved in this specification is not the mathematics of finite sets but rather that of finite strings. Intuitively, strings record order and allow multiplicity (repetition), while sets do not. This suggests designing the obvious relationship between the two levels: the value of a `Set` variable at the higher level is represented by having a `Queue` containing exactly one copy of each of the elements of the `Set` variable's value, in arbitrary order.

```
contract SetTemplate (type Item)

    uses UnboundedIntegerFacility

    math subtype SET_MODEL is finite set of Item

    type Set is modeled by SET_MODEL
        exemplar s
        initialization ensures
            s = { }

    procedure Add (updates s: Set, clears x: Item)
        requires
            x is not in s
        ensures
            s = #s union {#x}

    procedure Remove (updates s: Set, restores x: Item,
                      replaces xCopy: Item)
        requires
            x is in s
        ensures
            s = #s \ {x}  and  xCopy = x

    procedure RemoveAny (updates s: Set, replaces x: Item)
        requires
            s /= { }
        ensures
            x is in #s  and  s = #s \ {x}

    function Contains (restores s: Set, restores x: Item): control
        ensures
            Contains = (x is in s)

    function IsEmpty (restores s: Set): control
        ensures
            IsEmpty = (s = { })

    function Size (restores s: Set): Integer
        ensures
            Size = |s|

end SetTemplate
```

Figure 7: `SetTemplate` Specification

```
contract QueueTemplate (type Item)

    uses UnboundedIntegerFacility

    math subtype QUEUE_MODEL is string of Item

    type Queue is modeled by QUEUE_MODEL
        exemplar q
        initialization ensures
            q = < >

    procedure Enqueue (updates q: Queue, clears x: Item)
        ensures
            q = #q * <#x>

    procedure Dequeue (updates q: Queue, replaces x: Item)
        requires
            q /= < >
        ensures
            #q = <x> * q

    function IsEmpty (restores q: Queue): control
        ensures
            IsEmpty = (q = < >)

    function Length (restores q: Queue): Integer
        ensures
            Length = |q|

end QueueTemplate
```

Figure 8: `QueueTemplate` Specification

```
realization QueueRealization (
        function AreEqual (restores i: Item, restores j: Item): control
            ensures
                AreEqual = (i = j)
    ) implements SetTemplate

    uses QueueTemplate
    uses Concatenate for QueueTemplate
    uses IsPositive for UnboundedIntegerFacility

    facility QueueFacility is QueueTemplate (Item)
        enhanced by Concatenate

    type representation for Set is (
            items: Queue
        )
        exemplar s
        convention
            |s.items| = |elements (s.items)|
        correspondence function
            elements (s.items)
    end Set

    local procedure SplitAndExtract (updates q1: Queue,
            restores x: Item, replaces q2: Queue, replaces y: Item
        )
        requires
            |q1| > 0
        ensures
            #q1 = q2 * <y> * q1  and
            (if x is in elements (#q1) then y = x)

        Clear (q2)
        Dequeue (q1, y)
        loop
            maintains
                q2 * <y> * q1 = #q2 * <#y> * #q1  and
                x is not in elements (q2)  and  x = #x
            decreases |q1|
        while not IsEmpty (q1) and not AreEqual (x, y) do
           Enqueue (q2, y)
           Dequeue (q1, y)
       end loop
    end SplitAndExtract

    procedure Add (updates s: Set, clears x: Item)
        Enqueue (s.items, x)
    end Add
```

Figure 9: QueueTemplate Specification, part 1

```
    procedure Remove (updates s: Set, restores x: Item,
                      replaces xCopy: Item)
        variable tmp: Queue
        SplitAndExtract (s.items, x, tmp, xCopy)
        Concatenate (s.items, tmp)
    end Remove

    procedure RemoveAny (updates s: Set, replaces x: Item)
        Dequeue (s.items, x)
    end RemoveAny

    function Contains (restores s: Set, restores x: Item): control
        variable qLength: Integer
        qLength := Length (s.items)
        if IsPositive (qLength) then
            variable y: Item
            variable tmp: Queue
            SplitAndExtract (s.items, x, tmp, y)
            Contains := AreEqual (x, y)
            Enqueue (s.items, y)
            Concatenate (s.items, tmp)
        end if
    end Contains

    function IsEmpty (restores s: Set): control
        IsEmpty := IsEmpty (s.items)
    end IsEmpty

    function Size (restores s: Set): Integer
        Size := Length (s.items)
    end Size

end QueueRealization
```

Figure 10: `QueueTemplate` Specification, part 2

The function **elements** maps a string to the set of its entries, thereby effectively removing information about the ordering and multiplicity of string entries. In addition, the (overloaded) set cardinality operator $|\bullet|$ and string length operator $|\bullet|$ are both integer-valued functions. This leads to a situation where most of the VCs arising from the code in Figures 9 and 10 simultaneously involve finite sets, strings, and integers, and mathematical functions and predicates from these theories—unions of finite sets, concatenation * of strings, set $\{\bullet\}$ and string $< \bullet >$ constructors from entries, etc. The design-by-contract approach with explicit mathematical modeling, in which programming variables' values range over their stated mathematical models, underlies VC generation. No further details are provided here because the present focus is on the nature of the VCs themselves rather than on how they are generated or why the rules for generating them are sound and relatively complete; the reader interested in such details is referred to [35, 23, 52, 51].

The code for `QueueRealization` is similar to the code one might write in any modern object-based imperative programming language. The primary difference is that the programmer of this code has had to write a few extra lines—of mathematics, not ordinary programming code. The two most important of these for proof of correctness of data representation are the **correspondence function** clause, which introduces the **abstraction function**; and the **convention** clause, which introduces the **representation invariant**. The former clause is the interpretation mapping that says precisely how the value of the higher-level `Set` variable, as described in the `SetTemplate` specification this code purports to implement, can be obtained from the lower-level `Queue` that represents it. The latter clause characterizes the domain of this mapping.[5] In addition, the programmer has had to write a contract for the local operation `SplitAndExtract` that is used as a "helper" operation within `QueueRealization`, as well as a loop invariant for the loop in that operation (the **maintains** clause) and a justification that the loop terminates (the **decreases** clause).

For the code of Figures 9 and 10, the RESOLVE VC generator emits 37 VCs, as summarized in Table 1. All 37 are proved automatically by SplitDecision, which is the least sophisticated of the back-end automated theorem-provers we use. SplitDecision does the entire job in a total of just over one second.

A couple of the most difficult VCs, where difficulty is based on proof time by SplitDecision, are shown in the screen-shots of Figures 11 and 12.

[5] The interpretation mapping in RESOLVE is currently limited to being a function to simplify certain aspects of the verification tools, though it is known that in general it might need to be a relation.

Table 1: VC Summary for `QueueRealization`

Source of VCs	Number of VCs
type representation for `Set` (initialization)	2
`SplitAndExtract`	11
`Add`	2
`Remove`	7
`RemoveAny`	4
`Contains`	9
`IsEmpty`	2
`Size`	0
Total for `QueueRealization`	37

Verification Condition #2 (state index: 3, ensures clause)

Prove

$\text{elements}\ (s.\text{items}_2 \circ \text{tmp}_2) = \text{elements}\ (\text{tmp}_2 \circ \langle x_0 \rangle \circ s.\text{items}_2) \setminus \{x_0\}$

Given

1. $x_0 \in \text{elements}\ (\text{tmp}_2 \circ \langle x_0 \rangle \circ s.\text{items}_2)$
2. $|\text{tmp}_2 \circ \langle x_0 \rangle \circ s.\text{items}_2| = |\text{elements}\ (\text{tmp}_2 \circ \langle x_0 \rangle \circ s.\text{items}_2)|$

Figure 11: A relatively difficult VC arising from `QueueRealization`

Verification Condition #9 (state index: 10, convention)

Prove

$|s.\text{items}_6 \circ \langle y_6 \rangle \circ \text{tmp}_6| = |\text{elements}\ (s.\text{items}_6 \circ \langle y_6 \rangle \circ \text{tmp}_6)|$

Given

1. $|\text{tmp}_6 \circ \langle y_6 \rangle \circ s.\text{items}_6| = |\text{elements}\ (\text{tmp}_6 \circ \langle y_6 \rangle \circ s.\text{items}_6)|$
2. $\text{is\_initial}\ (y_4)$
3. $x_0 \notin \text{elements}\ (\text{tmp}_6 \circ \langle y_6 \rangle \circ s.\text{items}_6)$
4. $\text{is\_initial}\ (y_8)$
5. $0 < |\text{tmp}_6 \circ \langle y_6 \rangle \circ s.\text{items}_6|$

Figure 12: Another relatively difficult VC arising from `QueueRealization`

The VC in Figure 11, which comes from the code for `Remove`, takes SplitDecision about 100 milliseconds to prove. It can be seen that this VC is not entirely trivial. Intuitively, its truth rests on interpreting Given #2 as a statement that there are no duplicate entries in a particular string. With this observation, an informal argument that the VC is true can be made by someone who knows what all the symbols[6] mean. The formal symbolic proof carried out by an automated prover requires not only Given #2 but also some knowledge relating strings and sets, e.g., a lemma about commutativity and associativity of string concatenation in the argument of **`elements`**, and/or expansion of elements of a concatenation of strings into a union of sets, etc.

The VC in Figure 12 comes from the code for `Contains`, and also takes SplitDecision about 100 milliseconds to prove. Its human proof is left to the reader.

The point of showing these VCs is that the human who developed this code was able to write it correctly without relying on any deep mathematical properties of sets, strings, or integers. It is little wonder, then, that the VCs needed to establish that the code is correct—fully formally and symbolically—do not demand deep mathematical insights for their proofs. We have done similar but more systematic analyses of hundreds of other VCs arising from the verifications of software components that are similarly representative of the kind of code written daily by thousands of software professionals [32, 55]. While not all are proved automatically, even the most difficult we have encountered are rather easily proved by humans in a matter of a few minutes of thinking, much of which involves wading through a sea of assumptions for the few (often just one or two) that are relevant to proving the conclusion.

4 Conclusion

Proofs of correctness of routine software inevitably will involve routine mathematical reasoning. Of course, not all software is routine in this sense. Software intended to be used, say, in support of a proof of the four-color conjecture might well be based on some deep mathematical results. Yet even here it is not likely that such results will appear spontaneously in the form of particularly tricky VCs. Instead, there will be some VCs whose proofs will rely on these deep mathematical results being proved off-line, with human assistance, and then invoked as lemmas in the proofs of VCs arising from that code. Such proofs are likely to be based on a reasonably small set

[6] In this human-readable view of VCs displayed by the RESOLVE tools, string concatenation is written using $\circ$, where the same operator appears as $*$ in specifications typed in by the software developer; set difference is denoted in both places by $\backslash$.

of mathematical lemmas about the mathematical theories involved in the software specifications. We now have strong empirical evidence of this [55], but the case can be argued directly as follows. Mathematics that is difficult for—let alone unknown to—the software developer is highly unlikely to be needed to prove VCs arising from that programmer's code.

There is, of course, considerable bookkeeping involved in proving some VCs. We have seen VCs with dozens of assumptions, all but a few of which are irrelevant to the proof of the conclusion (e.g., all but Given #1 in Figure 12). These few happen to be the ones the developer really needed to think about when writing the code. We therefore conjecture that automated theorem-provers that are able to order assumptions into decreasing likelihood of relevance to the proof, and to assume that if there are many then most of them are irrelevant, will rarely be distracted and/or run out of time or memory while carrying out proofs arising from software verification. This, combined with Friedman's research program of looking for decision procedures for theories that are restricted in ways consistent with actual VCs observed in practice, suggests that the logic problems to be addressed to make automated software verification practical are not exactly the same as those traditionally studied by logicians. To paraphrase Friedman, there seems to be quite a bit of fundamental logic "red meat" lurking in this realm of applied logic. If progress can be made on such questions, then a practical verifying compiler will not be far behind.

5 Acknowledgments

I am delighted to acknowledge the special contributions of Bruce Adcock, Jeremy Avigad, Derek Bronish, Paolo Bucci, Harvey M. Friedman, Wayne Heym, Jason Kirschenbaum, Bill Ogden, Murali Sitaraman, Hampton Smith, Aditi Tagore, Diego Zaccai, and the anonymous referees. This material is based upon work supported by the National Science Foundation under Grants No. DMS-0701260, CCF-0811737, ECCS-0931669, and DUE-0942542. Any opinions, findings, conclusions, or recommendations expressed here are those of the author and do not necessarily reflect the views of the National Science Foundation.

BIBLIOGRAPHY

[1] B. Adcock. *Working Towards the Verified Software Process.* PhD thesis, Computer Science and Engineering, The Ohio State University, 2010.

[2] J. Avigad, K. Donnelly, D. Gray, and P. Raff. A formally verified proof of the prime number theorem. *ACM Trans. Comput. Logic*, 9, December 2007.

[3] J. Avigad and H. M. Friedman. Combining decision procedures for the reals. *Logical Methods in Computer Science*, 2:1–42, 2006.

[4] J. Barnes. *High Integrity Software: The SPARK Approach to Safety and Security.* Addison-Wesley Longman, 2003.

[5] M. Barnett, B.-Y. Chang, R. DeLine, B. Jacobs, and K. Leino. Boogie: a modular reusable verifier for object-oriented programs. In *Formal Methods for Components and Objects*, volume 4111 of *Lecture Notes in Computer Science*, pages 364–387. Springer, 2006.

[6] M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# programming system: An overview. In *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*, volume 3362 of *Lecture Notes in Computer Science*. Springer, 2005.

[7] B. Beckert, R. Hähnle, and P. H. Schmitt. *Verification of Object-Oriented Software: The KeY Approach*, volume 4334 of *Lecture Notes in Computer Science*. Springer, 2007.

[8] Y. Bertot and P. Casteran. *Interactive Theorem Proving and Program Development*. Springer, 2004.

[9] A. Bradley and Z. Manna. *The Calculus of Computation: Decision Procedures with Applications to Verification*. Springer, 2007.

[10] F. P. Brooks, Jr. *The Mythical Man-Month (Anniversary Edition)*. Addison-Wesley Longman, 1995.

[11] B. Chandrasekaran and J. R. Josephson. Function in device representation. *Engineering with Computers*, 16:162–177, 2000.

[12] E. Cohen, M. Dahlweid, M. A. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, and S. Tobies. VCC: A practical system for verifying concurrent C. In *Proceedings of 22nd International Conference on Theorem Proving in Higher Order Logics*, volume 5674 of *Lecture Notes in Computer Science*, pages 23–42. Springer, 2009.

[13] R. A. DeMillo, R. J. Lipton, and A. J. Perlis. Social processes and proofs of theorems and programs. *Commun. ACM*, 22:271–280, May 1979.

[14] L. DeMoura and N. Bjørner. Z3: an efficient SMT solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.

[15] E. W. Dijkstra. Notes on structured programming. Technical Report T.H.-Report 70-WK-03 (EWD249), Technological University Eindhoven, The Netherlands, April 1970.

[16] I. Filipovic, P. O'Hearn, N. Torp-Smith, and H. Yang. Blaming the client: on data refinement in the presence of pointers. *Formal Aspects of Computing*, 22:547–583, September 2010.

[17] J. C. Filliâtre and C. Marché. The Why/Krakatoa/Caduceus platform for deductive program verification. In *Proceedings of the 19th International Conference on Computer Aided Verification*, volume 4590 of *Lecture Notes in Computer Science*, pages 173–177. Springer, 2007.

[18] R. Floyd. Assigning meanings to programs. In *Proc. of Symposium on Applied Mathematics*, volume 19, pages 19–32. AMS, 1967.

[19] H. M. Friedman. Decidability and undecidability involving string replacement. Technical Report OSU-CISRC-8/09-TR43, Computer Science and Engineering, The Ohio State University, 2009.

[20] H. M. Friedman. Deciding statements about strings with applications to program verification. Technical Report OSU-CISRC-8/09-TR42, Computer Science and Engineering, The Ohio State University, 2009.

[21] J. Harrison. Formal proof – theory and practice. *Notices of the American Mathematical Society*, 55:1395–1406, 2008.

[22] J. Harrison. *Handbook of Practical Logic and Automated Reasoning*. Cambridge University Press, 1st edition, 2009.

[23] W. D. Heym. *Computer Program Verification: Improvements for Human Reasoning*. PhD thesis, Computer and Information Science, The Ohio State University, Columbus, OH, 1995.

[24] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12:576–580, October 1969.
[25] C. A. R. Hoare. Proof of correctness of data representations. *Acta Inf.*, 1:271–281, 1972.
[26] C. A. R. Hoare. The verifying compiler: A grand challenge for computing research. *J. ACM*, 50(1):63–69, 2003.
[27] C. A. R. Hoare, J. Misra, G. T. Leavens, and N. Shankar. The verified software initiative: A manifesto. *ACM Comput. Surv.*, 41:22:1–22:8, October 2009.
[28] D. Jackson. *Software Abstractions: Logic, Language, and Analysis*. The MIT Press, 2006.
[29] B. Jacobs, J. Smans, and F. Piessens. VeriFast: Imperative programs as proofs. In *Workshop Proceedings of the 3rd International Conference on Verified Software: Theories, Tools, Experiments*, August 2010.
[30] M. Kaufmann and J. S. Moore. An industrial strength theorem prover for a logic based on common lisp. *IEEE Trans. Softw. Eng.*, 23:203–213, April 1997.
[31] J. C. King. *A Program Verifier*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA, 1970.
[32] J. Kirschenbaum, B. Adcock, D. Bronish, H. Smith, H. Harton, M. Sitaraman, and B. W. Weide. Verifying component-based software: Deep mathematics or simple bookkeeping? In *Proceedings of the 11th International Conference on Software Reuse: Formal Foundations of Reuse and Domain Engineering*, volume 5791 of *Lecture Notes in Computer Science*, pages 31–40. Springer, 2009.
[33] J. Kirschenbaum, H. K. Harton, and M. Sitaraman. A case study in automated, modular, and full functional verification. In *Proc. AFM '08: Third Workshop on Automated Formal Methods*, pages 53–58. ACM Press, July 2008.
[34] G. Klein, J. Andronick, K. Elphinstone, G. Heiser, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: formal verification of an operating-system kernel. *Commun. ACM*, 53:107–115, June 2010.
[35] J. E. Krone. *The Role of Verification in Software Reusability*. PhD thesis, Computer and Information Science, The Ohio State University, 1988.
[36] V. Kuncak, H. H. Nguyen, and M. Rinard. Deciding Boolean algebra with Presburger arithmetic. *J. Autom. Reason.*, 36:213–239, April 2006.
[37] I. Lakatos. *Proofs and Refutations: The Logic of Mathematical Discovery*. Cambridge University Press, 1976.
[38] G. T. Leavens. Tutorial on JML, the Java Modeling Language. In *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering*, pages 573–573. ACM Press, 2007.
[39] D. Leinenbach, W. Paul, and E. Petrova. Towards the formal verification of a C0 compiler: Code generation and implementation correctness. In *Proceedings of the Third IEEE International Conference on Software Engineering and Formal Methods*, pages 2–12. IEEE Computer Society, 2005.
[40] K. R. M. Leino. Dafny: An automatic program verifier for functional correctness. In *16th International Conference on Logic Programming for Artificial Intelligence and Reasoning (LPAR-16)*, 2010.
[41] K. R. M. Leino, G. Nelson, and J. Saxe. ESC/Java user's manual. Technical Report 2000-002, Compaq Systems Research Center, 2000.
[42] X. Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52:107–115, July 2009.
[43] B. Liskov and J. Guttag. *Abstraction and Specification in Program Development*. MIT Press, 1986.
[44] B. Meyer. On formalism in specifications. *IEEE Software*, 2(1):6–26, Jan. 1985.
[45] G. Nelson and D. C. Oppen. Simplification by cooperating decision procedures. *ACM Trans. Program. Lang. Syst.*, 1(2):245–257, 1979.

[46] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL—A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002.

[47] P. W. O'Hearn, H. Yang, and J. C. Reynolds. Separation and information hiding. *ACM Trans. Program. Lang. Syst.*, 31:11:1–11:50, April 2009.

[48] E. Poll, P. Chalin, D. Cok, J. Kiniry, and G. T. Leavens. Beyond assertions: Advanced specification and verification with JML and ESC/Java2. In *Formal Methods for Components and Objects (FMCO) 2005, Revised Lectures*, volume 4111 of *Lecture Notes in Computer Science*, pages 342–363. Springer, 2006.

[49] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science*, pages 55–74. IEEE Computer Society, 2002.

[50] P. Ross. The exterminators. *IEEE Spectrum*, 42:36–41, 2005.

[51] M. Sitaraman, B. Adcock, J. Avigad, D. Bronish, P. Bucci, D. Frazier, H. Friedman, H. Harton, W. Heym, J. Kirschenbaum, J. Krone, H. Smith, and B. Weide. Building a push-button RESOLVE verifier: Progress and challenges. *Formal Aspects of Computing*, 23(5):607–626, 2011.

[52] M. Sitaraman, S. Atkinson, G. Kulczycki, B. W. Weide, T. J. Long, P. Bucci, W. D. Heym, S. M. Pike, and J. E. Hollingsworth. Reasoning about software-component behavior. In *Software Reuse: Advances in Software Reusability, 6th International Conference*, volume 1844 of *Lecture Notes in Computer Science*, pages 266–283. Springer, 2000.

[53] M. Sitaraman and B. W. Weide. Component-based software using RESOLVE. *SIGSOFT Softw. Eng. Notes*, 19:21–67, October 1994.

[54] M. Sitaraman, B. W. Weide, and W. F. Ogden. On the practical need for abstraction relations to verify abstract data type representations. *IEEE Trans. Softw. Eng.*, 23:157–170, March 1997.

[55] A. Tagore, D. Zaccai, and B. Weide. To expand or not to expand: Automatically verifying software specified with complex mathematical definitions. Technical Report OSU-CISRC-5/11-TR18, Computer Science and Engineering, The Ohio State University, September 2011.

[56] B. W. Weide and W. D. Heym. Specification and verification with references. In *Proc. OOPSLA Workshop on Specification and Verification of Component-Based Systems*, 2001.

[57] B. W. Weide, W. D. Heym, and J. E. Hollingsworth. Reverse engineering of legacy code exposed. In *Proceedings of the 17th International Conference on Software Engineering*, pages 327–331. ACM, 1995.

[58] B. W. Weide, M. Sitaraman, H. K. Harton, B. Adcock, P. Bucci, D. Bronish, W. D. Heym, J. Kirschenbaum, and D. Frazier. Incremental benchmarks for software verification tools and techniques. In *Proceedings of the 2nd International Conference on Verified Software: Theories, Tools, Experiments*, volume 5295 of *Lecture Notes in Computer Science*, pages 84–98. Springer, 2008.

[59] J. Yang and C. Hawblitzel. Safe to the last instruction: automated verification of a type-safe operating system. In *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 99–110. ACM Press, 2010.

[60] K. Zee, V. Kuncak, and M. Rinard. Full functional verification of linked data structures. In *Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 349–361. ACM Press, 2008.

[61] K. Zee, V. Kuncak, and M. C. Rinard. An integrated proof language for imperative programs. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 338–351. ACM Press, 2009.