Part I

# Background

# CHAPTER 1

# Introduction

"Behold, You desire truth* in the inward parts,

And in the hidden part You will make me to know wisdom."

— Psalm 51:6 [1]

* "**truth,** *'emet* (*eh*-met); Strong's #571: Certainty, stability, truth, rightness, trustworthiness. *'Emet* derives from the verb *'aman*, meaning "to be firm, permanent, and established." *'Emet* conveys a sense of dependability, firmness, and reliability. Truth is therefore something upon which a person may confidently stake his life."

— The Spirit-Filled Life Bible, Thomas Nelson Publishers, 1991, page 774.

Good software is very difficult to produce. This contradicts expectations, for building software requires no large factories or furnaces, ore or acres. It consumes no rare, irreplaceable materials, and generates no irreducible waste. It requires no physical agility or grace, and can be made in any locale.

What good software does require, it demands of the intelligence and character of the person who makes it. These demands include patience, perseverance, care,

[1] All quotations from the Bible are taken from the New King James Version, copyright ©1991 Thomas Nelson, Inc., unless otherwise indicated.

craftsmanship, attention to detail, and a streak of the detective, for hunting down errors. Perhaps most central is an ability to solve problems logically, to resolve incomplete specifications to consistent, effective designs, to translate nebulous descriptions of a program's purpose to definite detailed algorithms. Finally, software remains lifeless and mundane without a well-crafted dose of the artistic and creative.

Large software systems often have many levels of abstraction. Such depth of hierarchical structure implies an enormous burden of understanding. In fact, even the most senior programmers of large software systems cannot possibly know all the details of every part, but rely on others to understand each particular small area.

Given that creating software is a human activity, errors occur. What is surprising is how difficult these errors often are to even detect, let alone isolate, identify, and correct. Software systems typically pass through hundreds of tests of their performance without flaw, only to fail unexpectedly in the field given some unfortunate combination of circumstances. Even the most diligent and faithful applications of rigorous disciplines of testing only mitigate this problem. The core remains, as expressed by Dijkstra: "Program testing can be used to show the presence of bugs, but never to show their absence!" [Dij72] It is a fact that virtually every major software system that is released or sold is, not merely suspected, but in fact *guaranteed* to contain errors.

This degree of unsoundness would be considered unacceptable in most other fields. It is tolerated in software because there is no apparent alternative. The resulting erroneous software is justified as being "good enough," giving correct answers “most of the time,” and the occasional collapses of the system are shrugged off as inevitable lapses that must be endured. Virtually every piece of software that is sold for a personal computer contains a disclaimer of *any* particular performance at all. For example, the following is *typical*, not extraordinary:

> “X” CORPORATION PROVIDES THIS SOFTWARE “AS IS” WITHOUT ANY WARRANTEE OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE IMPLIED WARRANTIES OR CONDITIONS OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. IN NO EVENT SHALL “X” CORPORATION BE LIABLE FOR ANY LOSS OF PROFITS, LOSS OF BUSINESS, LOSS OF USE OR DATA, INTERRUPTION OF BUSINESS, OR FOR INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES OF ANY KIND, EVEN IF “X” CORPORATION HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES ARISING FROM ANY DEFECT OR ERROR IN THIS SOFTWARE.

The limit to which many companies stand behind their software is to promise to reimburse the customer the price of a floppy disk, if the physical medium is faulty. This means that the customer must hope and pray that the software performs as advertised, for he has no firm assurance at all. This lack of responsibility is not tolerated in most other fields of science or business. It is tolerated here because it is, for all practical purposes, impossible to actually create perfect software of the size and complexity desired, using the current technology of testing to detect errors.

There is a reason why testing is inadequate. Fundamentally, testing examines a piece of software as a “black box,” subjecting it to various external stimuli, and observing its responses. These responses are then compared to what the tester expected, and any variation is investigated. Testing depends solely on what is externally visible. This approach treats the piece of software as a mysterious locked chest, impenetrable and opaque to any deeper vision or understanding of its internal behavior. A good tester does examine the software and study its structure in order to design his test cases, so as to test internal paths, and check circumstances around boundary cases. But even with some knowledge of the internal structure, it is very difficult in many cases to list a sufficient set of cases that will exhaustively test all paths through the software, or all combinations of circumstances in which the software will be expected to function.

In truth, though, this behavioral approach is foreign to most real systems in physics. Nearly all physical systems may be understood and analyzed in terms of their component parts. It is far more natural to examine systems in detail, by investigating their internal structure and organization, to watch their internal processes and interrelationships, and to derive from that observation a deep understanding of the "heart" of the system. Here each component may be studied to some degree as an entity unto itself, existing within an environment which is the rest of the system. This is essentially the "divide and conquer" strategy applied to understanding systems, and it has the advantage that the part is usually simpler than the whole. If a particular component is still too complex to permit immediate understanding, it may be itself analyzed as being made up of other smaller pieces, and the process recurses in a natural way.

This concept was recognized by Floyd, Hoare, Dijkstra, and others, beginning about 1969, and an alternative technique to testing is currently in the process of being fashioned by the computing community. This approach is called "program correctness" or "software verification." It is concerned with analyzing a program down to the smallest element, and then synthesizing an understanding of the entire program by composing the behaviors of the individual elements and subsystems. This attention to detail costs a good deal of effort, but it pays off in that the programmer gains a much deeper perception of the program and its behavior, in a way that is complete while being tractable. This deeper examination allows for stronger conclusions to be reached about the software's quality.

As opposed to testing, verification can trace *every* path through a system, and consider *every* possible combination of circumstances, and be certain that nothing has been left out. This is possible because the method relies on mathematical methods of proof to assure the completeness and correctness of every step. What is actually achieved by verification is a mathematical proof that the program being studied satisfies its specification. If the specification is complete and correct, then the program is guaranteed to perform correctly as well.

However, the claims of the benefits of program verification need to be tempered with the realization that substantially what is accomplished may be considered an exercise in redundancy. The proof shows that the specification and the program, two forms of representing the same system, are consistent with each other. But deriving a complete and correct formal specification for a problem from the vague and nuanced words of an English description is a difficult and uncertain process itself. If the formal specification arrived at is not what was truly intended, then the entire proof activity does not accomplish anything of worth. In fact, it may have the negative effect of giving a false sense of certainty to the user's expectations of how the program will perform. It is important, therefore, to remember that what program verification accomplishes is limited in its scope, to proving the consistency of a program with its specification.

But within that scope, program verification becomes more than redundancy when the specification is an abstract, less detailed statement than the program. Usually the specification as given describes only the external behavior of the program. In one sense, the proof maps the external specification down through the structure of the program to the elements that must combine to support each requirement. In another sense, the proof is good engineering, like installing steel reinforcement within a largely concrete structure. The proof spins a single thread through every line of code—but this single thread is far stronger than steel; it has the infinite strength of logical truth. Clearly this greatly increases one's confidence in the finished product. Here is the relevance of the introductory quote from Psalm 51. A system is far stronger if it has internal integrity, rather than simply satisfaction of an external behavioral criterion. The heart of the system must be correct, and to achieve this requires "wisdom" (truth) in the "hidden part."

The theory for creating these proofs of program correctness has been developed and applied to sample programs. It has been found that for even moderately sized programs, the proofs are often long and involved, and full of complex details. This raises the possibility of errors occurring in the proof itself, and brings into question *its* credibility.

This situation naturally calls for automation. Assistance may be provided by a tool which records and maintains the proof as it is constructed step by step, and ensures its soundness. This tool becomes an agent which mechanically verifies the proof's correctness. The Higher Order Logic (HOL) proof assistant is such a mechanical proof checker. It is an interactive theorem-proving environment for higher order logic, built originally at Edinburgh in the 1970's, based on an approach to mechanical theorem proving developed by Robin Milner. It has been used for general theorem proving, hardware verification, and software verification and refinement for a variety of languages. HOL has the central quality that only true theorems may be proved, and is thus secure. It performs only sound logical inferences. A proof is then a properly composed set of instructions on what inferences to make. Each step is thus logically consistent with what was known to be true before. The result of a successful proof is accredited with the status of "theorem," and there is no other way to produce a theorem. The derivation is driven by the human user, who makes use of the facilities of HOL to search and find the proof.

Even greater assistance for program verification may be provided by a tool which writes the proof automatically, either in part or in whole. One kind of mechanical tool that has been built is a *Verification Condition Generator* (VCG). Such a tool analyzes a program and its specification, and based on the structure of the program, constructs a proof of its correctness, modulo a set of lemmas called verification conditions which are left to the programmer to prove. This is a great aid, as it twice reduces the programmer's burden, lessening both the volume of proof and the level of proof. Many details and complexities can be automatically handled by the VCG, and only the essentials left to the programmer. In addition, the verification conditions that remain for him to prove contain no references to programming language phrases, such as assignment statements, loops, or procedures. The verification conditions only describe relationships among the underlying datatypes of the programming language, such as integers, booleans, and lists. All parts of the proof that deal directly with programming language constructs are handled automatically by the VCG. This does not mean that there cannot be depth and difficulty in proving the verification conditions; but the program proving task has been significantly reduced.

Several example Verification Condition Generators have been written by various researchers over the past twenty years. Unfortunately, they have not been enough to encourage a widespread use of program verification techniques. One problem area is the reliability of the VCG itself. The VCG is a program; and just as any other program, it is subject to errors. This is critical, however, because the VCG is the foundation on which all later proof efforts rest. If the VCG is not sound, then even after proving all of the verification conditions it produces, the programmer has no firm assurance that in fact he has proven his original program correct. Just stating a set of rules for proving each construct in a programming language is not enough; there is enough subtlety in the semantics of programming languages to possibly invalidate rules which were arrived at simply by intuition, and this has happened for actual rules that have been proposed in the literature. There is a need for these rules, and the VCGs that incorporate them, to be rigorously proven themselves.

This we have done in this dissertation. We present a verified Verification Condition Generator, which for any input program and specification, produces a list of verification conditions whose truth in fact implies the correctness of the original program with respect to its specification. This verification of the VCG is proven as a theorem, and the proof has been mechanically checked in every detail within HOL, and thus contains no logical errors. The reliability of this VCG is therefore complete.

Program verification holds the promise in theory of enabling the creation of software with qualitatively superior reliability than current techniques. There is the potential to forever eliminate entire categories of errors, protecting against the vast majority of run-time errors. However, program verification has not become widely used in practice, because it is difficult and complex, and requires special training and ability. The techniques and tools that are presented here are still far from being a usable methodology for the everyday verification of general applications. The mathematical sophistication required is high, the proof systems are complex, and the tools are only prototypes. However, the results of this dissertation point the direction to computer support of this difficult process that make it more effective and secure. Another approach than testing is clearly needed. If we are to build larger and deeper structures of software, we need a way to ensure the soundness of our construction, or else, inevitably, the entire edifice will collapse, buried under the weight of its internal inconsistencies and contradictions.