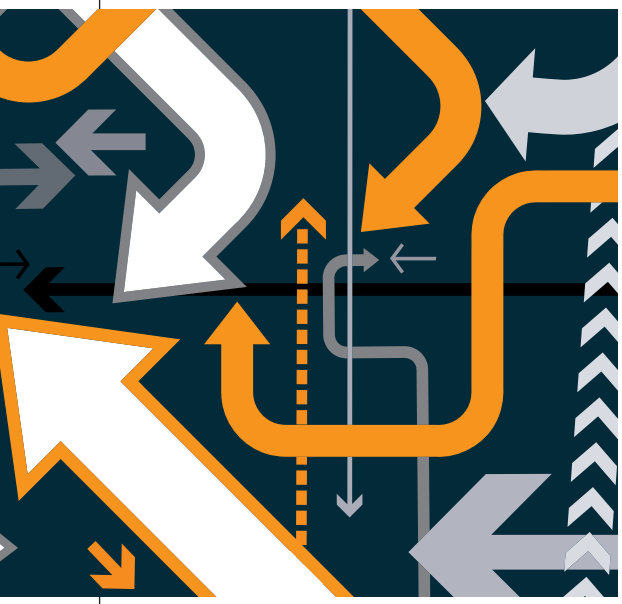

FORMAL VERSUS AGILE: SURVIVAL OF THE FITTEST?

Sue Black, *University of Westminster*
Paul P. Boca, *Hornbill Systems Ltd.*
Jonathan P. Bowen, *Museophile Ltd.*
Jason Gorman, *Codemanship Ltd.*
Mike Hinchey, *Lero—the Irish Software Engineering Research Centre*

The potential for combining agile and formal methods holds promise. Although it might not always be an easy partnership, it will succeed if it can foster a fruitful interchange of expertise between the two communities.

Software engineering as a discipline has gone through many phases. Barry Boehm describes this well in his view of 20th-/21st-century software engineering,[1] which has evolved from hardware engineering in the 1950s to software crafting in the 1960s, formality and the waterfall process in the 1970s, productivity and scalability in the 1980s, concurrent versus sequential processes in the 1990s, and agility and value in the 2000s. Each phase proved to be either a progression from or reaction to the previous one.

We argue that formal methods have been around since Charles Babbage's and Ada Lovelace's work on the difference and analytical engines: Brian Randell[2] points out that a concern with correctness was already present in the pre-electronic phase: Babbage wrote about the "Verification of the Formulae Placed on the [Operation] Cards."

Alan Turing provided what is probably one of the earliest examples of a formal proof, in which he proved a result about what was—and, by implication, was not—computable.[3] But formal methods probably came into their own in software development terms in the 1970s.

Agile methodologies are relatively new in comparison. In 2001, the Agile Manifesto (http://agilemanifesto.org) put forward four main-value preferences that focused on the following: responding to change, individuals and interaction, working software, and customer collaboration.

This can be seen as a reaction to the previous cumbersome waterfall model which focused on formalizing the customer's requirements at the beginning of the life cycle and delivering a product at the end of the life cycle, with not much interaction with customers in between. Agile has been said to work best with small groups of clever people (http://doi.acm.org/10.1145/1028664.1028720), but this might limit its scope.

Research into integrating formal methods and agile approaches is reasonably new. The First South East European Workshop on Formal Methods (SEEFM 2003) provided possibly the first substantial venue (http://delab.csd.auth.gr/bci1/SEEFM03) at which the two areas could mingle. Research carried out since then has mainly

0018-9162/09/$26.00 © 2009 IEEE Published by the IEEE Computer Society

focused on new formal methods, integrating formal methods into agile ones, and assessing the agility of formal methods.

NEW METHODOLOGIES

eXtreme Formal Modeling (XFM; http://doi.acm.org/10.1145/1109118.1109120) is an agile methodology that focuses on getting a system's specification transformed from a natural language into a formal model. The technology at this methodology's heart, *model checking*, checks the user stories expressed as linear temporal logic formulas. Case studies—the traffic-light controller and DLX pipeline—illustrate this approach.

The XFun methodology (http://delab.csd.auth.gr/~bci1/SEEFM03/seefm03_03.pdf) combines the unified process with X-machines, thus allowing systems to be built that are correct and reliable with respect to the given user requirements. The requirements are translated into X-machines verified with model-checking technology to determine whether various safety properties hold.

Formal methods can add value in the agile domain, acting as a sanity check and safety net.

INTEGRATING FORMAL METHODS INTO THE AGILE PROCESS

Some researchers have argued that model checking can be used to verify evolving agile frameworks by capturing changes at the architectural level[4]—such as the Symbolic Model Verifier (SMV) model checker, with the specifications captured in CTL. X-machines has been used in different ways to bolster agile methods, such as for documentation[5] and for modeling use cases.[6]

Three areas of eXtreme Programming can benefit from formal methods[7]: unit testing, incremental development, and refactoring. The concept of specification-driven development[8] combines two compatible and complementary approaches: Design by Contract and test-driven development.

FORMAL METHODS' AGILITY

At the 6th International KeY symposium in 2007, Reiner Hähnle gave a talk on agile formal methods (http://i12www.iti.uni-karlsruhe.de/~key/keysymposium07/slides/haehnle-agile.pdf). Hähnle's main thrust was to argue that newer formal methods align well with the agile process. He then summarized what is required for a formal method to be agile.

Hähnle argued that the KeY methodology, a formal software development tool for object-oriented systems, is such a method. Peter Amey[9] argued that the correctness by construction approach for developing correct software combines formal and agile methods because it "takes precise notation from the former and from the latter incremental development."

As can be seen, the question of whether formal and agile methods can be combined—and if so, how—has yet to be answered definitively.

FRIENDS NOT FOES

There is no escaping formal methods in software development. They are everywhere: Programming languages have a formal semantics, program generators such as YACC generate finite-state machines, coding standards are language subsets defined with some "formal" rationale such as preventing program crashes. Some IDEs, like Visual Studio, have integrated static analysis tools, and the processor on which the compiled code will run has been formally verified.

These methods work in the background, letting developers go about their work while taking the underpinnings for granted. Formal methods can be in the foreground too, making precise the system's intended behavior and assuring the end product's correctness. These methods have been deployed in many domains, including automobiles, trains, air-traffic control systems, and medical systems. There is compelling evidence that they add value.[10] The deployment of formal methods has largely been within the traditional software development framework.

We argue that formal methods can add value in the agile domain, acting as a sanity check and safety net. Moreover, formal methods can open up possibilities currently closed to the agile world: deployment in safety- and mission-critical domains, for example.

ADDING VALUE

We argue our case by demonstrating that formal methods can add value in four key areas of software development.

Testing

Writing tests prior to implementation lies at the heart of agile development, continuously evolving a regression suite for onward development. Developers run the regression suite for subsequent changes to the code base to ensure that functionality does not degrade. But how do we determine when we have tested enough? Have adequate edge cases been considered? Most interesting domains require an infinite number of test cases, which is clearly impractical if not impossible to achieve. So the developer must decide which tests will be considered and which ruled out.

Autogenerating test programs, achieved through scripting in Python or Perl, can partially address the coverage problem. Developers can use a functional programming language such as Standard ML instead of one of these traditional scripting languages to map a high-level specification—expressed as a function—over a list of tuples built up from the domains of interest to yield the test cases.

This approach, while keeping the agile philosophy's spirit, has its drawbacks—it works for finite domains only; infinite domains cannot be enumerated. A random selection of test vectors could be generated for the infinite domains, but then the same questions would apply.

Static-analysis and theorem-proving tools provide a more reliable solution. The idea is to annotate the code in various places with logical statements asserting properties that should hold true. These assertions can be checked without running the code to see whether they are violated.

Static checkers have reached a level of sophistication and maturity that allows a high percentage of assertions—typically 97 percent in the SPARKAda toolset, for large-scale examples—to be checked automatically.

Requirements

In the agile world, requirements change rapidly—developers expect this and are not fazed by the possibility of having to discard their work and start over. Although this way of working eventually creates a product that can be shipped, there must be requirements traceability, otherwise there can be little guarantee that the end product will meet the customer's requirements.

Once again, assertions can help. The informal requirements—typically expressed in stylized natural language—must be translated into the formal notation in which the assertion is expressed and then embedded in the code. This process has its pitfalls, as errors may be introduced during the translation stage. Once expressed as assertions, the requirements can be machine-checked for inconsistencies. Finding errors in the requirements at an early stage will reduce the amount of rework. Alas, formal methods cannot help with the problem of requirements creep—it is a fact of life.

Refactoring

Developers change code to improve its performance, make it more maintainable, and beautify it. Such changes are made with the safety net of a regression suite, the assumption being that if a change affects functionality, this will be caught. But refactoring is a human activity, and therefore prone to error. Testing alone cannot guarantee that a refactoring step has not changed the code's meaning. It is conceivable that refactoring may introduce functional errors that running the regression suite will not detect. Assertions can be helpful here: Each refactoring step can be machine-checked to see whether the contract still holds. This is by no means foolproof either—the assertion might be incorrect or too wide in scope—but the two approaches together can give greater assurance that errors have not been introduced.

Greater reliability in refactoring code can be achieved through use of automated assistants: Common code changes that a developer would apply can be captured as correctness-preserving transformations and applied automatically, ensuring that refactoring preserves meaning. But tools cannot invent refactorings for the user, for this is an intellectual activity that requires inspiration—a Eureka step.

What tools *can* help with, though, is encouraging the developer to split the refactoring process into several manageable steps, taking charge of the housekeeping required to ensure that these steps are correct; some steps could simply be the result of applying "canned" transformations, while others might require manual proof steps.

In the future, more software will be adaptive, changing itself to cope with new requirements or unforeseen circumstances or to ensure resilience in harsh environments.

Documentation

Many developers abhor anything to do with documentation, but this is not a problem in the agile world: Pairwise programming is often used as an argument against writing down technical information. By rotating teams, everyone gets to learn how the system being developed works.

This might work in an environment where staff turnover is low, but what if senior team members leave? How will the knowledge be retained? Some might therefore regard failure to document as a risk associated with using the agile development process.

There are many examples of long-lived software, including shutdown loops on nuclear power plants running for more than 30 years and a NASA spacecraft running for more than four decades and soon to be unreachable by communications link. We typically don't build our systems to be that long-lived, but we are focusing more on evolving our software and even critical applications, with many organizations having an evolve-buy-build policy. Moreover, in the future, more software will be adaptive, changing itself to cope with new requirements or unforeseen circumstances or to ensure resilience in harsh environments. Without documentation, in the long term it will be impossible to tell what such systems were originally intended to do as opposed to what has resulted from adaptation and

evolution. By introducing some formality into the agile process, we can get documentation for free, such as assertions and specifications of tests to be generated.

Parallelism

Our discussion has deliberately avoided any mention of parallelism. Sequential programs are hard enough to get right; moving to parallelism brings new challenges: avoiding deadlock, livelock, and race conditions. Such phenomena might go undetected through testing alone, and so it is here that the formal-methods safety net becomes essential.

Formal methods and agile software development have the common aim of producing reliable software, and combined methodologies have been proposed.

Model checkers, such as FDR[11] (Failures Divergences Refinement), can prove that a system is deadlock free. Static analyzers, such as the one marketed by Coverity (www.coverity.com), can check code for potential race conditions. These technologies have improved in recent years, making them scalable to industrial-sized examples.

Speed and availability

As well as being scalable, formal methods tools must be fast if they are to succeed in the agile world. Agile projects have frequent delivery schedules, so verifiers, for example, must discharge verification conditions in "quick time." The agile developer does not want to be sitting around waiting for verification conditions to be discharged—the developer just wants to move on to the next feature implementation. Harnessing multicore architectures can be helpful in this respect, and research in this direction is under way.

Success depends on tool availability too—open source will ensure wide adoption within the industry. There is a move toward this, particularly in the Rodin and Deploy projects (http://rodin.cs.ncl.ac.uk; www.deploy-project.eu), where a community building freely available tools built upon Eclipse is growing. Another community initiative has provided tools for the Z specification language (http://czt.sourceforge.net). Recently, Praxis High Integrity Systems (www.praxis-his.com) announced an open source version of its SPARK toolset (www.praxis-his.com/news/sparkPro.asp). All these initiatives offer good news for formal methods and even better news for the agile community.

TOOL SUPPORT

Formal methods[12-14] are often seen as inflexible, whereas agile software development is designed to be the antithesis of this approach. However, the two approaches do have the common aim of producing reliable software, and combined methodologies have been proposed.[15] Agile methods seek to involve small incremental tasks, minimizing the need for planning.

Some formal approaches can make this difficult. Despite this, there are formal methods tools that allow a great deal of flexibility and can achieve worthwhile results with a small amount of effort.

Flexible tools

Alloy is one tool and associated language[16] that makes this approach possible. It is highly influenced by the Z notation, a well-known formal specification language based on set theory and predicate logic. Developers can use it to check models up to a specified size. However, problems in the modeled system are typically found using relatively small models in practice, so there is no practical benefit to continuing with very large models.

Other formal methods tools and approaches could also be compatible with agile methods. Specifically, model checkers could be effective if used at an early stage to ensure system correctness. Model checkers allow a complete search space to be analyzed automatically once the model has been formulated to ensure there are no inconsistencies. The model can be changed relatively quickly and rechecked following an agile methodology.

Another such tool, FDR, checks models in the style of CSP (Communicating Sequential Processes). This has been applied in the verification of protocols.[11] Other model checkers, such as Spin,[17] could also be used effectively in an agile setting.

More traditional state-based and even refinement-based approaches might also be compatible. These allow specification of software involving an abstract state that can be changed using a set of operations over time, starting with some initial state. They can also, although at greater cost and potentially less flexibility, be used to refine the specification toward an implementation in a formal manner. The Rodin tool (Rigorous Open Development Environment for Complex Systems; www.bcs.org/upload/pdf/ewic_fm07_paper2.pdf), based on the B-Method,[12] allows the refinement of a specification in formal as well as tool-supported environments. Its facilities explicitly consider changes in the formal specification and development as it attempts to minimize the amount of reproof needed when the design changes. This could be helpful if formal refinement is required in an agile development context.

Alloy example

Software typically consists of a set of operations on some defined state, with inputs to those operations and outputs from them. Checking if such operations have desired and expected properties is a useful part of software

development that aids in validating these operations. If this can be done rapidly and conveniently, it becomes compatible with the agile approach to software development.

As a specific example, consider an e-mail address book. The components of the state might be made up of people's names and their e-mail addresses. Initially, its structure might not be important, so the components could be defined as signatures in Alloy. These names and e-mail addresses could then be related together in an address book:

```
sig Name, Addr { }
sig Book { addr: Name -> Addr }
```

A desirable property could be an invariant that determines that the number of e-mail addresses associated with names in the address book can be at most one per name:

```
pred invBook (b:Book) { all n:Name
| # b.addr[n] =< 1 }
```

Software with a state normally has additional constraints on the initial state. For example, it could be seen as desirable initially for the address book to have no addresses in it:

```
pred init (b : Book) { no b.addr }
```

To check that the initial state is feasible, we can assert that for all possible initial address books the invariant holds and checks for all models up to a specified size (here, three):

```
assert initOK { all b : Book
| init[b] implies invBook[b] }
check initOK for 3
```

If there is a counterexample, it will be displayed. Given a "before" state (b:Book), an operation transforms this to an "after" state (b':Book). An "add" operation could add a name "n" and associated address "a" to the address book:

```
pred add (b, b' : Book, n : Name, a :
Addr) { b'.addr = b.addr ++ n -> a }
```

The "++" operator acts like the Z relational overriding operator, meaning that any existing address for the name is replaced. Having operations respect the invariant on the state is desirable. This can be asserted and checked for models up to a specified size:

```
assert addOK{ all b, b' : Book, n :
Name, a:Addr
| invBook[b] and add[b,b',n,a] implies
invBook[b'] }
```

Deleting an entry is similar to adding an entry, but only the name is needed as input, and the entry is removed from the address book:

```
pred del (b, b' : Book, n : Name)
{ b'.addr = b.addr - n->Addr }
```

Checking a property that an add operation, followed by a delete operation for the same name, results in the address book remaining unchanged can be asserted and checked as follows:

```
assert delUndoesAdd { all b, b', b'' :
Book, n: Name, a : Addr
|add[b,b',n, a] and del[b', b'', n]
implies b.addr = b''.addr }
check delUndoesAdd for 3
```

This uncovers a possibly unexpected counterexample where an existing name is not added but is removed by the deletion operation, thus changing the state. Finding such a problem quickly with a tool like Alloy helps ensure that the software is implemented correctly once coding starts.

This example is simple by necessity. However, Alloy has been shown to scale up to larger, more realistic problems. For example, it has been used to model the Mondex electronic purse,[18] a real product in the banking world, where security is paramount.

Compared to other formal approaches, developers can explore the security properties very quickly by modeling the system using Alloy. If used in the actual development, this would allow discovering potentially insecure situations more quickly. This would be entirely compatible with the agile software development process, allowing responsiveness to change, while simultaneously enhancing the software reliability.

A promise of synergy

Formal methods are well established in the field of high-integrity systems development.[1] Traditionally seen as inflexible, they can be quite effective if the right tool is chosen and used judiciously. Agile methods, on the other hand, promise a flexible framework for software development, which is often needed as requirements and understanding of the system change and improve. Thus, the combination of formal and agile methods promises an effective synergy if used together sensibly, with appropriate engineering judgment.

TIME, EFFORT, AND OTHER MYTHS

Many developers hold a widespread belief that formal methods are expensive and raise development costs. Even

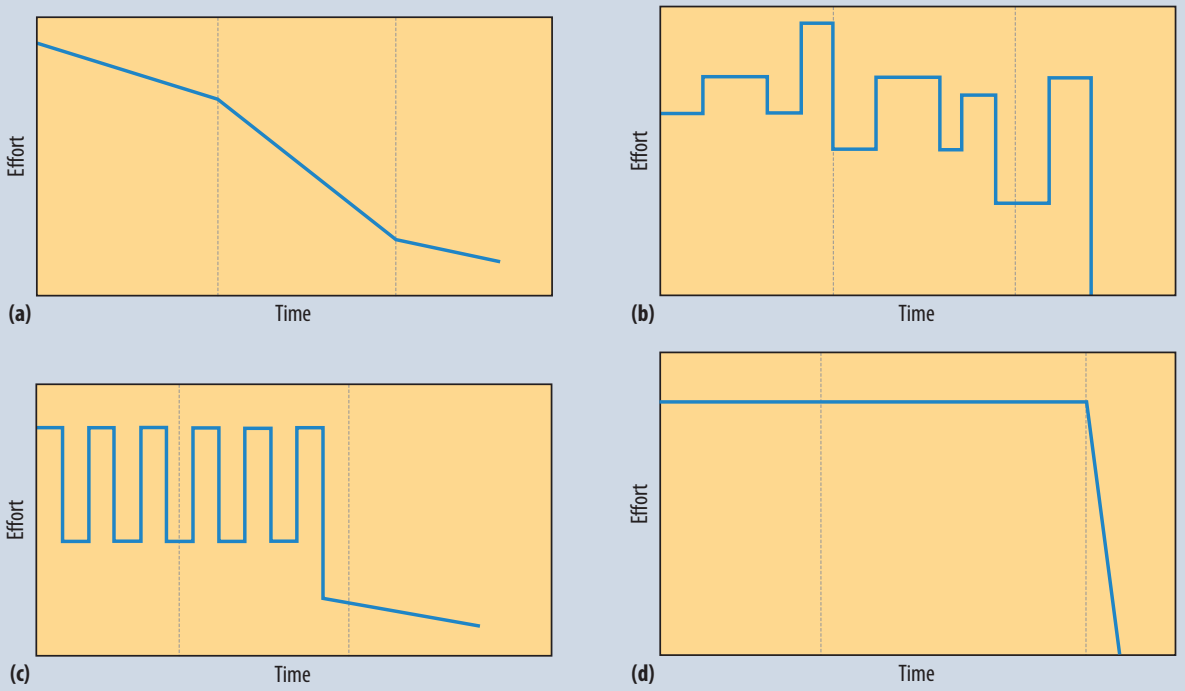

Figure 1. Effort-time curve for software development project. (a) Typical "shape" for the effort-time curve for formal development, showing high levels of effort initially being mitigated by lower effort later. (b) Typical effort-time curve for an XP development, showing story implementations as spikes. (c) Typical Scrum effort-time curve showing sprints. (d) Typical effort-time curve for the Dynamic System Development Method, emphasizing that effort can suddenly fall as resources are exhausted.

formal methods' supporters admit to this, while their detractors use the same statistics to justify avoiding formal methods, even in critical applications. "Formal methods light," as suggested by Cliff Jones, emphasizes the use of formal methods for specific parts of a system's development, rather than for the entire system. Other researchers advocate a similar approach.[19]

Experience has shown that, in many cases, formal methods can help reduce lead times and lower development costs,[20] yet the myths of increased development costs[21] and delayed development[19] persist.

How costly?

Increased setup costs can imply that development costs will be greater. Slower progress in early phases gives the impression that overall progress will be slower, yet there is no tangible reason to believe this. Figure 1 shows the typical "shape" of the effort-time curve for formal development: The cost of development closely relates to that for effort. Figure 1a also shows that reduced implementation costs quickly mitigate the significant increase in effort at the outset, which also reduces postimplementation maintenance. Further, it has been proven that formal methods can greatly help ensure that we build the right system correctly, significantly reducing the amount of perfective and even adaptive maintenance.

Faster development time?

Proponents of agile methods claim that they decrease development time and lower development costs. While there have been several great success stories, the jury has yet to rule on critical applications. Moreover, agility has nothing to do with speed. Agile methods specifically address the need to respond to customer requirements and, in particular, changing requirements. Their aim is to satisfy customers' needs rather than to be speedy.

That agile developers emphasize working code, rather than documentation, might give the impression that development is proceeding more quickly because tangible progress is being made. We do not consider this a bad thing, but we must be careful not to incorrectly assume that speed is of the essence with agile methods—nor should it be assumed that less effort is involved.

Effort-versus-time comparison

Figure 1 shows the "shape" of the effort-time graph for three popular agile methods. Figure 1b shows the typical shape of an eXtreme Programming (XP) project. The spikes in the graph depict story implementations of various lengths. Note also the explicit "death" phase where effort is followed by a direct fall in the graph.

The spikes in Figure 1c represent the sprints in Scrum. These are of roughly equal duration, with typically three to

eight sprints per development project. There is no specific death phase in Scrum, but as we might expect, following the final sprint, the level of effort falls off dramatically in postimplementation.

As Figure 1d shows, the Dynamic System Development Method (DSDM) presents a compelling, almost flat graph that ends suddenly. This occurs because DSDM focuses on an acceptable level of effort. As resources allow, developers implement functionality. If resources are limited, compromises must be made to determine implementation priorities. While the flat graph and fixed overall cost might seem attractive, this means certain requirements might never be implemented.

Moreover, the likely casualties of any resource limitations will be nonfunctional requirements, which customers often don't even realize they need, and which can have significant implications for critical applications. Formal methods perform best when highlighting many such requirements, helping us understand them, and enabling us to ensure they are fully implemented.

Something's a myth

The details of these graphs are not important, however. Rather, they show an interesting phenomenon: Notwithstanding the significant differences in the graphs' "shapes," the area beneath the curve—the total effort required and, as a consequence, the total cost—is not significantly different except in DSDM, which places a firm limitation on resources.

This means that neither formal nor agile methods offer significant reductions in effort over the competing approach. Rather, formal methods bring the advantage of certainty in dealing with critical applications, assurance, and solid documentation, while agility brings the benefit of flexibility, customer satisfaction, and tangible progress. Yet the claimed benefits of development speed and reduced effort for agile methods might be overstated.

CULTURAL DIVIDE

Many people in and out of the formal methods community make the mistake of believing that agile software development is about rapidly "throwing together" software to ensure quick delivery of valuable features, at the expense of qualities like reliability, security, and maintainability.

Agile methods have earned this dubious reputation for several reasons, but two in particular stand out. First, some confuse agile methods with rapid-application development (RAD). This technique employs user-interface prototypes to garner quick feedback from customers so that designs can evolve rapidly toward something more useful.

What agile methods and RAD share is that both are highly feedback-driven and arguably fall under the banner of evolutionary software design to a larger extent than traditional engineering methods like SSADM (Structured System Analysis and Design Method).[22] Continuous customer involvement throughout the evolutionary design process is a shared characteristic that can make doing agile methods and RAD look, on the surface, very similar.

Quick and dirty?

Most teams purporting to be doing agile software development are not applying the level of technical rigor necessary to succeed at it. Most "agile" teams have actually only adopted Scrum's project-management practices and have failed to effectively adopt "the hard disciplines" like test-driven development, refactoring, pair programming, simple design (writing the simplest code possible to satisfy the customer's requirements), and continuous integration.

The main misunderstanding about agile methods is that teams deliver software faster. Agility is not about speed, but about being responsive to changing requirements. A faster car is not necessarily a more maneuverable one.

Formal methods bring the advantage of certainty in dealing with critical applications, assurance, and solid documentation, while agility brings the benefit of flexibility, customer satisfaction, and tangible progress.

Agile projects deliver working software sooner, but they do this by creating small, frequent releases and effectively prioritizing features. This creates more and earlier opportunities to adapt to customer feedback, just as with RAD. The critical difference is that the software delivered in each iteration is of the highest quality the team can achieve (www.parlezuml.com/tutorials/agile_qa/Example_Agile_Quality_Assurance_Strategy.pdf).

In this sense, agile software development learned an important lesson from RAD: Prototypes tended to wind up being the end product the customer used. The RAD discipline hinged on the prototypes serving as a high-fidelity specification created quickly using a user-friendly interface and technologies like Microsoft Visual Basic. Once the customer had signed off on a finished prototype, RAD teams were then supposed to apply rigorous methods to build a high-quality software system in a "serious" language like C++, based on the UI prototype. All things being equal, the reality was that many RAD teams were forced to release software built around the prototype code itself, with its cobbled-together and largely untested underlying logic. Such was the psychological pitfall of showing customers something that looked like a finished product.

Incremental releases

In an agile methods project, customers see a finished product in each small release—the underlying code is of a high-enough quality for production release, should the customer demand it. Another important lesson that agile development addresses is that the first release of a software system usually fosters many such releases. Experience has shown that we can be sure a system's design will change as our understanding of the problem domain evolves and users' needs change with time.

"Embracing change" is thus a cornerstone of agile software development's values, but it is more than a noble sentiment. For agile teams to embrace long-term change release after release, the delivered software must accommodate change. Experience demonstrates that, over time, as software grows, change becomes increasingly harder and more expensive. A feature request that might have taken a day or two in the first few weeks of a project could, a year or two later, take several weeks.

Technical debt

As they deliver software, teams accrue what the agile community refers to as "technical debt" in their code. This includes things like bugs, design issues, and other code-quality problems that are potentially introduced with every addition or change to the code. These issues have a detrimental impact on developer productivity. Bug fixes divert effort from adding valuable new features. Unmanaged dependencies cause a ripple effect[23] that can turn tiny changes into mammoth wholesale restructurings. Badly thought-out naming can make code very difficult to understand, even for the developers who wrote it. As technical debt builds up, the net effect is to hamper the teams' efforts to accommodate changes to requirements.

Teams that do not take serious steps to minimize technical debt quickly find that change can become prohibitively expensive, making the project unresponsive to new customer requirements. For this reason, we find that it is not only possible to apply rigor in agile software development, it is actually necessary to succeed at it.

Making it work

We propose that by rigorously applying techniques like test-driven development (www.parlezuml.com/tutorials/tdad.ppt; www.agileitea.org/public/deliverables/ITEA-AGILE-D2.7_v1.0.pdf),[24,25] and by running our tests through practices such as adversarial pair programming and mutation testing, agile methods can lead to software of a high-enough integrity for the majority of computing applications (www-users.cs.york.ac.uk/~paige/Writing/issre04.pdf). We also see no impediments to complementing agile practices with orthogonal techniques like Design by Contract, guided inspections, symbolic execution, static analysis, model checking, and even theorem proving—provided they are applied sympathetically to the core agile values and principles and, most importantly, only when they are truly needed and will add value.

What conflict?

We see no reason why there should be any practical conflict between agile software development and formal methods. Clearly, however, a possible cultural divide currently makes the agile and formal methods communities incompatible collaborators. This divide could stem largely from a lack of understanding between the two communities. Therefore, an ongoing process of interaction and exchange might well be the best remedy for helping agile and formal methods practitioners discover and apply the best of both worlds.

Formal methods can survive in an agile world; they are not obsolete and can be integrated into it. The potential for combining agile and formal methods holds promise. It might not always be an easy partnership, and succeeding will depend on a fruitful interchange of expertise between the two communities. Conducting a realistic trial project using a combined approach with an appropriate formal methods tool in a controlled environment will help assess the effectiveness of such an approach.

Acknowledgments

Jonathan Bowen thanks Anthony Hall for inspiration with Alloy while teaching an MSc course on validation and verification at University College London in 2007. Bowen is a visiting professor at King's College London. This work was supported in part by Science Foundation Ireland grant 03/CE2/I303_1 to Lero—the Irish Software Engineering Research Centre.

References

1. B. Boehm, "A View of 20th- and 21st-Century Software Engineering," *Proc. 28th Int'l Conf. Software Eng.*, ACM Press, 2006, pp.12-29.
2. B. Randell, *The Origins of Digital Computers: Selected Papers*, Springer-Verlag, 2nd ed., 1975.
3. A.M. Turing, "Checking a Large Routine," *Report of a Conference on High Speed Automatic Calculating Machines*, University Mathematical Laboratory, Cambridge, 1949, pp. 67-69.
4. N. Niu and S. Easterbrook, "On the Use of Model Checking in Verification of Evolving Agile Software Frameworks: An Exploratory Case Study," *Proc. 3rd Int'l Workshop on Modelling, Simulation, Verification and Validation of Enterprise Information Systems*, INSTICC Press, 2005, pp. 115-117.
5. C. Thomson and M. Holcombe, "Using a Formal Method to Model Software Design in XP Projects," *Annals of Mathematics, Computing and Teleinformatics*, vol. 1, no. 3, 2005, pp. 44-53.

6. D. Dranidis, K. Tigka, and P. Kefalas, "Formal Modeling of Use Cases with X-Machines," *Proc. 1st South-East European Workshop on Formal Methods*, CD-ROM, 2003, pp. 72-83.
7. Á. Herranz and J.J. Moreno-Navarro, "Formal Agility, How Much of Each?" *Taller de Metodologías Ágiles en el Desarrollo del Software, VIII Jornadas de Ingeniería del Software y Bases de Datos* (JISBD 2003), Grupo ISSI, 2003, pp. 47-51.
8. J.S. Ostroff, D. Makalsky, and R.F. Paige, "Agile Specification-Driven Development," *Proc. Extreme Programming*, LNCS 3092, Springer-Verlag, 2004, pp. 104-112.
9. P. Amey, "Correctness by Construction"; https://buildsecurityin.us-cert.gov/daisy/bsi/articles/knowledge/sdlc/613-BSI.pdf, 2006.
10. A. Hall, "Realising the Benefits of Formal Methods," *J. Universal Computer Science*, vol. 13, no. 5, 2007, pp. 669-678.
11. S. Creese and J. Reed, *Verifying End-to-End Protocols Using Induction with CSP/FDR, Parallel and Distributed Processing*, LNCS 1586, Springer-Verlag, 1999, pp. 1243-1257.
12. J-R. Abrial, "Formal Methods in Industry: Achievements, Problems, Future," *Proc. 28th Int'l Conf. Software Engineering*, ACM SIGSOFT, 2006, pp. 761-768.
13. P.P. Boca, J.P. Bowen, and J.I. Siddiqi, eds., *Formal Methods: State of the Art and New Directions*, Springer, 2009.
14. M.G. Hinchey et al., "Software Engineering and Formal Methods," *Comm. ACM*, vol. 51, no. 9, Sept. 2008, pp. 54-59.
15. G. Eleftherakis and A.J. Cowling, "An Agile Formal Development Methodology," *Proc. 1st South-East European Workshop on Formal Methods* (SEEFM 03):Agile Formal Methods: Practical, Rigorous Methods for a changing world, CD-ROM, 2003, pp. 36-47.
16. D. Jackson, *Software Abstractions: Logic, Language, and Analysis*, MIT Press, 2006.
17. G.J. Holzmann, *The Spin Model Checker: Primer and Reference Manual*, Addison-Wesley, 2003.
18. T. Ramananandro, "Mondex, an Electronic Purse: Specification and Refinement Checks with the Alloy Model-Finding Method," *Formal Aspects of Computing J.*, vol. 20, no. 1, 2008, pp. 21-39.
19. J.P. Bowen and M.G. Hinchey, "Seven More Myths of Formal Methods," *IEEE Software*, vol. 12, no. 4, 1995, pp. 34-41.
20. M.G. Hinchey and J.P. Bowen, eds., *Applications of Formal Methods*, Prentice-Hall, 2005.
21. J.A. Hall, "Seven Myths of Formal Methods," *IEEE Software*, vol. 5, no. 7, 1990, pp. 11-19.
22. L. Rackley and A. Walker, *SSADM in Practice*, Palgrave Macmillan, 1995.
23. S.E. Black, "Deriving an Approximation Algorithm for Automatic Computation of Ripple Effect Measures," *J. Information and Software Technology*, June 2008, pp. 723-736.
24. A. Marchenko, P. Abrahamsson, and T. Ihme, "Long-Term Effects of Test-Driven Development: A Case Study," *Agile Processes in Software Engineering and Extreme Programming*, LNBIP, 2009, pp. 13-22.
25. E.M. Maximilien and L. Williams, "Assessing Test-Driven Development at IBM," *Proc. 25th Int'l Conf. Software Eng.*, IEEE CS Press, 2003, pp. 564-569.

Sue Black *heads the Department of Information and Software Systems at the University of Westminster. Her research interests are software measurement, software quality, and software evolution. Black received a PhD in software engineering from London South Bank University. She is a member of the IEEE and the ACM, a Fellow of the Royal Society of Arts, and a Fellow of the British Computer Society. Contact her at sueblack@gmail.com or www.sueblack.co.uk.*

Paul P. Boca *is quality engineering manager at Hornbill Systems, Ltd. His research interests are software quality, program transformation, and formal methods. Boca received a PhD in computer science from Queen Mary, University of London. He is a member of the IEEE Computer Society, the ACM, Formal Methods Europe, American Society for Quality, and the British Computer Society. Contact him at paul.boca@googlemail.com.*

Jonathan P. Bowen *is chairman at Museophile Ltd. He is also an emeritus professor at London South Bank University, a visiting professor at King's College London (2007-2009), and a visiting professor at the University of Westminster from 2010. His research interests are in the area of software engineering in general and formal methods in particular. Bowen received an MA in engineering science from the University of Oxford. He is a Fellow of the British Computer Society and Royal Society of Arts and a member of the IEEE and the ACM. Contact him at jpbowen@gmail.com, http://jpbowen.googlepages.com.*

Jason Gorman *is a director at Codemanship Ltd. His research interests are test-driven development, UML, and software analysis and measurement. He received a BSc in physics from the University of Surrey. He is a member of the British Computer Society, the International Association of Software Architects, and the Agile Alliance. Contact him at jason.gorman@codemanship.com; www.codemanship.com.*

Mike Hinchey *is scientific director of Lero—the Irish Software Engineering Research Centre—and a professor of software engineering at the University of Limerick, Ireland. His research interests include self-managing software and formal methods for system development. Hinchey received a PhD in computer science from the University of Cambridge. He is a senior member of the IEEE and currently chairs the IFIP Technical Assembly. Contact him at mike.hinchey@lero.ie.*