Preface

As a programmer working for Logica UK in London in the mid-1980's, I became a passionate advocate of formal methods. Extrapolating from small successes with VDM and JSP, I was sure that widespread use of formal methods would bring an end to the software crisis.

One approach especially intrigued me. John Guttag and Jim Horning had developed a language, called Larch, which was amenable to a mechanical analysis. In a paper they'd written a few years earlier [23], and which is still not as widely known as it deserves to be, they showed how questions about a design might be answered automatically. In other words, we would have real software "blueprints"—a way to analyze the essence of the design before committing to code. I went to pursue my PhD with John at MIT, and have been a researcher ever since.

As a researcher though, I soon discovered that formal methods were not the silver bullet I'd hoped they would be. Formal models were hard to construct, and specifying every detail of a system was too hard. Theorem proving, the kind of analysis that Larch relied on, could not be fully automated. Even now, after 20 more years of research, it still requires the careful guidance of a mathematical guru. In my doctoral work, therefore, I took a more conservative route, and worked on automatic detection of bugs in code. But I kept an interest in the more ambitious world of formal methods and design analysis, and hoped one day to return to it.

In 1992, I visited Carnegie Mellon University. By then, I'd become enamored, like many in the formal methods community, with the Z language. The inventors of Z had dispensed with many of the complexities of earlier languages, and based their language on the simplest notions of set theory. And yet Z was even less analyzable than Larch; the only tool in widespread use was a pretty printer and type checker.

On that visit, Ken McMillan showed me his SMV model checker: a tool that could check a state machine of a billion states in seconds, without any aid from the user whatsoever. I was awestruck.

With the invention of model checking, the reputation of formal methods changed almost overnight. The word "verification" became fashionable again, and the adoption of model-checking tools by chip manufac-

turers showed that engineers really could write formal models, and, if the benefit was great enough, would do it of their own accord.

But the languages of model checkers were not suitable for software. They were designed for handling the complexity that arises when a collection of simple state machines interacts concurrently. In software design, complexity arises even in a single machine, from the complex structure of its state. Model checkers can't handle this structure—not even the indirection that is the essence of all software design.

So I began to wonder: could the power of model checking be brought to a language like Z? Here were two cultures, an ocean apart: the gritty automation of SMV, reflecting the steel mills and smokestacks of Pittsburgh, the town of its invention, and the elegance and simplicity of Z, reflecting the beautiful quads of Oxford.

This book is the result of a 10-year effort to bridge this gap, to develop a language that captures the essence of software abstractions simply and succinctly, with an analysis that is fully automatic, and can expose the subtlest of flaws.

The language, Alloy, is deeply rooted in Z. Like Z, it describes all structures (in space and time) with a minimal toolkit of mathematical notions, but its toolkit is even smaller and simpler than Z's. Alloy was also strongly influenced by object modeling notations (such as those of OMT and Syntropy). Like them, it makes it easy to classify objects, and associate properties with objects according to the classification. Alloy supports "navigation expressions," which are now a mainstay of object modeling, with a syntax that is particularly simple and uniform.

The analysis, embodied in the Alloy Analyzer, actually bears little resemblance to model checking, its original inspiration. Instead, it relies on recent advances in SAT (boolean satisfiability) technology. The Alloy Analyzer translates constraints to be solved from Alloy into boolean constraints, which are fed to an off-the-shelf SAT solver. As solvers get faster, so Alloy's analysis gets faster and scales to larger problems. Using the best solvers of today, the analyzer can examine spaces that are several hundred bits wide (that is, of 10^{60} cases or more). Hardware advances must also get some of the credit. Even had this technology been available 10 years ago, an analysis that takes only seconds on today's machines would have taken an hour back then. (Incidentally, Alloy was by no means the first application of SAT to this kind of problem. SAT had been used for analyzing railway control systems [68], for checking hardware [69], and for planning [45, 17]. Since its adoption in Alloy [33], it has been incorporated into model checkers too [5].)

The experience of exploring a software model with an automatic analyzer is at once thrilling and humiliating. Most modelers have had the benefit of review by colleagues; it's a sure way to find flaws and catch omissions. Few modelers, however, have had the experience of subjecting their models to continual, automatic review. Building a model incrementally with an analyzer, simulating and checking as you go along, is a very different experience from using pencil and paper alone. The first reaction tends to be amazement: modeling is much more fun when you get instant, visual feedback. When you simulate a partial model, you see examples immediately that suggest new constraints to be added.

Then the sense of humiliation sets in, as you discover that there's almost nothing you can do right. What you write down doesn't mean exactly what you think it means. And when it does, it doesn't have the consequences you expected. Automatic analysis tools are far more ruthless than human reviewers. I now cringe at the thought of all the models I wrote (and even published) that were never analyzed, as I know how error-ridden they must be. Slowly but surely the tool teaches you to make fewer and fewer errors. Your sense of confidence in your modeling ability (and in your models!) grows.

You can use analysis to make models not only more correct but also more succinct and more elegant. When you want to rework a constraint in the model, you can ask the analyzer to check that the new and old constraint have the same meaning. This is like using unit tests to check refactoring in code, except that the analyzer typically checks billions of cases, and there are no test suites to write.

I sometimes call my approach "lightweight formal methods" [39], because it tries to obtain the benefits of traditional formal methods at lower cost, and without requiring a big initial investment. Models are developed incrementally, driven by the modeler's perception of which aspects of the software matter most, and of where the greatest risks lie, and automated tools are exploited to find flaws as early as possible.

But at the same time as I have argued against some of the assumptions of traditional formal methods, my experience in the last decade—teaching software engineering to students at Carnegie Mellon and MIT, building tools with students, and consulting on industrial developments—has convinced me of the validity of their central premise. As Tony Hoare famously put it in his Turing Award lecture [31]:

> *There are two ways of constructing a software design: One way is to make it so simple there are obviously no deficiencies and*

> *the other way is to make it so complicated that there are no obvious deficiencies.*

A commitment to simplicity of design means addressing the essence of design—the abstractions on which software is built—explicitly and up front. Abstractions are articulated, explained, reviewed and examined deeply, in isolation from the details of the implementation. This doesn't imply a waterfall process, in which all design and specification precedes all coding. But developers who have experienced the benefits of this separation of concerns are reluctant to rush to code, because they know that an hour spent on designing abstractions can save days of refactoring.

In this respect, the Alloy language and its analysis are a Trojan horse: an attempt to capture the attention of software developers, who are mired in the tar pit of implementation technologies, and to bring them back to thinking deeply about underlying concepts.

That is why I have chosen the title *Software Abstractions* for this book. The lure of coding, and pressure to deliver elaborate features on short schedules, often draw programmers away from designing abstractions to coping with the intricacies of transient technologies, and to inventing clever tricks to overcome their limitations. If we focused instead on the underlying concepts, and struggled not for small performance gains or ever more complex features, but for simplicity and clarity, our software would be more powerful, more dependable, and more enjoyable to use. Like the best artifacts of civil and mechanical engineering, the best software systems would be a marriage of utility and beauty. And as software designers, we'd have more fun: we'd spend less time working around basic structural flaws in our software, and our ideas would have more lasting impact.