Software Mythbusters Explore Formal Methods

Ciera Jaspan, Michael Keeling, Larry Maccherone, Gabriel L. Zenarosa, and Mary Shaw, *Carnegie Mellon University Institute for Software Research*

In spring 2009 a graduate seminar at Carnegie Mellon discussed a set of significant software engineering papers, including many of the *IEEE Software* "Top Picks." This article summarizes our discussion of Anthony Hall's 1990 "Seven Myths of Formal Methods" (see Figure 1).[1] The discussion group included 14 students and faculty. Three participants had over nine years of industrial software development experience, nine had two to five years, and two had less than two years.

Participants' backgrounds included embedded real-time systems, medical informatics, enterprise applications, development tools, military systems, business process management, Web applications and services, and desktop applications. Most had used UML, and three had used more-formal methods in industrial settings, including executable UML (xUML), Promela, the Software Requirements Engineering Methodology (SREM), the Distributed Computing Design System (DCDS), predicate logic, and language-based type systems for customized analysis.

Anthony Hall's "Seven Myths of Formal Methods" (Sept./Oct 1990, pp. 11–19) is the subject of the final installment of the updates on the seven distinguished articles selected from the *Software* editorial boards' 25th-anniversary top picks list. We decided to try a different format for this update. When Mary Shaw told me about her plan to cover Hall's piece in her graduate course, I asked Shaw and her students and colleagues at Carnegie Mellon University's Institute for Software Research to expand and refocus their discussion with a slant to produce this update. Hall graciously provided them with irreplaceable guidance direct from the source. Here's the result of that stimulating intellectual effort.

—*Hakan Erdogmus, Editor in Chief*

The Power of Myth

A myth isn't simply a refutable claim; in its purest form it's a heroic narrative illuminating the human condition. It can establish a model for behavior and uphold social traditions. Myths have enduring appeal—people want to believe in them, notwithstanding contradictory evidence. This "wish to believe" elevates a myth from a simple, objectively testable statement to a phenomenon whose appeal and persistence require interpretation.

During our discussions it became clear that the space of formal methods is much richer now than when Hall identified the seven myths. So, we take formal methods to include any methods based on discrete symbolic systems that allow precise specification and analysis of software. This interpretation includes not only formal logics, but also notations such as Spec#, the Java Modeling Language (JML), the Standard Annotation Language (SAL), and domain-specific languages (DSLs). Indeed, certain UML diagrams can be formal methods when used for analysis as well as description.

With this broader view, four themes emerged in our discussions:

Figure 1. The Seven Myths. In "Seven Myths of Formal Methods," Anthony Hall identified and challenged seven myths about formal methods.[1]

> **Myth 1:** Formal methods can guarantee that software is perfect.
> **Myth 2:** Formal methods are all about program proving.
> **Myth 3:** Formal methods are only useful for safety-critical systems.
> **Myth 4:** Formal methods require highly trained mathematicians.
> **Myth 5:** Formal methods increase the cost of development.
> **Myth 6:** Formal methods are unacceptable to users.
> **Myth 7:** Formal methods are not used on real, large-scale software.

- *Formal methods for both understanding and precision.* We see formal methods used in situations where the primary benefit is for precisely understanding and specifying requirements. We've benefited from this type of use even without proofs or model checking.
- *Formal methods embedded in tools and languages.* Industry is adopting formal methods that are embedded in tools or languages; developers who use these tools don't always recognize the connection to formal methods.
- *Formal methods' demand for mathematical maturity.* Embedding formalisms in tools or languages reduces the need for certain mathematical skills, but developers still need some mathematical maturity to effectively apply many formal methods.
- *Formal methods in agile practices.* The desire for agility might appear to conflict with formal methods, but our experience with using them to enable understanding, communication, and system evolvability shows otherwise.

We discuss how each of these themes reflects the current status—both the validity and the perception—of Hall's myths.

Formal Methods for Both Understanding and Precision

In our experience, emerging formal methods continue to debunk the myths of guaranteeing software perfection and proving total program correctness by following one of two philosophical paths:

- *They allow for more precise understanding of software requirements.* Our experiences continue to reaffirm Hall's observations—that even without verification, the mere act of formally specifying requirements can make crucial distinctions early in the process. DSLs, for instance, provide precise formalisms that help achieve understanding, even before automatic verification (see the "Formality for Understanding" sidebar).
- *They can make strong guarantees about narrow aspects of software.* Spec# and SAL both allow developers to prove specific, narrow classes of assertions, but they make no claims about other aspects of program correctness. This targeting makes the techniques highly adoptable and cost-effective.

Our observation of the increasing industry use of formal methods provides growing evidence that refutes myths 1 and 2. We believe the key to successful adoption of formal methods is opportunistic application (see the "Cost-Effective Abstractions" and "Formality for Understanding" sidebars). Which parts of the software are most critical? Which are used most? Where are the greatest degrees of uncertainty?

Myths 1 and 2 seem to arise from our desire for the simplicity of a single formalism for specifying all aspects of a system and for the reassurance that proof would provide. Such generality adds power but demands mathematical sophistication. Specification without verification or narrowly focused techniques can skirt around these myths. Thus,

Myths 1 and 2: Partially busted by continued and emerging uses of formal methods.

Formal Methods Embedded in Tools and Languages

We hear developers say they wouldn't use formal methods on a serious project, but these same developers create UML state diagrams or use language-based verification such as SAL. Unlike older approaches, these methods mask the underlying formality: the formal notations are embedded in tools, then marketed as diagramming techniques, languages, or lightweight verifiers. This maturation of the tools relieves humans of tedious, error-prone work.

Delving into our own reactions to myths 3 and 7, we realized that belief in these myths stems from the term "formal methods" itself; we don't see formal methods' mythic power extending to specific tools such as Microsoft's Static Driver Verifier and SAL. These tools hide their formality behind automation and abstraction,

---

Formality for Understanding

One class participant shared his experience using a formal method in an agile setting:

> *When a change was needed in a particularly confusing part of the code which had previously been written with flags, conditionals, and loops, we created a state diagram so we could understand what was going on. Discussion about how to ensure this documentation was avail-* *able for future changes led to the idea of having the code automatically consume the state machine specification. We created a DSL to capture our specification, which was checked for self-consistency and completeness upon loading. Offline checking for unreachable states, traps, and deadlocks was also envisioned.*

This change helped both understanding and accuracy.

Cost-Effective Abstractions

Cost-effective formal methods must match notations to problems. One class participant shared the challenges he faced in using a formal method that mostly matched the problem domain but lacked the ability to express a few important temporal properties concisely and in a readable manner:

We found that predicate logic allows us to elegantly specify most of the input validation requirements for our line of Web applications for electronic data capture. In our specifications, we list the data entry failure cases for a given set of input fields on a form along with the corresponding error message to display. [For example:]

When Product = Shirt AND Inseam <> NULL
    Error (*"Inseam must be blank when Product is 'Shirt.'"*)
When EXISTS (PreviousOrder) WHERE
    PreviousOrder.Cust = ThisOrder.Cust AND
    PreviousOrder.ShipDate = ThisOrder.ShipDate
    Warn (*"Another order (*{**PreviousOrder.OrderID**}*)
    is already scheduled to ship on this date."*)

However, we found a few complex requirements difficult to express concisely—making them equally hard to read without accompanying prose. The following example describes a date consistency check when out-of-order report submissions are allowed. It states that users should be warned whenever the report being submitted ends after a report for a later cycle starts; the message should reference the earliest reporting period with the earliest start date.

When EXISTS (FutureReport) WHERE
    FutureReport.CycleNumber > ThisReport.CycleNumber AND
    FutureReport.StartDate < ThisReport.EndDate AND
    [NOT EXISTS (OtherFutureReport) WHERE
      OtherFutureReport.CycleNumber > ThisReport.CycleNumber AND
      OtherFutureReport.StartDate < FutureReport.StartDate]
    Warn (*"The end date for this report is later than the start date of a future report*
    (*{**FutureReport.StartDate**} for cycle {**Min(FutureReport.CycleNumber)**}).*"*)

It took some time—perhaps longer than necessary—to specify this requirement and for the implementer to understand it. Although referencing the correct unique date and cycle number is clarified—allowing users to more readily explain or investigate the report date inconsistencies—it could be more cost-effective if it is clearly explained in prose.

which enables immediate use. This suggests that although the myths persist, belief is waning. Thus,

Myths 3 and 7: Busted by masking the formality of formal methods.

Formal Methods' Demand for Mathematical Maturity

Formal methods are commonly taught in computer science and software engineering programs, both graduate and undergraduate. The IEEE and ACM curriculum guidelines for software engineering recommend topics in mathematics and formal methods. At Carnegie Mellon, one of five core courses in the Master of Software Engineering program is devoted to formal methods, and the computer science undergraduate degree requires an introduction to formal methods. These courses emphasize selecting appropriate formal methods for a given problem as a key to success.

On the basis of personal experience, we believe students should study set theory, combinatorics, elementary graph theory, predicate logic, and state machines. Although some current formal methods, such as SAL and JML, require significantly less mathematical sophistication than earlier methods, selecting a method still requires critical judgment and some degree of mathematical maturity—someone who understands the choices and their ramifications.

We still see resistance to mathematical techniques among software developers, perhaps because they lack appropriate mathematical education, perhaps because mathematics retains a special mystique. Although the belief in Myth 4 is diminishing, it still persists. Thus,

Myth 4: Plausible because appropriate selection requires some mathematical maturity

Formal Methods in Agile Practices

Agile methods are one of the more interesting development philosophies to emerge in the last 20 years. Although at first glance, agile and formal methods seem incompatible, we see many opportunities to combine them effectively. Most important, formal methods enable effective communication (see the "Formal Methods for Communication" sidebar) and user involvement. Developers commonly use DSLs to specify aspects of systems in notations that closely match the systems' problem domain. Developers can use these specifications for code generation or runtime configuration (see "Formality for Understanding" sidebar).

By promoting direct implementation as well as developer and customer comprehension, these formal methods let systems evolve by simply modifying their domain-based descriptions, which improves communication and reduces costs. Their simplicity and understandability make them palatable to developers and users whose belief in myths 5 and 6 might persist. Thus,

Myths 5 and 6: Partially busted through language support.

Myths have an enduring appeal that resists objective evidence. People see relationships in data where none exist through a process psychologists call "illusory correlation." These biased observations reinforce the

myths despite objective evidence to the contrary.

Sometimes changing the story behind a myth changes people's attachment to the myth. Consider the wolf, often cast in myths as a fierce and aggressive animal, much to be feared. Domesticating the wolf gave us "man's best friend." This domestication produced St. Bernards (mountain rescue), Doberman pinschers (protection), golden retrievers (companions and guides), and a variety of hunting and working breeds. Both wolves and dogs are canines, but the wolf's mythic stature gave way to the dramatically different image of a companion and house pet.

We've seen a similar domestication of formal methods over the past 20 years. Formal methods that were once wild and ferocious have been tamed for use in a variety of specific ways. Some folks say that when a formal method is automated or embedded in a tool, it's no longer a true formal method; in other words, the domestication of the formal method has bred out its ferocity. Without accepting the identification of formality with ferocity, we note that such automation might sacrifice generality in favor of power for a specific task. But is our objective to preserve the ferocity of the wild methods, or is it to tame their complexity in the service of software developers? ⑩

Acknowledgments

Reference

1. A. Hall, "Seven Myths of Formal Methods," *IEEE Software*, vol. 7, no. 5, 1990, pp. 11–19.

For more information on this or any other computing topic, please visit our Digital Library at www.computer.org/csdl.

Formal Methods for Communication

One class participant shared his experiences working on a large US Department of Defense system modeled in xUML:

Using xUML, developers defined a single, verified (through simulation), platform-independent model that each service branch generated into a platform-specific implementation. Although not an agile project, the system's experimental nature required time-boxed releases to enable close communication among customers and developers. For several months the model served as the primary medium of communication among customers and developers. Customers, who were domain experts but neither seasoned programmers nor mathematicians, found it easier to understand the model and troubleshoot problems than code from previous systems.
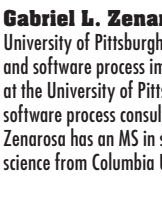
About the Authors

Ciera Jaspan is a graduate student in the Software Engineering PhD program at Carnegie Mellon University. Her research interests include software frameworks, cost-effective specification and program analysis systems, and software engineering education. She was previously a lead developer at Vizolutions, working on oil borehole surveying software and content management systems. Jaspan has a BS in software engineering from California Polytechnic State University, San Luis Obispo. She's a member of the ACM. Contact her at ciera@cmu.edu.
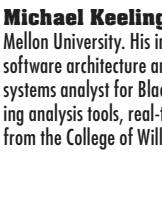
Michael Keeling is a graduate student in the software engineering MS program at Carnegie Mellon University. His interests include the pragmatic application of software engineering methods, software architecture and design, and the human aspects of software engineering. He was previously a systems analyst for Black Knight Technology, where he worked on a variety of software projects including analysis tools, real-time systems, and Web-based applications. Keeling has a BS in computer science from the College of William and Mary. Contact him at mkeeling@neverletdown.net.

Larry Maccherone is a graduate student in the software engineering PhD program at Carnegie Mellon University. His research interests include agile measurement, analysis, and visualization for software and systems engineering. He was previously chief engineer and CEO of Comprehensive Computer Solutions, a systems integrator for factory floor automation, and founded QualTrax, which creates software for measurement and management for ISO-9000 and other standards compliance. Contact him at larry@maccherone.com.

Gabriel L. Zenarosa is a graduate student in the industrial engineering PhD program at the University of Pittsburgh. His research interests include software test automation, formal methods, and software process improvement. He was previously a software quality assurance test engineer at the University of Pittsburgh National Surgical Adjuvant Breast and Bowel Project, an independent software process consultant, a client support engineer at Nyfix, and a software development consultant. Zenarosa has an MS in software engineering from Carnegie Mellon University and an MS in computer science from Columbia University. He's a member of the ACM. Contact him at gzen@cs.cmu.edu.

Mary Shaw is the Alan J. Perlis Professor of Computer Science at Carnegie Mellon University. Her research interests are value-based software engineering, everyday software, software engineering research paradigms, and software architecture. Shaw has a PhD in computer science from Carnegie Mellon University. She's a fellow of the ACM, IEEE, and American Association for the Advancement of Science. Contact her at mary.shaw@cs.cmu.edu.