Developing High Quality Software with Formal Methods: What Else Is Needed?

Constance Heitmeyer

Naval Research Laboratory, Washington, DC 20375
heitmeyer@itd.nrl.navy.mil

Abstract. In recent years, many formal methods have been proposed for improving software quality. These include new specification and modeling languages, whose purpose is to precisely describe the required software behavior at a high level of abstraction, and formal verification techniques, such as model checking and theorem proving, for mechanically proving or refuting critical properties of the software. Unfortunately, while promising, these methods are rarely used in software practice. This paper describes improvements in languages, specifications and models, code quality, and code verification techniques that could, along with existing formal methods, play a major role in improving software quality.

1 Introduction

During the past two decades, many specification and modeling languages have been introduced whose purpose is to precisely describe the required software behavior at a higher level of abstraction than the code. Examples of these languages include the synchronous languages, such as Lustre [Halbwachs1993]; the design language Statecharts [Harel1987]; requirements languages such as RSML [Heimdahl and Leveson1996], and SCR [Heitmeyer et al.2005]; and design languages offered by industry, such as UML and Stateflow [Mathworks1999], a version of Statecharts included in Matlab's Simulink graphical language.

Specifications and models expressed in these languages can have important advantages in software development. First, they provide a solid basis for both evaluating and improving the software code. Moreover, because they usually exclude design and implementation detail, these specifications and models are more concise than code. As a result, they can serve as an effective medium for customers and developers to communicate precisely about the required software behavior. In addition, they allow both manual and automated analysis of the required software behavior. Analyzing and correcting a software specification is usually cheaper than analyzing and correcting the code itself because the specification is most often smaller—thus finding and correcting bugs in a specification is easier than finding and correcting bugs in code.

Significant advances have also occurred in formal verification. These advances include automated analysis of software or a software artifact using an automated theorem prover such as PVS [Owre et al.1993] or a model checker, such as Spin

F. Wang (Ed.): FORTE 2005, LNCS 3731, pp. 13–19, 2005.

[Holzmann1997] or SMV [McMillan1993]. Such analysis is useful in analyzing a software specification, a model, or software code for critical properties, such as safety and security properties. Because the analysis is largely mechanical, these techniques can be a cost-effective means of either verifying or refuting that a software artifact or software code satisfies a specified property.

Unfortunately, while promising, formal specifications, models and verification techniques are rarely used by most software developers. After reviewing how formally-based tools can help developers improve the quality of both software and software artifacts, this paper describes four enhancements to the software development process which should not only improve software quality directly but should also encourage the use of existing formal methods.

2 On the Role of Tools

Many automated techniques and tools have been developed in recent years to improve the quality of software and to decrease the cost of producing quality software [Heitmeyer2003]. Such tools can play an important role in obtaining high confidence that a software system is correct, i.e., satisfies its requirements. Described below are five different roles that tools can play in improving the quality of both software and software artifacts. The first four help improve the quality of a specification or model. The fifth uses a high-quality specification to construct a set of tests for use in checking and debugging the software code.

2.1 Demonstrate Well-Formedness

A *well-formed* specification is syntactically and type correct, has no circular dependencies, and is *complete* (no required behavior is missing) and *consistent* (no behavior in the specification is ambiguous). Tools, such as NRL's consistency checker [Heitmeyer et al.1996], can automatically detect well-formedness errors.

2.2 Discover Property Violations

In analyzing software or a software artifact for a property, a tool, such as a model checker, can uncover a property violation. By analyzing the counterexample returned by the model checker, a developer may trace the problem to either a flaw in the specification or to one or more missing assumptions. Alternatively, the formulation of the property, rather than the specification, may be incorrect. Detecting and correcting such defects can lead to higher quality specifications and to higher quality code.

2.3 Verify Critical Properties

Either a theorem prover or a model checker may be used to verify that a software artifact, such as a requirements specification or a design specification, satisfies a critical property. Verifying that an artifact satisfies a set of properties can help practitioners develop high confidence that the artifact is correct.

2.4 Validate a Specification

A developer may use a tool, such as a simulator or animator, to check that a formal specification captures the intended software behavior. By running scenarios through a simulator (see, e.g., [Heitmeyer et al.2005]), the user can ensure that the system specification neither omits nor incorrectly specifies the software system requirements.

2.5 Automatically Construct a Suite of Test Cases

Specification-based testing can automatically derive a suite of test cases satisfying some coverage criterion, such as *branch coverage*, from a formal specification [Gargantini and Heitmeyer1999]. Automated test case generation can be enormously useful to software developers because 1) the cost and time needed to automatically construct tests is much lower than the cost and time needed in manually constructing tests, and 2) a suite of test cases mechanically generated from a specification usually checks a wider range of software behaviors than manually generated tests and hence may uncover more software defects.

3 What Else Is Needed?

For practitioners to apply existing formal methods more widely, a number of improvements in the software development process are needed. Described below are four areas where improvements are needed—in specification and modeling languages, in the quality of specifications and models, in the quality of manually generated code, and in improved techniques for software verification.

3.1 Improved Languages

One area that should be revisited is specification and modeling languages. In recent years, researchers have proposed many new languages for specifying and modeling software. Although these languages have been applied effectively in some specialized areas, for example, in control systems for nuclear power plants and in avionics systems, they are still not used widely by software practitioners. While languages introduced by industry, such as UML and Stateflow, are more widely used, they lack a formal semantics. Moreover, the specifications and models that practitioners produce using these languages usually include significant design and implement detail. Given the lack of formal semantics and the large specifications and models that result when design and implementation detail are included, the opportunity to analyze these specifications and models using formally-based tools is severely limited.

Hence, existing languages either need to be enhanced with features (such as fancy graphical interfaces) to encourage practitioners to use them, or new languages need to be invented. One promising approach is to design languages for specialized domains. For example, one or more languages could be designed to specify and model the required behavior of networks and distributed systems. Significantly different specification and modeling languages are likely to

be needed to specify and model the required behavior of software used in automobiles or in avionics systems.

The benefits of using specification and modeling languages with an explicit formal semantics and which minimize implementation detail could be enormous. First, precise, unambiguous specifications can be analyzed automatically for well-formedness, such as syntax and type correctness, consistency (no unwanted non-determinism), and completeness (no missing cases), for critical application properties, such as security and safety properties, and for validity (a check that the specification captures the intended behavior). Specifications that are well-formed, correct with respect to critical application properties, and validated using simulation also provide a solid foundation both for automatic test generation and for generating efficient, provably correct source code.

3.2 Improved Specifications and Models

The specifications and models produced by practitioners (and some researchers) usually include significant design and implementation detail (i.e., are close to the code). Moreover, often they do not use abstraction effectively to remove redundancy and to enhance readabililty. The result is large, hard to understand specifications and models, filled with unnecessary detail and redundancy, which do not distinguish between the required software behavior and implementation detail. In part, this problem can be solved through education. The attributes of good specifications and models and how to construct them are topics that need to be taught and emphasized in software engineering curricula. The problem of poor quality specifications and models can also be ameliorated by improved specification and modeling languages. Such languages should reduce the opportunity for implementation bias and contain mechanisms which encourage the construction of precise, concise, and readable specifications and models. Well-thought out examples of high-quality specifications and models would also help practitioners produce better specifications and models.

3.3 Improved Methods for Building Code

However, improved specification and modeling languages and improved specifications and models are not enough. In the end, what is needed is correct, efficient code. As noted above, an important benefit of a formal specification is that it provides a solid basis for automatically generating provably correct, efficient code. While many techniques have been proposed for constructing source code from specifications, and many software developers use automatic code generators developed by industry, the code produced by these generators is often inefficient and wasteful of memory. Urgently needed are improved, more powerful methods for automatic generation of provably correct, efficient code from specifications.

Another promising approach to producing high quality code is to use "safe" languages, such as Cyclone [Trevor et al.2002]. Using a language such as Cyclone, which is designed to improve the quality of C programs, can reduce code vulnerabilities, such as uninitialized variables and potential sources of buffer

overflows and arithmetic exceptions. A third promising approach to constructing high quality code is to encourage programmers to annotate their code with assertions that a compiler can check at run-time. Hardware designers routinely include such assertions in their designs. Moreover, some C, C++, and Java programmers routinely use assertions as an aid in both detecting and correcting software bugs. Increased use of both safe languages and the annotation of programs with assertions should help improve the quality of software code.

3.4 Improved Methods for Verifying Code

Although improved specifications and models and automatic code generation from high quality specifications can help improve the quality of software, it is highly likely that in the near future, most source code will be generated manually. Urgently needed therefore are improved methods for demonstrating that a manually generated program satisfies critical properties.

One promising approach is to encourage programmers to annotate their code with assertions and to then check those assertions automatically. While current compilers for C, C++, and Java, support and check assertions that annotate the code, the set of assertions that can be analyzed is very limited. Needed are compilers that can not only check simple Boolean inequalities, e.g., $x > 0$, but more complex assertions, e.g., $\texttt{priv}(P, x) = \texttt{R}$, which means that process P has read privileges for variable x. Such assertions can be translated into logic formulae, such as first-order logic formulae, and then a compiler should be able to use decision procedures to check that the code satisfies these logic assertions. In addition to helping practitioners document and detect bugs in their code, such assertions may also be used to prove that the code satisfies critical properties, such as security and safety properties.

Recently, we applied this approach to a software-based cryptographic system called CD (Cryptographic Device) II, the second member of a family of systems, each of which decrypts and encrypts data stored on two or more communication channels [Kirby et al.1999]. An essential property of this system is to enforce *data separation*, that is, to guarantee that data stored on one channel does not influence nor is influenced by data stored on a different channel. Satisfying this property is critical since data stored on one channel may be classified at a different level (e.g., Top Secret) than data stored on another channel (e.g., Unclassified).

A technique which could automatically check code annotated with assertions for a security property such as data separation would be extremely useful. However, rather than directly checking the code for conformance to the security property, an alternative is to construct a high-level formal specification of a system's required behavior, check the high-level specification for the property, and then check that the code is a refinement of the specification. The benefit of the high-level specification is that it describes precisely the set of services that the software is required to support. In checking CD II for data separation, we followed the latter approach: we constructed a high-level specification of the required behavior of CD II, used PVS to prove that the specification satisfied the

data separation property, and then used inspection to show that the CD code, annotated with assertions, satisfied the high-level specification.

More automation of the process we used in verifying that the CD II code satisfies the data separation property would have been extremely useful. Not only would more automation dramatically reduce the human effort need to construct and check the code assertions against both the code and the formal specification, it would also significantly enhance our confidence that the code satisfied the assertions and therefore enforced data separation since manual construction and manual checking of assertions is somewhat error-prone. Two steps were especially labor-intensive: 1) annotating the code with assertions and 2) verifying that the code satisfied the assertions. Also expensive in terms of human effort was the process of demonstrating that the code assertions satisfied the high-level specification.

Adding some annotations to code is straightforward and, as mentioned above, automatic checking of simple code annotations is already supported by many source language compilers. However, generating more complex annotations from code (e.g., constructing inductive invariants from loops) is a problem that requires more research. Checking the conformance of the code with a set of assertions using decision procedures is a promising approach that should be explored by researchers and should help in automating the second step described above. Finally, checking the conformance of a set of validated assertions with a formal specification is also a problem that may require further research.

4 Summary

The improvements described above will require new research in specification and modeling languages, in checking and constructing more complex code assertions, in automatic code generation, and in code verification. They will also require the transfer of existing research, for example, the use of safe languages such as Cyclone, and of formal techniques, such as model checking, theorem proving, and decision procedures, into programming practice. In addition, better educated software developers are needed; such developers will know how to build high quality specifications and models and will routinely include assertions in their code. Finally, existing methods and tools must be better engineered. The result of improved and more automated methods, better educated practitioners, and better engineered tools should allow software practitioners to construct software in an environment in which tools do the tedious analysis and book-keeping and software developers are liberated to transform vague notions of the required software behavior into precise, readable specifications that minimize design and implementation detail and into code that is both provably correct and efficient.

Acknowledgments

Section 3.4, which proposes improved methods for verifying code, benefited from discussions with my NRL colleagues, Michael Colon, Beth Leonard, Myla Archer, and Ramesh Bharadwaj.

References

[Gargantini and Heitmeyer1999] Angelo Gargantini and Constance Heitmeyer. 1999. Using model checking to generate tests from requirements specifications. In *Proceedings, 7th European Software Engineering Conference and 7th ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE-7)*, LNCS 1687, pages 146–162, Toulouse, FR, September. Springer-Verlag.

[Halbwachs1993] Nicolas Halbwachs. 1993. *Synchronous Programming of Reactive Systems*. Kluwer Academic Publishers, Boston, MA.

[Harel1987] David Harel. 1987. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June.

[Heimdahl and Leveson1996] Mats P. E. Heimdahl and Nancy Leveson. 1996. Completeness and consistency in hierarchical state-based requirements. *IEEE Transactions on Software Engineering*, 22(6):363–377, June.

[Heitmeyer et al.1996] C. L. Heitmeyer, R. D. Jeffords, and B. G. Labaw. 1996. Automated consistency checking of requirements specifications. *ACM Transactions on Software Engineering and Methodology*, 5(3):231–261.

[Heitmeyer et al.2005] C. Heitmeyer, M. Archer, R. Bharadwaj, and R. Jeffords. 2005. Tools for constructing requirements specifications: The SCR toolset at the age of ten. *Computer Systems Science and Engineering*, 20(1):19–35, January.

[Heitmeyer2003] Constance Heitmeyer. 2003. Developing high assurance systems: On the role of software tools. In *Proceedings, 22nd Internat. Conf. on Computer Safety, Reliability and Security (SAFECOMP 2003)*, Edinburgh, September. (invited).

[Holzmann1997] G. J. Holzmann. 1997. The model checker Spin. *IEEE Transactions on Software Engineering*, 23(5):279–295, May.

[Kirby et al.1999] J. Kirby, M. Archer, and C. Heitmeyer. 1999. SCR: A practical approach to building a high assurance COMSEC system. In *Proceedings, 15th Annual Computer Security Applications Conference (ACSAC '99)*, pages 109–118, Phoenix, AZ, December. IEEE Computer Society.

[Mathworks1999] The Mathworks Inc. 1999. Stateflow for use with Simulink, User's Guide, Version 2 (Release 11). Natick, MA.

[McMillan1993] K. L. McMillan. 1993. *Symbolic Model Checking*. Kluwer Academic Pub., Englewood Cliffs, NJ.

[Owre et al.1993] Sam Owre, Natarajan Shankar, and John Rushby. 1993. User guide for the PVS specification and verification system (Draft). Technical report, Computer Science Lab, SRI Int'l, Menlo Park, CA.

[Trevor et al.2002] Jim Trevor, Greg Morrisett, Dan Grossman, Michael Hicks, James Cheney, and Yanling Wang. 2002. Cyclone: A safe dialect of C. In *Proceedings, USENIX Annual Technical Conf.*, pages 275–288, Monterey, CA, June.