ods applied to those (simply) critical applications on which the future of a company depends or whose software systems can affect society at large.

A DANGER OF THIS FM-LIGHT approach is that formalists will obtain no thanks for their efforts! Many contributions of formalism are now absorbed into everyday computing know-how (for example, context-free grammars, finite-state diagrams, and so on), yet there is still a question as to the impact of formal methods. When states, data-type invariants, retrieve functions, loop invariants, rely/guarantee-conditions are all part of general computer knowledge, the formalists will be challenged to justify their continuing research on other topics. ▮

LIGHTWEIGHT FORMAL METHODS

Daniel Jackson and Jeannette Wing,
Carnegie Mellon University

Many benefits promised by formal methods are shared with other approaches. The precision of mathematical thinking relies not on formality but on careful use of mathematical notions. You don't need to know Z to think about sets and functions. Likewise, the linguistic advantages of a formal notation rely more on syntax than semantics.

Mechanical analysis, in contrast, is a benefit unique to formal approaches. An engineer's sketch can communicate ideas to other engineers, but only a detailed plan can be rigorously examined for flaws. Informal methods often provide some analysis, but since their notations are generally incapable of expressing behavior, the results of the analysis bear only on the properties of the artifact's description, not on the properties of the artifact itself.

For everyday software development, the purpose of formalization is to reduce the risk of serious errors in specification and design. Analysis can expose such errors while they are still cheap to fix. Formal methods can provide limited guarantees of correctness too, but, except in safety-critical work, the cost of full verification is prohibitive, and early detection of errors is a more realistic goal.

To make analysis economically feasible, the cost of specification must be dramatically reduced, and the analysis itself must be automated. Experience (of several decades) with interactive theorem proving has shown that the cost of proof is usually an order of magnitude greater than the cost of specification. And yet the cost of specification alone is often beyond a project's budget. Industry will have no reason to adopt formal methods until the benefits of formalization can be obtained immediately, with an analysis that does not require further massive investment.

Existing formal methods, at least if used in the conventional manner, cannot achieve these goals. By promoting full formalization in expressive languages, formalists have unwittingly guaranteed that the benefits of formalization are thinly spread. A lightweight approach, which in contrast emphasizes partiality and focused application, can bring greater benefits at reduced cost. What are the elements of a lightweight approach?

PARTIALITY IN LANGUAGE. Until now, specification languages have been judged primarily on their expressiveness, with little attention paid to tractability. Some languages—such as Larch—were from the start designed with tool support in mind, but they are the exception. Tools designed as an afterthought can provide only weak analysis, such as type checking. The tendency (in Z especially) to see a specification language as a general mathematical notation is surely a mistake, since such generality can only come at the expense of analysis (and, moreover, at the expense of the language's suitability for its most common applications).

PARTIALITY IN MODELING. Since a complete formalization of the properties of a large system is infeasible, the question is not whether specifications should focus on some details at the expense of others, but rather which details merit the cost of formalization. The naive presumption that formalization is useful in its own right must be dropped. There can be no point embarking on the construction of a specification until it is known exactly what the specification is for; which risks it is intended to mitigate; and in which respects it will inevitably prove inadequate.

PARTIALITY IN ANALYSIS. A sufficiently expressive language, even if designed for tractability, cannot be decidable, so a sound and complete analysis is impossible. Most specifications contain errors, and so it makes more sense to sacrifice the ability to find proofs than the ability to detect errors reliably. A common objection to this approach is that it reduces analysis to testing: No reported errors does not imply no actual errors. But this much-touted weakness of testing is not its major flaw. The problem with testing is not that it cannot show the absence of bugs, but that, in practice, it fails to show their presence. A model checker that exhausts an enormous state space finds bugs much more reliably than conventional testing techniques, which sample only a minute proportion of cases.

PARTIALITY IN COMPOSITION. For a large system, a single partial specification will not suffice, and it will be necessary to compose many partial specifications, at the very least to allow some analysis of consistency. How to compose different views of a system is not well understood

Daniel Jackson *is an assistant professor of computer science at Carnegie Mellon. After receiving a BA in physics from Oxford, he worked as a programmer for Logica UK. He has a PhD in computer science from MIT. He is interested in automated analysis of specifications and designs; requirements; and reverse engineering. E-mail daniel.jackson@cs.cmu.edu.*

Jeannette Wing *is an associate professor of computer science at Carnegie Mellon. She received her SB, SM, and PhD in computer science all from MIT. Her research interests are in formal methods, programming languages, and distributed systems. E-mail wing@cs.cmu.edu.*

and has only minimal support from specification languages, since it does not fit the standard pattern of "whole-and-part" composition.

MUCH OF WHAT WE SAY HERE IS AT ODDS with the conventional wisdom of formal methods. The notion of a lightweight approach is radical, however, only in its departure from a dogmatic view of formal methods that is detached from mainstream software development. In the broader engineering context, the suggestion of pragmatic compromise is hardly new.

A lightweight approach, in comparison to the traditional approach, lacks power of expression and breadth of coverage. A surgical laser likewise produces less power and poorer coverage than a light bulb, but it makes more efficient use of the energy it consumes, and its effect is more dramatic. ∎

Industrial Practice

WHAT IS THE FORMAL METHODS DEBATE ABOUT?

Anthony Hall, ***Praxis***

It is extraordinary that formal methods cause such fierce debate. Some proponents seem committed with an almost religious fervor; some opponents seem hostile beyond all reason. As far as I know, no issue in software engineering causes as much passion, unless it is the use of the goto statement.

One reason for this polarization may be that the two sides are arguing from completely different premises. Perhaps the argument has been between those who say that formal methods are essential because they are the only way to gain assurance and those who say formal methods are impossible because they are too expensive. No amount of argument will resolve that difference unless the two sides start to recognize each other's objectives.

I have been using formal methods in real projects for the past 10 years, and recently I have begun to see a fundamental shift in the argument. Ten years ago, the argument was that formal methods were hugely expensive (they were!) but that you had to use them because there was no other way to ensure that your software was correct. Now, the argument is quite different. We know that it is possible to produce software, even critical software, without formal methods; we also know that it is horribly expensive. What is only recently becoming clear is that it is practical to produce software, even noncritical software, using formal methods; it is also, as far as we can tell, *cheaper* to do it that way.

I say "as far as we can tell" because it is notoriously difficult to get any useful information from software metrics. What I can do is describe some of the projects we have done at Praxis and how we perceive the costs and benefits of using formal methods.[1-2]

Anthony Hall is a principal consultant with Praxis, a British software engineering company. He led the analysis and design team on CDIS and has promoted the use of formal methods on many projects. His current interests are in formal aspects of software architectures and in tool-based verification of formal specifications and designs. E-mail jah@praxis.co.uk.

One of the largest applications of formal methods I know of is a project we completed a few years ago to develop an air traffic control information system called CDIS. This is a safety-related system, but there was no regulatory pressure to use formal methods for that reason. However, we wanted to make sure that we understood the requirements accurately and decided to use formal methods at the early stages of the life cycle to help us do that. We therefore wrote a formal specification of the whole system as the basis for our development. There are about 150 user-level operations in CDIS (the final system is about 200,000 lines of code), so the specification is a large document (about 1,000 pages). This in turn means that we are making fairly "shallow" use of formality—we did not attempt any proofs of consistency or of particular properties. Nevertheless, we found the specification enormously useful in pinning down just what it was that we were going to build.

The system specification was not the only way we used formality on CDIS. We also wrote a similar specification of the main design-level modules, again at a shallow level. In one particular part of the design we used formality in a much deeper way, writing detailed process specifications and attempting to prove them correct. The fact that some proofs failed demonstrated that our design was in fact incorrect, and as it turned out, incorrect in a way that might well have escaped detection in our tests. Fault metrics for CDIS confirmed our hope that it would be of higher quality than systems built using conventional methods. They also showed an unusual distribution in that, unlike many other systems, very few of the faults that survived system test into the delivered system were requirements or specification faults.

Most interesting is the fact that none of this good news cost us anything—our productivity on the project was as good as or better than if we had done it conventionally. I believe that one reason is that the work we put in at the early stages was effective in finding lots of errors that would, if we had not found them, have proved very expensive to correct later. The formal specification enabled us to find these errors effectively.

While CDIS is an example where formal methods at the front of the life cycle pay off, they can also show economic benefits at the code and test stages. For Lockheed, we have recently been analyzing the code for the avionics software for the C130J.[2] The software is coded in the Spark-annotated subset of Ada, working from specifications in the Software Productivity Consortium's Core notation. Here, too, many people would expect that the use of Spark would add to the software's cost, while improving its quality. In fact, however, the added quality decreases the cost