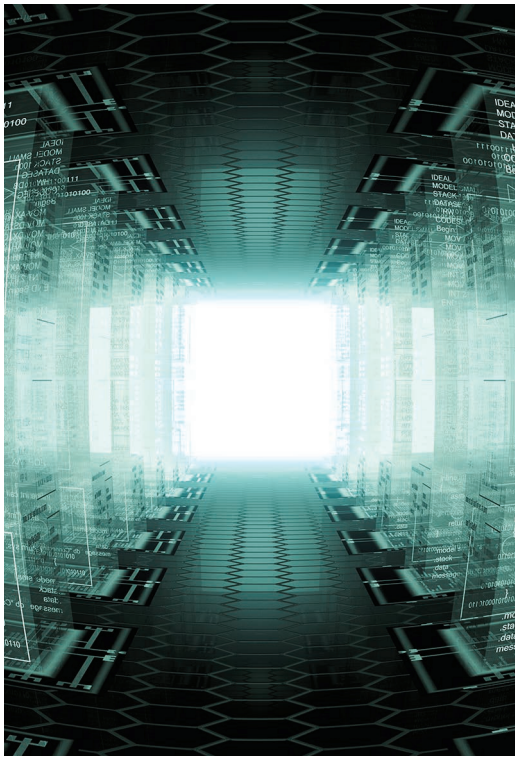

Formality, Agility, Security, and Evolution in Software Development

Jonathan P. Bowen, *Birmingham City University*
Mike Hinchey, *Lero—the Irish Software Engineering Research Centre, University of Limerick*
Helge Janicke and Martin Ward, *De Montfort University*
Hussein Zedan, *Applied Science University*

Combining formal and agile techniques in software development has the potential to minimize change-related problems.

Complex systems have always been problematic with respect to software development. Simplicity is desirable, but the reality of dealing with a customer means that requirements are likely to change, resulting in a corresponding loss of elegance in the solution.

A good software engineer will design with the knowledge that the system is likely to evolve over time, even if the exact nature of the changes is unknown. Such expertise only comes with experience and an innate aptitude, especially in the understanding and use of abstraction. Software engineering approaches such as object orientation and modularization—for example, the Z notation schema construct—can help minimize change-related problems if used carefully, following standard patterns of use. Formal methods have been advocated for improving the correctness of software systems,[1] and agile software development has been promoted by the Agile Manifesto (http://agilemanifesto.org) and others for enabling adaptive development in the face of changing requirements, typically introducing additional complexity in the process.

FORMALITY

Although using formal methods to develop a software system can be slow and cumbersome, these methods deliver lower error rates because they use a mathematical approach from the requirements specification onward.[2] Formal methods are an important software engineering technique in cases when safety and security are central aspects of the system to be designed. While a formal approach can be used to derive a proof, potentially at great cost, a formal specification also provides a framework for undertaking rigorous testing, potentially with cost savings since the specification can be used to direct the tests that are needed.[3] Without a formal specification, software testing is a much more haphazard affair.

Misunderstanding the term "formal methods" can also be an issue. The relevance of the formal approach must be understood by every team member, even if formality won't actually be used by everyone. In fact, teaching software engineers to *read* formal specifications is much easier than teaching engineers how to *write* them. Reading and understanding

 Published by the IEEE Computer Society

the specification is necessary for most members of a software development team—for example, programmers and testers—whereas specification writing requires the involvement of a much smaller number of more highly trained people.

AGILITY

Agile software development provides an iterative approach where the requirements and the associated solution evolve through the collaboration of team members, and rapid response to change is encouraged. This contrasts with the traditional view of formal methods, but modern tools can enable a much more agile approach even within formal development. For example, the RODIN tool (http://rodin.cs.ncl.ac.uk) minimizes the amount of re-proof that is needed if the system is changed. Tools like the Alloy Analyzer (http://alloy.mit.edu) are relatively easy for a capable software engineer to learn and can be quickly used in an agile manner.[4]

The concept of agile formal methods first appeared in the literature in the mid-2000s, with events such as the Formal Methods and Agile Methods (FM+AM) Workshop explicitly addressing the issue.[5] In 2009, a position paper on formal methods and agile software development was published in *Computer* to highlight the debate.[4]

While we don't really hold the view that agility may be used by "unscrupulous" developers, in general we do believe that agile developers could benefit from some training on formal methods, at least in reading formal specifications, even if developers don't apply the approach rigorously.

SECURITY

Can agile methods produce secure software, perhaps in cases in which the security properties have been formalized? Opinions regarding this question remain split. While agile methodologies such as Scrum and XP continue their advance into mainstream software development, with sometimes impressive results, there are still reservations in areas where security is a paramount concern to stakeholders. Until relatively recently there was little focus on how to integrate security best practices into agile development, although these shortcomings have existed for at least a decade.

It's necessary to integrate security activities in the agile development process and its practices.

To address this problem, it's important to make security requirements explicit and to include security concerns in the product backlog with adequate priority, so that the concerns continue to be addressed by the development team during the iterations. Agile methods strive to satisfy the customer, which frequently prioritizes the development of functionality that produces the primary business value. Security, on the other hand—because it's concerned with managing risk and preventing the developed functionality from being misused—is often unfortunately not as highly valued and therefore isn't addressed in early sprints. Consequently, it's necessary to integrate security activities in the agile development process and its practices.

One way to address security early in an agile development context is by using *evil stories*. This is an agile adoption of misuse cases to describe the functionality that an attacker would be able to exploit. The development then takes on two dimensions: to implement user stories and to avoid implementing evil stories. Another practice that integrates security principles in agile development is a technique called *protection poker,* in which security risks are quantified by the agile team. This process is similar to the widely known agile technique *planning poker*, which is used to establish effort estimates for user stories.

The recent adaptation of Microsoft's Secure Development Lifecycle (SDL) to integrate with agile products shows that security is a major concern for companies employing agile methodologies. The key problem addressed by Bryan Sullivan is how activities from the SDL can be integrated effectively into the short-release cycles that characterize agile development projects.[6] This approach divides SDL activities into three categories: every sprint, such as running automated security-analysis tools and updating the threat model; bucket requirements consisting of verification tasks, design review activities, and response planning; and one-time requirements such as the development of a baseline threat model.

So, while efforts to integrate security practices in agile methodologies are well underway, there is a risk that many actual development efforts will ignore these efforts or that development teams lack the knowledge and training to actually create secure products in the short term. However, agile principles such as communication and practices such as pair programming may help disseminate knowledge throughout the development team. Certification remains a potential issue, as it requires detailed and sometimes formal documentation. The approach discussed in Sullivan's article[6] can help by mandating activities that need to be included in every sprint and by requiring a (scope-reduced) final security review at the end of each sprint.

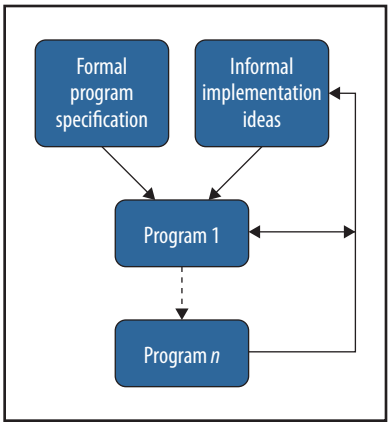

Figure 1. Transformational programming method of algorithm derivation.

EVOLUTION

Real software systems continually evolve. Their evolution is often in response to the evolution of their environments: new functionalities are added and/or existing ones are removed due to the need to address changes in business requirements and/or economic forces, and new platforms are developed and/or new technologies are introduced. This necessitates a paradigm shift in the way software systems are and will be developed, in which agility (without sacrificing trustworthiness) is required.

Software migration, especially assembler migration, is an important type of evolution. More than 70 percent of all business-critical software currently runs on mainframes, and more than 10 percent of all code currently in operation is implemented in assembler, which totals around 140–220 billion lines.

However, the pool of experienced assembler programmers is decreasing rapidly. As a result, there is increasing pressure to move away from assembler, including moving fewer critical systems away from the mainframe platform, so the legacy assembler problem is likely to become increasingly severe.

Analyzing assembler code is significantly more difficult than analyzing high-level language code, so if a large body of assembler code can be replaced by a smaller amount of high-level language code—preserving its correctness without seriously affecting performance—then the potential savings in the form of software and hardware maintenance costs are very large.

A sound agile approach to software evolution that is based on transformation theory and associated with the industrial-strength FermaT program transformation system has been developed. The approach to agility and evolution via transformation involves four stages:

- Translate the assembler to Wide Spectrum Language (WSL).
- Translate and restructure data declarations.
- Apply generic semantics-preserving WSL to WSL transformations.
- Apply task-specific operations for migration (translate the high-level WSL to the target language) and analysis (apply slicing or abstraction operations to the WSL to raise the abstraction level even further).

Martin P. Ward and Hussein Zedan's "Deriving a Slicing Algorithm via FermaT Transformations" fully explains WSL, the transformation theory, and how program slicing can be defined as a transformation within the theory.[7] This approach enables a formal agile methodology to be applied to real software, aiding the software evolution process.

As Figure 1 shows, the transformational programming method of algorithm derivation[8] starts with a formal specification of the result to be achieved, together with some informal ideas as to which techniques will be used in the implementation. The formal specification is then transformed into an implementation by means of correctness-preserving refinement and transformation steps, guided by the informal ideas. The transformation process will typically include the following stages:

- formal specification;
- elaboration of the specification;
- dividing and conquering to handle the general case;
- recursion introduction;
- recursion removal, if an iterative solution is desired; and
- optimization, if required.

Subspecifications can be extracted and transformed separately at any stage in the process. The main difference between this approach and the invariant-based programming approach (and similar stepwise refinement methods) is that loops can be introduced and manipulated while maintaining program correctness with no need to derive loop invariants. Another difference is that at every stage in the process, we're working with a correct program: there's no need for a separate "verification" step. These factors help ensure that the method is capable of scaling up to the development of large, complex software systems.

The algorithm derivation method has been applied to the derivation of the "polynomial addition" problem described by Donald Knuth, which is a complex linked list algorithm that uses four-way linked lists.[9,10] The derived source code turned out to be more than twice as fast as Knuth's code.

This method was also used to derive a polynomial multiplication algorithm using the same data structures. It started with a trivial change to the formal specification (replace "+" with "*"), and the transformational derivation was guided by the same informal ideas as the addition algorithm. The result was an efficient implementation of polynomial multiplication, which worked the first time. Another paper uses this method to derive an implementation of program slicing from the formal definition of slicing, defined as a *program transformation*.[8]

We believe that a combination of formal and agile approaches is a worthwhile goal in the application of software engineering methodologies to real computer-based systems. Of course, a note of caution is in order.[5] The use of heavyweight formal methods with full program proving is likely to be difficult and not worthwhile in a typical agile setting. On the other hand, this is a rare approach, largely due to scaling problems and the difficulty of using the available tools. A lighter touch with the use of formal specification is much more typical when applying formal methods to improve early understanding, guide the programmer, and perhaps aid proper testing. In this context, combined use with agile development is more likely to succeed, if applied judiciously with experienced engineering judgment.

There are additional issues to consider in areas such as safety or security-related systems, but the combination of formal and agile techniques has potential even in these cases. An agile approach is especially applicable in response to evolving systems; for example, where the software may need to be completely upgraded to a new language. By adding a formal foundation, agile evolution can be applied in a sound manner to large software systems with suitable formalized transformation rules.

Acknowledgments

Jonathan Bowen thanks Museophile Limited for financial support. Mike Hinchey acknowledges support from Science Foundation Ireland under grant 10/CE/I1855.

References

1. J.P. Bowen and M.G. Hinchey, "Ten Commandments of Formal Methods…Ten Years Later," *Computer*, vol. 39, no. 1, 2006, pp. 40–48.
2. J.P. Bowen and M.G. Hinchey, "Formal Methods," *Computing Handbook,* 2nd ed., vol. 1, CRC Press, 2014, pp. 1–25.
3. R.M. Hierons et al., "Using Formal Specifications to Support Testing," *ACM Computing Surveys*, vol. 41, no. 2, 2009; doi:10.1145/1459352.1459354.
4. S. Black et al., "Formal Versus Agile: Survival of the Fittest," *Computer*, vol. 42, no. 9, 2009, pp. 37–45.
5. P.G. Larsen, J. Fitzgerald, and S. Wolff, "Are Formal Methods Ready for Agility? A Reality Check," *Proc. 2nd Int'l Workshop Formal Methods and Agile Methods* (FM+AM 10), vol. P-179, 2010, pp. 13–25.
6. B. Sullivan, "Streamline Security Practices for Agile Development," *MSDN Mag.*, vol. 23, no. 12, 2008, pp. 52–55.
7. M.P. Ward and H. Zedan, "Deriving a Slicing Algorithm via FermaT Transformations," *IEEE Trans. Software Eng.*, vol. 37, no. 1, 2011, pp. 24–47.
8. M.P. Ward and H. Zedan, "Provably Correct Derivation of Algorithms using FermaT," *Formal Aspects of Computing*, vol. 26, no. 5, 2013, pp. 993–1031.
9. D. Knuth, *The Art of Computer Programming, Vol. 1: Fundamental Algorithms,* 3rd ed., Addison-Wesley, 1997.
10. M. Ward and H. Zedan, "Provably Correct Derivation of Algorithms Using FermaT," *Formal Aspects of Computing*, vol. 26, no. 5, 2014, pp. 993–1031.

Jonathan P. Bowen *is a professor of computer science at Birmingham City University, UK. Contact him at jonathan.bowen@bcu.ac.uk.*

Mike Hinchey *is director at Lero—the Irish Software Research Centre and a professor of software engineering at the University of Limerick, Ireland. Contact him at mike.hinchey@lero.ie.*

Helge Janicke *is a reader in computer science and head of the Software Technology Research Laboratory at De Montfort University, UK. Contact him at heljanic@dmu.ac.uk.*

Martin Ward *is a reader in the Software Engineering Department at De Montfort University, UK and CTO of Software Migrations Ltd. Contact him at martin@gkc.org.uk.*

Hussein Zedan *is Dean of Graduate Studies & Research at the Applied Science University, Bahrain. Contact him at hussein.zedan@gmail.com.*