CAMBRIDGE

Cambridge University Press
978-0-521-87903-3 - Practical Formal Software Engineering: Wanting the Software You Get
Bruce Mills
Frontmatter
More information

Practical Formal Software Engineering is a textbook aimed at final-year undergraduate and graduate students, emphasizing formal methods in writing robust code quickly. This book takes an engineering approach to illuminate the creation and verification of large software systems in which theorems and axioms are intuited as the formalism materializes through practice.

Where other textbooks discuss business practices through generic project management techniques or detailed rigid logic systems, this book examines the interaction between code in a physical machine and the logic applied in creating the software. These elements create an informal and rigorous study of logic, algebra, and geometry through software.

Assuming prior experience with C, C++, or Java programming languages, chapters introduce UML, OCL, and Z from scratch. Organized around a theme of the construction of a game engine, extensive worked examples motivate readers to learn the language through the technical side of software science.

Bruce Mills holds a Ph.D. in computer science and mathematics from the University of Western Australia. He has twenty years of experience in the industrial electronics and software fields and as a lecturer in his native country, Wales, and the Middle East. Dr. Mills is the author of *Theoretical Introduction to Programming*. He is currently a software engineer at ABB in Perth, Australia.

© Cambridge University Press www.cambridge.org

CAMBRIDGE

Cambridge University Press
978-0-521-87903-3 - Practical Formal Software Engineering: Wanting the Software You Get
Bruce Mills
Frontmatter
More information

Practical Formal Software Engineering

Wanting the Software You Get

Bruce Mills

ABB, Perth, Australia

CAMBRIDGE UNIVERSITY PRESS

© Cambridge University Press www.cambridge.org

CAMBRIDGE

Cambridge University Press
978-0-521-87903-3 - Practical Formal Software Engineering: Wanting the Software You Get
Bruce Mills
Frontmatter
More information

CAMBRIDGE UNIVERSITY PRESS
Cambridge, New York, Melbourne, Madrid, Cape Town, Singapore, São Paulo, Delhi

Cambridge University Press
32 Avenue of the Americas, New York, NY 10013-2473, USA

www.cambridge.org
Information on this title: www.cambridge.org/9780521879033

© Bruce Mills 2009

This publication is in copyright. Subject to statutory exception
and to the provisions of relevant collective licensing agreements,
no reproduction of any part may take place without the written
permission of Cambridge University Press.

First published 2009

Printed in the United States of America

A catalog record for this publication is available from the British Library.

Library of Congress Cataloging in Publication data

Mills, Bruce, 1962–
Practical formal software engineering : wanting the software you get / Bruce Mills.
p. cm.
Includes index.
ISBN 978-0-521-87903-3 (hardback)
1. Software engineering – Textbooks. I. Title.
QA76.758.M575 2009
005.1–dc22 2008042407

ISBN 978-0-521-87903-3 hardback

Cambridge University Press has no responsibility for the persistence or
accuracy of URLs for external or third-party Internet Web sites referred to in
this publication and does not guarantee that any content on such Web sites is,
or will remain, accurate or appropriate. Information regarding prices, travel
timetables, and other factual information given in this work are correct at
the time of first printing, but Cambridge University Press does not guarantee
the accuracy of such information thereafter.

© Cambridge University Press www.cambridge.org

CAMBRIDGE

Cambridge University Press
978-0-521-87903-3 - Practical Formal Software Engineering: Wanting the Software You Get
Bruce Mills
Frontmatter
More information

This book is dedicated to my wife, Lan Pham Mills.
Remember, you promised to read it.

© Cambridge University Press www.cambridge.org

CAMBRIDGE

Cambridge University Press
978-0-521-87903-3 - Practical Formal Software Engineering: Wanting the Software You Get
Bruce Mills
Frontmatter
More information

Some thoughts on methods by which software can be produced that will satisfy humans. The target human may be the operator, the programmer, or the person who paid for the software. No strong assumption is made on this issue. How can we state what we want from a piece of software, how can we find imprecision and inaccuracy in our statements, how can we make our statements more precise and accurate, and how can we know that the software does what we decided that we want it to do? How do we know that what we said is what we will want?

© Cambridge University Press www.cambridge.org

CAMBRIDGE

Cambridge University Press
978-0-521-87903-3 - Practical Formal Software Engineering: Wanting the Software You Get
Bruce Mills
Frontmatter
More information

Contents

© Cambridge University Press www.cambridge.org

CAMBRIDGE

Cambridge University Press
978-0-521-87903-3 - Practical Formal Software Engineering: Wanting the Software You Get
Bruce Mills
Frontmatter
More information

© Cambridge University Press www.cambridge.org

CAMBRIDGE

Cambridge University Press
978-0-521-87903-3 - Practical Formal Software Engineering: Wanting the Software You Get
Bruce Mills
Frontmatter
More information

© Cambridge University Press www.cambridge.org

CAMBRIDGE

Cambridge University Press
978-0-521-87903-3 - Practical Formal Software Engineering: Wanting the Software You Get
Bruce Mills
Frontmatter
More information

© Cambridge University Press www.cambridge.org

CAMBRIDGE

Cambridge University Press
978-0-521-87903-3 - Practical Formal Software Engineering: Wanting the Software You Get
Bruce Mills
Frontmatter
More information

© Cambridge University Press www.cambridge.org

CAMBRIDGE

Cambridge University Press
978-0-521-87903-3 - Practical Formal Software Engineering: Wanting the Software You Get
Bruce Mills
Frontmatter
More information

Acknowledgments

Lan Pham Mills, my wife, for editorial advice.
Ray Scott Percival for general discussions.
Greg Restall for an e-mail conversation about Curry's paradox.
Peter Warren, Safuat Hamdy, and Anastassios (Tasos) Tsoularis for editorial advice.

© Cambridge University Press www.cambridge.org

CAMBRIDGE

Cambridge University Press
978-0-521-87903-3 - Practical Formal Software Engineering: Wanting the Software You Get
Bruce Mills
Frontmatter
More information

Maxims

1. Make it clean first, then lean.
2. Without documentation, it is not engineering.
3. Without robustness, it is not engineering.
4. Every software engineer understands induction.
5. You are worth what is in your head, not what is on the Web.
6. Natural language is imprecise; formal language is inaccurate.
7. Be concrete.
8. Write clearly from the beginning; make everything explicit.
9. Abstraction means the same thing as modularity.

© Cambridge University Press www.cambridge.org

CAMBRIDGE

Cambridge University Press
978-0-521-87903-3 - Practical Formal Software Engineering: Wanting the Software You Get
Bruce Mills
Frontmatter
More information

Preface

This paragraph is so far short of explaining what this book is about that it is a grotesque, but at least it tells you that it is not a book on cookery or poetry.

Practical means using the material at hand. Formal means clear, explicit rules. Software is anything that can be manipulated exactly, including algebraic terms and mechanical puzzles. Engineering is obtaining an approximate result from approximate material. Science is useful to the engineer, but engineering is not science: science obtains exact results but requires exact material. Engineering is not a commercial subject, even though it has commercial implications. Mathematics is useful to the software engineer, but software engineering is not mathematics. Mathematics is not introspective enough. This book is a discussion of practical formal software engineering.

A preface is an informal chat the speaker has with the few who arrive early for a seminar. Assuming goodwill from the reader, this preface says, without elaboration or apology, where I am coming from: *software engineering is engineering*. An electronic engineering book is filled with circuit diagrams. This book describes software engineering in the same mode. Version control, the product life cycle, and project management are all important to engineering workshops, but they are not engineering.[1,2]

For the details, see the bulk of the book.

This sentence says what the book is about.

I take an explicit para-consistent stance on mathematics and other formal studies: formal studies are empirical sciences, their justification is empirical, and no proof ever gives certainty, contradiction is not avoidable in serious work, the principles of the excluded middle and explosion are not always useful, and paradox arises because of unacknowledged *material* conjecture about metalogic.[3]

Some experts in formal areas will respond, *Joe Blogs solved that problem*. But, in my reading in preparation for writing this book, I found, in the words of Mr. Henry Albert Bivvens, *Nay, nay, not so, but far otherwise*. Debate continues on whether the reals are countable, infinite sets exist, the principle of explosion is true, set theory is consistent, the axiom of choice is correct, or any nontrivial logic system can be consistent. For practical reasons, these concerns cannot be ignored when engineering software.[4]

It is a myth that mathematics is static or monotonic, a myth maintained by selective amnesia.

Constructive mathematics, with which this book is aligned, is sometimes said to make bizarre, difficult-to-believe statements. In fact, the constructive

© Cambridge University Press www.cambridge.org

CAMBRIDGE

Cambridge University Press
978-0-521-87903-3 - Practical Formal Software Engineering: Wanting the Software You Get
Bruce Mills
Frontmatter
More information

Not that the classical results are wrong, but they are not provably right.

approach says less, not more, than the existential approach. Its main impact is to promote caution, to avoid going beyond what the practical facts actually require. Once this point is made clear, many logical objections evaporate.

But a practical book on software theory should not contain a discussion of the esoteric foundations any more than a book on practical circuit theory should discuss the conflict between Maxwellian and Newtonian mechanics. This book is about the use of, not the justification for, a para-consistent attitude. The real question remains: is this a useful attitude for a software engineer to have in practice? During two decades of writing software, I have found it to be so.

The software

Software is not *what you find on a desktop computer*: spreadsheets and games. The natural limit of software is the limit of software techniques. This limit includes all that *might* be found on a desktop computer: any program in any existing language, including C, Haskell, Scheme, Prolog, Java, Assembler, and PERL. Beyond even this, it includes all the precise notations of formal logic and mathematics, as well as physics and chemistry. It includes the way tiles cover a wall. Software is precisely defined operations of precisely defined mechanisms: *finitary* in the sense of Hilbert.[5]

A doctor must have a good bedside manner, but a broken leg is not repaired with sweet talk.

What a program is intended to mean is vital to the social purpose of engineering. But to achieve the social purpose, the software must be built. To build and debug software, the engineer must see the meaningless mechanics. The rules of software *can be followed without knowing the meaning*. The raw material of software engineering is formal manipulation. Programs are built from this as electronic devices are built from chips.[6]

The computer does what you say, not what you mean.

Imagine a tree. You experience clarity; you can say whether it is bare, or tall, or round. You can imagine it over a piece of paper and pick up a pencil. But as you move to trace it, it evaporates. Your experience of clarity is an illusion.

Similarly, programming begins with an idea that seems clear until an attempt is made to write it down exactly. It is not a language problem. In your mind are some property descriptions, incomplete, inconsistent, and incorrect. Going from idea to English to OCL to Java is formalization: developing an imprecise idea into a precise idea. The original idea might not be satisfied by any piece of code at all. The process of writing code includes changing your mind about what you want, to make it possible to write.

Game software is a good context for software engineering. Games are generic. A game constructs a virtual universe: a tic-tac-toe board or the world of *Star Trek*. Just as every program defines a language, every program defines a world. The player views part of the world, thinks, and then acts. The universe might pause

© Cambridge University Press www.cambridge.org

CAMBRIDGE

Cambridge University Press
978-0-521-87903-3 - Practical Formal Software Engineering: Wanting the Software You Get
Bruce Mills
Frontmatter
More information

for this or continue regardless. Games do this explicitly. I claim this is a good way for humans to think about all software.

The logic

Logic is the *science* of correct reasoning using text and diagrams. There are no self-evident principles. The correctness of a logic of software is a material conjecture, as is the correctness of a mechanics of billiard balls. Logics are mechanisms, as are Turing machines. The rules of reasoning can be followed without knowing the meaning. This prevents subjective interpretation from invalidating a conclusion. The statements in a logic are not inherently true, false, or meaningful.

Even if humans could compute the noncomputable, the method could not be recorded; it would be revelation, not reasoning.

Each program is a logic, so a logic of programs is recursive: software of software. Compilers and debuggers are everyday examples. There is no hierarchy of language and metalanguage. English can describe itself, and so can C. The limitations this implies are natural and fundamental. The limitations are generic to reasoning; changing the language does not evade the problem, nor can humans compute anything provably impossible for a computer.

A program is a finite expression. Only the finite exists in software. The infinite is coded as finite logic, as a potential, not an actual, infinity. Infinite axiom schemes are finite second-order axioms. Software engineering builds finite expressions with desired algebraic behavior.

A software logic that is complete and correct would help, but twentieth-century research says that no practical system can be both. Where there is conflict, mathematics tries to be correct and engineering, complete. Thus, bugs, known as paradoxes, exist in software theory, to be dealt with as they arise. But, a bug in a logic system does not justify discarding recursive logic, any more than a bug in a C program justifies discarding C or even the C program itself.

A conservative extension to generalized functions exists.

The equation $S = \{x | x \notin x\}$ has no solutions, because $S \in S \equiv S \notin S$. But this is no more mysterious than there being no real valued function that satisfies the definition of the impulse function. However, it is not always possible to know in advance when such a problem will occur.

Insisting on syntactic limitations makes the logical development cumbersome and does not remove the problem but only disguises it. To avoid paradox, orthodox mathematics rejects self-reference, so it is inadequate to describe software. In practice, attempts to use correct software logic lead to simulation of complete software logic, which only disguises the paradox. It is like having a machine that cannot crash running a simulation of one that can. The distinction is a useless technicality.

All logical difficulties arise from *material* self-reference, in which the logic is *conjectured* to refer to its own behavior. This recursion generates a logical

© Cambridge University Press www.cambridge.org

CAMBRIDGE

Cambridge University Press
978-0-521-87903-3 - Practical Formal Software Engineering: Wanting the Software You Get
Bruce Mills
Frontmatter
More information

Removing recursion is shooting the messenger.

equation that might have no solution. But there is no mind-bending paradox. The halting problem is based on a paradox. It would be esoteric if not that it is embedded in seemingly harmless problems, which are thus unsolvable: finding the set of integer solutions of multivariate polynomials over the integers, for example. Self-referential logic does not cause this problem; it allows it to be studied.

The absolute truth

Mathematics as absolute truth is historically inaccurate. Mathematics changes over time, not just by addition, but by revision and retraction. Ancient concepts still used today exist in so mutated a form as to be no more (or less) recognizable to the ancients than modern physics. Old proofs become fallacies; results still used are accepted only under the burden of a different method of definition and demonstration. It is circular to suggest that those things *truly* proved, not just believed, remain accepted today. That which is not accepted today is declared likewise to have never been properly proved.

Infinity is a common changing theme. The ancient Greeks, on the whole, rejected infinity. Where we prove today with limits, they used exhaustion; where we use real numbers, they used pairs of line segments (a similar principle is often used in modern algebra). In each case to avoid infinity. But, the modern concepts of limits and real numbers come, largely, from the nineteenth century: not so long ago.

The victory of limits over exhaustion is not one of truth, but one of utility; limits are easier to manipulate. Philosophical correctness is outgunned by convenience. This was especially true in the seventeenth century, when infinite sums and infinitely small quantities were used to increase the power of algebra. The ideas led to many contradictions and many objections. Eventually, several concepts – the limit, the infinitesimal as a function, and so on – were created, and the fallacy of the raw infinitesimal was laid to rest with a stake through its heart. But in the mid-twentieth century, the raw infinitesimal clawed its way back to the surface with a proof that if the raw infinitesimal was a fallacy then so was the limit.

The raw infinite was rejected for centuries, but it emerged again in the late nineteenth century, with the theory of transfinites. After initial rejection, the theory was glorified as putting the infinite on a firm foundation. But in the twentieth century, it was shown that the size of transfinite sets is relative to the logic used. Classically uncountable, reals are countable in other logics.

Far from a pathological rarity, this is normal for the foundations of mathematics. The problem in writing this section was not a lack of examples, but of

© Cambridge University Press www.cambridge.org

CAMBRIDGE

Cambridge University Press
978-0-521-87903-3 - Practical Formal Software Engineering: Wanting the Software You Get
Bruce Mills
Frontmatter
More information

selecting a few details from a vast sea of material, most of which is not even hinted at here. Changes in the *officially correct* continue, unabated, through the twentieth century, to the time of this writing.

The algebra

Lists are logically prior to sets. To speak about sets, we need language, and that language is a sequence, or list, of symbols. Finite sets are an equivalence class on lists. Infinite sets are a finite logic.

Algebra began as the algebra of numbers. Principles such as the commutativity of addition, `x+y==y+x`, are compact expressions describing an infinite number of pure substring replacements. Whenever the pattern `x+y`, such as `1+1`, `2+3`, or `1562+98`, appears in a finite source string such as `5*(2+3)`, it can be replaced by the string `3+2` constructed according to the rule, to generate a target string `5*(3+2)`. In so doing, there is a local matching `{x==2,y==3}`. Both the source and the target are said to be states of the *host* string.

Rules also apply to rules. Combining `a+b==b+a` with `a*b=b*a` produces the rule `(x+y)*z==z*(y+x)`. The action of several rules can be summarized as a rule. For any initial set of rules, there is an abstract set of all the rules generated by those initial rules.

Leibniz observed that *all* precise reasoning is like this.

Software extends this to one-way rules, `x+y`→`y+x`, in which the replacement must work from the left-hand pattern to the right-hand pattern. This gives a concept of direction, of working from the available to the desired.

The simplest case is that each rule is a pair of literal strings. A rule states that an instance of exactly the left-hand string may be replaced by exactly the right-hand string. This turns out to be a general principle of computation. That is, any set of rules using patterns is generated by some set of literal rules.

Any one specific application of a pattern rule replaces one literal substring with another literal substring, no matter how the decision to do so was made. Part of the host string, possibly beyond the piece that materially changed, was examined to validate this replacement. But there is a limit beyond which nothing affects or is affected by the replacement. This larger string is the scope of the specific application of the rule. The scope might be the whole host string, but because each host string is finite, a finite scope always exists.

The natural scope of an application of a rule is the smallest string in which the replacement is context free. Relative contextual freedom is when extra information is known about the outside string (such as no use of a certain construct), corresponding to information about the programming paradigm.

© Cambridge University Press www.cambridge.org

CAMBRIDGE

Cambridge University Press
978-0-521-87903-3 - Practical Formal Software Engineering: Wanting the Software You Get
Bruce Mills
Frontmatter
More information

Replacement reasoning does work in software. However, replacements tried in a naive analogue to traditional mathematics are not always correct. The replacement is affected by other parts of the string: the effect is not as local as expected. This is an error in the judgment of the programmer, not in the *principle* of algebra.

In modifying a function in a C program, if there are global variables then a code change might have a complex effect on the rest of the program. But with local variables only, the effect will be limited to the function itself. The entire code inside the function can be replaced without worrying about any context beyond the function. This is modularity, but the important property is localization in the *expression*.

Without knowledge about which parts can be modified locally, it is impossible in practice to write code. There must be some limit to where to look. The key to programming algebra is to know which pieces of code can be changed without looking at their context.

The bigger this string, the more errors are possible.

The methodology

Knowing the software methodology can be important. If the design rule "never use a global variable called Fred" is being used in the rest of the code, then without looking at the context, changes to the internals of the one function that does use Fred can be made. The algebraic rules depend on how the program was written. Turning this around, designing the program to support a useful class of algebraic rules can lead to improved code.

Some say that declarative language is best for this, and it seems so in practice, but it is due to social factors. Spaghetti C code can be given declarative state transformation semantics, but it does not untangle the logic; however, cleanly written C code can be proved. Declarative-language designers had formal training, and their languages pressure the programmer to write clean code. Most declarative programmers had similar training and responded easily to the pressure.

Traditional declarative programs were clear and often provable. But non-traditional programmers introduced to using monads in a syntax that looks like Fortran, and encouraged to think in an imperative style, produce spaghetti: it does not just look like traditional imperative code, it *behaves* like it. Traditional mathematics has a style in which reasoning proceeds rapidly with few errors. It is the unwritten details of how to say things that make the difference.

A human works on a thousand-character expression as a Turing machine works on its tape: making local changes and moving on. All rules of good software engineering come down to locality. Declarative programming is not

© Cambridge University Press www.cambridge.org

CAMBRIDGE

Cambridge University Press
978-0-521-87903-3 - Practical Formal Software Engineering: Wanting the Software You Get
Bruce Mills
Frontmatter
More information

the issue, object programming is not the issue, and structured programming is not the issue. When locality is strongly violated, the trouble starts.

The algebra of imperative operators is different from that of the equivalent declarative state–change operators. Orthodox matrix multiplication is associative, but when the time taken is noted, the operation is no longer associative. These are the problems of using mathematics naively in software. The algebra of the action is different from the algebra of the value of expressions. Most mathematics uses the value, and so a common mistake is to transliterate the wrong algebra into the software.

Real numbers are impossible as software. The algebra of reals is different from the algebra of floats. Transliterating real algebra results into a program is a mistake. The correct approach is to map compound expressions to compound expressions. The *algebra* of the reals *is* possible in software, but it requires sophisticated code and is often not justified in practice.

Algebra is the manipulation of some expressions to create others with desired properties. This also describes programming. The algebraic tool is substring replacement. Each replacement has a context (part of the host string); nothing outside this affects it. Every piece of code admits algebraic replacement, within context. But the context might be the whole program. The larger the context, the more likely are errors. Programming paradigms can be described by the algebra they admit. Designing the code to a style that admits useful, local, replacement rules is a good way to improve the quality of code.

Half a proof

Still, half a sixpence is better than half a penny
is better than half a farthing is better than none.

In a chain of deductions, if one deduction is not valid, then the validity of the chain is destroyed. But is this principle of validity important?

You cannot become sure of *anything*, material or abstract, if you are not already sure of *something*. Aristotle showed that a deductive proof from nothing is an infinite regression. Lewis Carroll showed that *modus ponus* is an infinite regression. Bacon said that an induction never makes a conclusion *certain*. Popper said that deducing a conjecture false is impossible because there may be an error in the refutation method. The no-free-lunch theorem states that for any two methods of reasoning there is *some* conceivable universe in which the one is no better than the other, including random guessing and even picking the worst model each time.[7]

After proving the program, there is the proof of the compiler, and the operating system, and the hardware, and the system that manufactured the hardware, and

© Cambridge University Press www.cambridge.org

CAMBRIDGE

Cambridge University Press
978-0-521-87903-3 - Practical Formal Software Engineering: Wanting the Software You Get
Bruce Mills
Frontmatter
More information

the system that proved that, and so on. But exactly the same problem occurs with any formal proof in all of mathematics and logic. In practice, the reasoner eventually stops and says, *Enough.*[8]

Are formal studies at least a precise game played like chess? Perhaps they could be, but mathematics is not played this way. Machine checking of journal proofs shows that almost every proof in practice has errors and omissions. Mathematicians claim that these holes can be filled, but the proofs are still not *strictly* valid. It is *fallible intuition* based on instinct and experience that leads a proof to be accepted by the mathematical community. Paradoxically, if only *strictly* correct proofs were useful, then mathematics would be useless. In practice, mathematics is robust. A partial proof from a good mathematician is strong evidence that *something like that is true.*[9]

At least, tests have led to its rejection as *truth*.

Classical mechanics is false. But it is still used to build cars, planes, and bridges. It is used because, in practice, it returns results that are close enough with reasonable effort. A false theory that is robust is much more useful than a true theory that breaks when the information is not exact.

Robust means good results even when the axioms are false.

A typical complexity proof shows only that the limit to *infinity* of the ratio of a formula to the behavior is finite. Ask whether it is hot at the beach and be told that it is 20° on Saturn. This is useful only because a serious *attempt to prove* such a limit gives informal information about the *finite* behavior. But this is a *material* fact about the *generation of the proof*, not a logical fact about the proof itself.

Most logicians are prepared to say natural induction is true.

Many formal theories make material conjectures. The Peano theory of the naturals includes mathematical induction. But, as applied to the other Peano axioms, this is a material assertion about an infinite number of proofs. It cannot just be demanded to be true; it might be false.[10]

The correctness of nontrivial logical formalisms is always hostage to material conjectures about proof in *some* logical formalism. But this does not mean that attempted formalism is useless. The proof of an electronic device depends on the assumptions made in its design. In practice, proofs are never certain, but the process of trying to make formal proofs is correlated with better code. This is a material fact about computers, as is electromagnetics.

In the field

Formal methods mean explicit theory, not certain proof. At best, they mean proof given assumptions. But the assumptions are explicit, so it is clear where to *improve* the rigor. The assumptions might be wrong, but *some theory* is better than none. The theory can always be improved later. Software engineering constructs theory with code. It is a scaffolding approach. To build a pyramid, build a big sand ramp at the same time.

© Cambridge University Press www.cambridge.org

CAMBRIDGE

Cambridge University Press
978-0-521-87903-3 - Practical Formal Software Engineering: Wanting the Software You Get
Bruce Mills
Frontmatter
More information

Nothing is ever certain.

One criticism points out that formal software engineering does not give certainty but fails to point out that no other engineering discipline does either. Electronic engineering uses formal proofs of circuit properties together with rules of thumb and testing to reduce the error rate. Likewise software engineering. Does partial formal verification reduce the error rate in practice? Yes, up to a point. How much effort is justified? The precise cutoff depends on the social context of the application, but it is never zero or infinity. Also, formal methods can be used to gain a benefit, without any proof at all, by precisely describing the intended behavior. In fact, what you *really* want is not provable: you want to know that the specification satisfies your *desires*.[11]

Just like structured or object-oriented code and documentation, formal methods are not all-or-nothing. Half structured is better than total spaghetti. But formal methods in software have been criticized for an inability to start with an arbitrary piece of code and verify it. This observation is true but misleading. When a program is built from the beginning in a formal manner, it is fairly easy to continue. Trying to prove spaghetti code is like trying to modularize it or document it. No protagonist of object-oriented programming would suggest that the power of OOP includes the ability to easily neaten spaghetti code.

Engineering

Leslie Lamport said that to make software, an engineering discipline needs an engineering theory of mathematical proof. But such a theory must involve *partial* proofs and will be justified by the working technology built using it and will not be liked by mathematicians, any more than physicists like engineering circuit theory. If a system is set up just so, so that an exact proof is possible, then it is mathematics, not engineering. But we should not hope to entirely prove software systems in practice if only because humans will always push the technology beyond their ability to prove, even if that ability is improving.

Part of software engineering is handling large proofs: bookkeeping, modularity, strategy. Practical software expressions are thousands of times bigger than those commonly used by mathematicians. But this still involves, in software terms, empirical matters, parts that are not proved and parts that cannot be proved.

Electronics engineering is not about a particular brand of transistor, but about the principle of how transistors work in general; likewise software engineering is not about specific code elements in a given language; it is about the general issues that are true of many languages – not how does a while-loop work in C, but how does it work in general, and why is it the same thing as a Haskell recursion. Such things as a VLSI chip in electronics mean code libraries in software. A technology – CMOS, TTL, and so on – is roughly analogous to

© Cambridge University Press www.cambridge.org

CAMBRIDGE

Cambridge University Press
978-0-521-87903-3 - Practical Formal Software Engineering: Wanting the Software You Get
Bruce Mills
Frontmatter
More information

a software language. Several technologies are discussed so that intuition can be gained about general principles, rather than slavish adherence to a single paradigm promoted.

Project management

Stuff the philosophy, how do we write the software?

But the agnostic is hated more than the atheist.

Today, software engineering is split. Players have competing motivations and backgrounds. There will be those who object to one element or another of this book, feeling it is false or misguided. I hope that they will still find something here. Most of the material is neutral; it presents practical mental tools for those for whom the primary goal is to build software that works. It does not *deny* other mindsets; it simply does not *require* them.

Project management ceases to be engineering when the code cannot be seen. But some organizational matters directly affect the programmer. While they are not a large part of this book, they should be mentioned.

Specification, design, implementation, testing, debugging, documentation, and maintenance are activities, not stages. To a degree, each occurs at all stages of the life of the code. The fact of "leaving the design stage" should be observed, not legislated. Documentation, including specification, is to be done at all times. The code is a side effect of the documentation. What the code is to do is considered, and then the code is shown and demonstrated to be correct. This is the theorem-proof style of mathematics. The code is the theorems, and most of a book is either proof or discussion.

Every program is derived from the hello-world program. Speculative prototyping is used to clarify the specification, which might be modified as a result. Code that has been exercised enough is classified as tentatively completed, to be used in the final version. Top-down, bottom-up, middle-out: all are parts of design. The key is to design the proof and the program at the same time and modify whichever part should be modified. Versions should be kept to a minimum. Rather, small, stable modules should be spawned, if there are multiple issues of exactly how code should be written. Stepwise refinement (sequential prototyping) is the rule.

The final program, as big as it may be, should be expressed as a short, simple piece of code, using the utility modules that have been designed in the process.

Notes

1. David Parnas, author of the classic "On Criteria to Be Used in Decomposing Systems into Modules" [1972], has argued in "Software Engineering Programmes Are Not Computer Science Programmes" [1997] that software engineering is engineering.

© Cambridge University Press www.cambridge.org

CAMBRIDGE

Cambridge University Press
978-0-521-87903-3 - Practical Formal Software Engineering: Wanting the Software You Get
Bruce Mills
Frontmatter
More information

2. This is not an algorithms book. An electronic equivalent of that is a book of circuits; a mathematical equivalent is a book of standard sums and integrals.
3. Every program can be shortened by at least one instruction and contains at least one bug. Thus, every program can be reduced to a single instruction that does not work.
4. In the early nineteenth century, Augustus De Morgan complained bitterly about the logical absurdity of negative numbers; in the early twentieth century Skolem said that set theory was an absurd thing to found mathematics on. Godel proved you could not prove arithmetic, but his PhD student proved you can. In the 1960s, Robinson turned the logical absurdity of infinitesimals into an accepted concept. Popper has shown that science has virtually no formal foundation. In the early twenty-first century, debate rages about how to respond to Curry's paradox. Whether any nontrivial logic can be consistent is questioned, as is whether the real numbers are countable. Are the axioms of choice and continuum true, or are there different versions of set theory, as there are different versions of geometry?
5. Most programs are also character strings – those that are not are diagrams or mechanical devices. All are some network of elements related in discrete ways. The action of the machine is replacement of substrings or motion of parts. Strings and diagrams correspond to human vision and hearing. We consider them precise because they are how we observe the world. A dog, however, might program with smells.
6. There is a big overlap between electronic and software engineering, especially with logic gates and flip-flops, registers, arithmetic units, logic units, central processor units, and communication blocks.
7. *Modus ponus* is deducing B from A and A-implies-B. The problem is that this rule is an instance of itself; so strictly, if your audience does not already believe it, you cannot conjure it into existence deductively. Lewis Carroll, *What the Tortoise Said to Achilles*. Mind, Vol. 4, No. 14, April (1895),
8. Albert Einstein said as far as the laws of mathematics refer to reality, they are not certain, and as far as they are certain, they do not refer to reality. (In J.R. Newman (ed.), *The World of Mathematics*, New York, Simon and Schuster, 1956, Bertrand Russell said (in reference to the relation of mathematical topics to anything nonmathematical) that mathematics is the subject where we never know what we are talking about nor whether what we are saying is true).
9. Similarly, an experienced software engineer can tell that an approach is *likely* to go wrong in practice, even though it is technically correct.
10. This is more traditionally expressed as a lack of certainty that Peano arithmetic is consistent.
11. *Seven Myths of Formal Methods*. A. Hall, Software IEEE, Vol. 7, Issue 5, September 1990, pp. 11–19. (Hall promotes the Z specification language and emphasizes specification as the aspect of formal methods that is usually the most important. *Seven More Myths of Formal Methods: Dispelling Industrial Prejudices*. J.P. Bowen, and M.G. Hinchey. http://www.jpbowen.com/pub/fmep4.pdf.) A longer version of the Bowen paper is available at Oxford University, Computing Laboratory, Programming Research Group, Technical Report TR-7-94. http://web.comlab.ox.ac.uk/oucl/publications/tr/tr-7-94.html.

© Cambridge University Press www.cambridge.org

CAMBRIDGE

Cambridge University Press
978-0-521-87903-3 - Practical Formal Software Engineering: Wanting the Software You Get
Bruce Mills
Frontmatter
More information

Further Reading

The following is a cursory list of books, mostly classics, related to the material in this book. Without trying to point out any specific connection, I recommend them all as good background reading. Readers are invited to arrive at their own conclusions.

The World of Origami, by Isao Honda
Origami Omnibus, by Kunihiko Kasahara
Tilings and Patterns, by Branko Grünbaum and Geoffrey Shephard
Non Standard Analysis, by Abraham Robinson
A Discipline of Programming, by Edsger Dijkstra
Polyominoes, by Solomon Golomb
Algorithms + Data Structures = Programs, by Niklaus Wirth
Fundamental Algorithms, by Donald Knuth
The TeXbook, by Donald Knuth
The Fractal Geometry of Nature, by Benoit Mandelbrot
The Elements of Geometry, by Euclid
Foundations of Geometry, by David Hilbert
Mathematics and Plausible Reasoning, by George Polya
Computers and Intractability, by Michael Garey and David Johnson
Foundations of Logic Programming, by John Lloyd
Lambda-Calculus, Combinators, and Functional Programming, by Gyorgy Revesz
A Concise Introduction to Logic, by Patrick Hurley
Logic and Design, by Krome Barratt
Computation: Finite and Infinite Machines, by Marvin Minsky
The Implementation of Functional Programming Languages, by Simon Peyton-Jones
A New Kind of Science, by Stephen Wolfram
Forever Undecided: A Puzzle Guide to Godel, by Raymond Smullyan
Science and Information Theory, by Leon Brillouin

© Cambridge University Press www.cambridge.org

CAMBRIDGE

Cambridge University Press
978-0-521-87903-3 - Practical Formal Software Engineering: Wanting the Software You Get
Bruce Mills
Frontmatter
More information

The Knot Book, by Colin Adams
Electronic Analogue and Hubrid Computers, by Granino Korn and Theresa Korn
Structured Programming, by Richard Linger, Harlan Mills, and Bernard Witt
Martin Gardner (popular mathematics)
Henry Dudeney (classical puzzleist)
Sam Loyd (classical puzzleist)

© Cambridge University Press www.cambridge.org

CAMBRIDGE

Cambridge University Press
978-0-521-87903-3 - Practical Formal Software Engineering: Wanting the Software You Get
Bruce Mills
Frontmatter
More information

To the Teacher

The chapter dependencies are the same as their order in the book. The book is loosely based on the writing of a software game.

The first part of the book is background material. Students with a sound logic training may skip this section. But the teacher may find, because the background is constructive, that it is easier to teach students who have no logic background than students who have learned classical logic only and must unlearn some of it first. The students should at least *read through* this material. The first chapter, especially, might seem elementary, but it has a strong point to make about the foundations of software. On the other hand, if the first section is done in all detail, then it could be the entire course – especially if combined with the chapters on plain text and state transformation.

The second part is about languages. It does introduce UML, OCL, and Z, which are specification languages, but it does not do so using the orthodox cultural background. Rather, the intention is to demonstrate that it is the logic, not the language, that matters. Each chapter introduces a point about the logic of software, within the context of describing the selected language. A specification language is not different in kind from a programming language, other than in being easier to write in and, as a result, harder or impossible to compile. The part concludes with a chapter on logic, to consolidate this principle, and a chapter on Java to demonstrate the practice.

Of course, there are many issues that make the practice of programming very different from the practice of, say, mathematics or logic. Two fundamental issues are the requirement for a decision process and that the expressions involved might have millions of characters, whereas in mathematics and logic there are usually tens of characters.

The third part is the second half of the book. It includes several chapters that are just exercises. Many of the exercises for the second part can be found here, interpreting each exercise for each language. This is why I separated them out. All the exercises could be done in each language, and it seemed a pity to waste them (so to speak) by putting them in just one chapter. Each of these chapters

© Cambridge University Press www.cambridge.org

CAMBRIDGE

Cambridge University Press
978-0-521-87903-3 - Practical Formal Software Engineering: Wanting the Software You Get
Bruce Mills
Frontmatter
More information

has a software focus, solid modeling, command line, graphics, security, and natural language. There is much less dependency between these chapters. The teacher may choose whichever seems appropriate. There are enough suggestions for practical projects that advanced students could spend the whole course here.

Although outcomes are given for each chapter, the real claim is that if a student reads and understands the material, then his or her ability to write software will improve in practice. The detailed outcomes tend to obscure this overall goal.

© Cambridge University Press www.cambridge.org

CAMBRIDGE

Cambridge University Press
978-0-521-87903-3 - Practical Formal Software Engineering: Wanting the Software You Get
Bruce Mills
Frontmatter
More information

To the Student

Although other references are given and may be more secure, at the time of this writing there are some significant Web resources: Wikipedia (Open Web-encyclopedia), Plato (The Stanford Encyclopedia of Philosophy), and Mathworld (hosted by Wolfram Research),

`http://en.wikipedia.org,`
`http://plato.stanford.edu, and`
`http://mathworld.wolfram.com,`

which have been fairly stable and contain nontrivial, valid information for many of the topics in this book, which can be obtained by relatively small amounts of effort with obvious keywords.

Any student with a reasonable combination of determination and ability should be able to learn a lot from these resources. Remember that it is what is in your head that matters, not what is on the Web.

Software is like cooking. If you watch a person making a lemon meringue pie, you do not learn much (unless you are already a good cook). And a photograph of a pie does not provide suitable information on how to make it. When a piece of code is given, it is not usually the piece of code that is important, but the process of obtaining it, the explanation, and the way in which it is transformed and expanded into a new piece of code. You learn software by doing. Software theory is intended to help you do software. If you find that the theory is *yet another thing you have to learn*, then you have missed the point. Software plus theory is *easier* to learn than just software. If you do not see this, then you have not absorbed the theory. Go back and have another look.

If you have 10 hours to spend on learning programming, spend 1 hour a day, or 1 hour a week, not 10 hours in 1 day. Your brain needs time to absorb the material. Your brain will process it, even when you are not thinking about it. When the pop-up toaster in your head produces a new idea, even if it is in the middle of a bath, do not throw it away, but cogitate on it for a while.

Above all, always ask the question, Is it true?, Is it *really* true? Not just as something that an authority said to you, nor just something that fits with your *prior* experience, try it out, see how it works on its own terms, and after that ask if you can use it to write software. Regardless of how it might sound.

In the end you must be able to say, *The concept is valid, no matter what the source.*

© Cambridge University Press www.cambridge.org