Available online at www.sciencedirect.com

SCIENCE DIRECT

Electronic Notes in Theoretical Computer Science

Electronic Notes in Theoretical Computer Science 108 (2004) 11–19

www.elsevier.com/locate/entcs

Managing Complexity in Software Development with Formally Based Tools

Constance Heitmeyer[1,2]

Center for High Assurance Computer Systems
Naval Research Laboratory (Code 5546)
Washington, DC 20375

Abstract

Over the past two decades, formal methods researchers have produced a number of powerful software tools designed to detect errors in, and to verify properties of, hardware designs, software systems, and software system artifacts. Mostly used in the past to debug hardware designs, in future years, these tools should help developers improve the quality of software systems. They should be especially useful in developing *high assurance software systems*, where compelling evidence is required that the system satisfies critical properties, such as safety and security. This paper describes the different roles that formally based software tools can play in improving the correctness of software and software artifacts. Such tools can help developers manage complexity by automatically exposing certain classes of software errors and by producing evidence (e.g., mechanically checked proofs, results of executing automatically generated test cases, etc.) that a software system satisfies its requirements. In addition, the tools allow practitioners to focus on development tasks best performed by people—e.g., obtaining and validating requirements and constructing a high-quality requirements specification.

Keywords: formal methods, software tools, formal specification, formal verification, model checking, theorem proving, SCR.

1 Introduction

Over the past decade, our research group at NRL has developed a formal state-machine semantics and a set of formally based tools to support requirements specification in the SCR (Software Cost Reduction) tabular notation [14,15,16,17]. The SCR notation has been used by a number of or-

[1] The author's research is sponsored by the Office of Naval Research.

[2] Email: heitmeyer@itd.nrl.navy.mil

doi:10.1016/j.entcs.2004.11.004

ganizations in industry to develop and analyze requirements specifications of practical systems, including flight control systems, weapons systems, and space systems. For example, in 2001, Lockheed Martin used the SCR notation as well as the SCR tools, together with a test case generator, to detect a critical error described as the "most likely cause" of a $165M failure in the software controlling landing procedures in the Mars Polar Lander [5].

Our target systems are *high assurance software systems*, such as avionics systems, safety-critical software for medical devices, and control systems for nuclear power plants, where compelling evidence is required that the system satisfies a set of critical properties. Among these properties are

- *security properties* (the system prevents the unauthorized disclosure and modification of sensitive information, denial of service, unauthorized intrusions, and other malicious actions),
- *safety properties* (the system prevents unintended events that could result in death, injury, illness, or property damage),
- *fault-tolerant properties* (the system guarantees a certain quality of service despite faults, such as hardware, workload, or environmental anomalies),
- *survivability properties* (the system continues to fulfill its mission in the presence of attacks, accidents, or failures), and
- *real-time properties* (the system delivers its outputs within specified time intervals).

Two high assurance systems currently under investigation by our group are CD, a cryptographic device for use in U. S. Navy systems, and the FPE (Fault Protection Engine), a safety-critical software component of NASA spacecraft. CD is a member of a family of software-based devices that will provide cryptographic processing of data stored on several different channels, each channel associated with a different host system. Because data on different channels may have different security classifications, CD must enforce *data separation*, i.e., ensure that data on one channel cannot influence, nor be influenced by, data on a different channel. We are currently developing a plan for formally specifying and verifying that the CD software, which uses a separation kernel [24] to mediate access to data, enforces data separation.

The FPE is a complex, safety-critical software component of current NASA spacecraft, a version of which will also be used in future spacecraft. The FPE's function is to monitor the health of the spacecraft's software and hardware and to coordinate and track responses to detected faults [8]. Because the FPE's function is crucial to the successful operation of the spacecraft, NASA needs high assurance that the FPE has been correctly implemented. To evaluate the correctness of the FPE implementation, NRL has developed a formal specifi-

cation of the most complex part of the FPE and a suite of test cases, derived automatically from the FPE specification, for evaluating the FPE software. The test cases were constructed using the algorithm described in [9].

Described below are 1) six classes of tools useful in constructing these and other high assurance software systems and components and 2) some areas in which such tools need improvement. In addition, an important aspect of developing a high assurance software system is discussed that is minimally dependent on tool support but is necessary for most tools to be effective. Despite its importance, this aspect—the construction of a high quality specification of the required behavior of a system or software component—has been largely ignored both by software engineering researchers and by software developers.

2 On the Role of Tools

Tools can play an important role in obtaining high confidence that a software system satisfies critical properties. Described below are six different roles that tools can play in improving the quality of both software systems and software system artifacts.

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 [15], can automatically detect well-formedness errors. References [15] and [22] describe how a consistency checker found missing cases and ambiguity in the specifications of both an avionics system and a flight guidance system. In both cases, the checker automatically detected serious errors overlooked by human inspections.

2.2 Discover property violations

In many cases, using a tool, such as a model checker, to analyze a system specification for some critical property uncovers a violation of the property. Given diagnostic information, such as a counterexample returned by the model checker, the developer may find a flaw in the specification or one or more missing assumptions. Alternatively, the formulation of the property, rather than the specification, may be incorrect. In all of these cases, the result of analysis can be extremely valuable. Reference [16] how model checking was used to detect a safety property violation in a contractor specification of a weapons control system. Recently, some researchers have begun using model

checking to detect property violations in software, rather than in software specifications. One notable example is Ball and Rajamani's SLAM project which uses software model checking to detect bugs in device drivers [2]. The result of SLAM's successful use of model checking to detect serious software bugs has led Microsoft to fund the development of a production-quality tool that will use the techniques pioneered in SLAM to detect bugs in device drivers and other similar programs.

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. For example, [21] describes the use of a theorem prover to verify that an early specification of CD satisfies a set of critical security properties.

2.4 Validate a specification

A developer or domain expert may use a tool, such as a simulator or animator, to check that a formal specification captures the intended system behavior. By running scenarios through a simulator, the user can ensure that the system specification neither omits nor incorrectly specifies the system requirements. In developing the FPE specification, for example, simulation was not only extremely valuable in debugging the specification, but also proved useful in obtaining feedback from domain experts about the required behavior, and in demonstrating the behavior captured by the FPE specification to the project sponsors.

2.5 Construct test cases

From a formal specification, a test case generator can automatically derive a suite of test cases satisfying some coverage criterion, such as *branch* coverage [9]. In specifications expressed in either SCR or RSML (Requirements State Machine Language) [12], a requirements language inspired by Statecharts, the value of each dependent variable in the specification is defined by a total function. In branch coverage, each part (i.e., branch) of each of these function definitions forms the basis for constructing a test case. Taken together, the suite of test cases constructed in this manner "cover" every condition (i.e., branch) in the specification. In the FPE project as well as other projects involving high assurance systems, automated test case generation is of high interest to software developers because 1) the cost of automatically constructed tests is much lower than the cost of manually constructed tests,

and 2) a set of test cases that "covers" the specification can provide greater confidence in the correctness of the software than a set of test cases developed in an ad hoc manner.

2.6 Detect coding errors and code vulnerabilities

A static analysis tool can analyze a piece of software without executing it. Such tools can automatically detect errors and vulnerabilities, such as uninitialized variables, erroneous pointers, and arithmetic and buffer overflows, in both source and assembler code. Examples of tools that help detect errors of this class in C code include Codesurfer [6] and Safer C, which finds dangerous vulnerabilities in code, such as those described in [11]. Bishop et al. describe how static analysis tools can help uncover vulnerabilities in the COTS software used in many safety-critical systems [4].

3 Needed Tool Improvements

Although tools can be enormously useful in debugging, and in producing evidence of correctness of, software and software artifacts, a number of tool improvements are urgently needed. These improvements, some previously recommended in [13], are described below.

3.1 Automated Abstraction

Before practical software specifications can be model checked efficiently, the *state explosion* problem must be addressed—i.e., the size of the state space to be analyzed must be reduced. An effective way to reduce state explosion is to apply abstraction. For example, model checking the large specification of a weapons control system [16] did not succeed until two kinds of abstraction were applied. Unfortunately, the most common approach is to develop the abstraction in ad hoc ways—the correspondence between the abstraction and the original specification is based on informal, intuitive arguments. Needed are mathematically sound abstractions that can be constructed automatically. Recent progress in automatically constructing sound abstractions has been reported in [3,16].

3.2 Understandable Feedback

When formal analysis exposes an error, the user should be provided with easy-to-understand feedback useful in correcting the error. Techniques for achieving this in consistency checking already exist (see, e.g., [18]). Although

counterexamples produced by model checkers often provide useful diagnostic information, they are sometimes hard to understand. One promising approach uses a simulator or animator to demonstrate and validate a counterexample.

3.3 Automatically Generated Invariants

Tools, such as the one described in [19], are needed that can automatically construct invariants from a specification. Known invariants have many uses in software development. They can be used as auxiliary lemmas in proving theorems about the software specification. For example, some of the security properties to be proven about an early CD specification [21] could not be proved without auxiliary invariants. These invariants were automatically generated using the algorithms described in [19,20]. Invariants can also be used in validating a requirements specification—domain experts can use automatically generated invariants to determine whether the specification correctly captures certain required system behavior.

3.4 More "Usable" Mechanical Theorem Provers

Although mechanical theorem provers have been used by researchers to verify various algorithms and protocols, they are rarely used in practical software development. For provers to be used more widely, a number of barriers need to be overcome. First, the specification languages provided by the provers must be more natural. Second, the reasoning steps supported by a prover should be closer to the steps produced in a hand proof; current provers support reasoning steps that are at too low and detailed a level. One partial solution to this problem is to build a prover front-end designed to support specification and proofs for a special class of mathematical models. An example of such a front-end is TAME, a "natural" user interface to PVS [25] that is designed to specify and prove properties about automata models [1]. Although using a mechanical prover will still require mathematical maturity and theorem proving skills, making the prover more "natural" and convenient to use should encourage more widespread usage.

4 What Else Is Needed for Effective Use of Tools?

While researchers (and many software developers) usually expend significant effort applying tools, they often exert much less effort and pay much less attention to creating a high-quality system specification. As a result, many current specifications are difficult to understand and change and are also poorly organized. Urgently needed are higher quality specifications of requirements and

software designs. Such specifications are critically important because they serve as a medium for precise communication between the customers, the developers, the verification team, and other stakeholders.

One way to improve the quality of specifications is to choose a "good" specification language. This language must be "natural"; to the extent feasible, a language syntax and semantics familiar to the software practitioner should be supported. The language must also have an explicitly defined formal semantics, and it should scale. Moreover, well thought-out example specifications expressed in the language should be available to practitioners. By studying such examples, practitioners can learn how the language may be used to create specifications that are both concise and easy to understand.

Our group and others (see, e.g., [7,5]) have successfully applied the SCR tabular notation to express the required behavior of a number of software systems and software components. The precise meaning of SCR specifications is given by the state-machine semantics described in [15]. Others, such as Heimdahl and Leveson [12], have proposed a hybrid notation that combines tables and graphics.

Table-based specification languages have many advantages. Not only are tabular specifications easy to understand and (relatively) easy for software practitioners to construct, in addition, tables provide a precise, unambiguous basis for communication among practitioners. They also provide a natural organization which permits independent construction, review, modification, and analysis of smaller parts of a large specification. Finally, tabular notations scale. Evidence of the scalability of tabular specifications was demonstrated in the early 1990s when Lockheed engineers, used a set of tables to specify the complete requirements of the C-130J Flight Program [7], a program containing over 250K lines of Ada code. In addition to tabular notations, other user-friendly notations should be explored. For example, a number of researchers and practitioners capture system requirements using scenarios represented as Message Sequence Charts (MSCs), a notation commonly used to describe communication protocols. Requirements represented by MSCs can be analyzed either directly (see, e.g., [23]) or translated to another representation for analysis as in [26].

Even if a good specification language is chosen, a high quality specification still requires great care and skill on the part of the specifier. Building a good specification is somewhat analogous to designing a good proof. Like a good proof, such a specification should be easy to understand. It should also, for the most part, be free of redundancy, although some planned redundancy is acceptable (e.g., a list of critical system properties). Reduction of redundancy produces a more concise specification, an important attribute of specifications

of large, complex systems. Moreover, the specification should be carefully organized both for ease of understanding and for ease of change. Finally, a good specification should be a reference document, so that information in the specification is easy to find.

5 Conclusion

Tools can be enormously useful in building high assurance software systems. They can find errors that human inspections miss, help validate a specification, provide mechanized support for verifying properties, reduce the time and effort required to construct (and execute) a set of test cases, and provide more confidence in the results of testing by constructing a suite of test cases based on some coverage criterion. Thus, a set of powerful tools can liberate people to do the hard intellectual work required to produce high quality, high assurance software systems. Part of this intellectual effort should be channeled into the acquisition of knowledge of the system and software requirements and into the production of easy-to-understand, well organized requirements specifications.

References

[1] Archer, M, *TAME: Using PVS strategies for special-purpose theorem proving*, Annals of Mathematics and Artificial Intelligence **29** (2000), 139–181.

[2] Ball, T. and S. K. Rajamani, *The SLAM project: Debugging system software via static analysis*, Proceedings, 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2002), ACM SIGPLAN Notices **37**, Portland, OR, ACM Press (2002), 1–3.

[3] Bensalem, S., Y. Lakhnech, and S. Owre, *Computing abstractions of infinite state systems compositionally and automatically*, Proceedings, Computer-Aided Verification, 10th Annual Conference (CAV '98), Vancouver, B.C., Canada, June 28 - July 2, 1998, LNCS **1427** (1998), 319–331.

[4] Bishop, P., R. Bloomfield, T. Clement, S. Guerra, and C. Jones, *Integrity static analysis of COTS/SOUP*, Proceedings, 22nd International Conference on Computer Safety, Reliability, and Security (SAFECOMP 2003), Edinburgh, UK, Sept. 23-26, 2003, LNCS **2788** (2003), 63–76.

[5] Blackburn M., R. Knickerbocker, and R. Kasuda, *Applying the test automaton framework to the Mars Lander Touchdown Monitor*, Proceedings, Lockheed Martin Joint Symposium (2001).

[6] *Codesurfer user guide and technical reference*, Version 1.0, Grammatech (1999).

[7] Faulk, S.R., J. Brackett, P. Ward, and J. Kirby, Jr., *The CoRE method for real-time requirements*, IEEE Software **9** (1992), 22–33.

[8] Feather, M.S., S. Fickas, and N. A. Razermera-Marny, *Model-checking for validation of a Fault Protection System*, Proceedings, 6th International Symposium on High Assurance Systems Engineering (HASE 2001), IEEE Computer Society (2001), 32–41.

[9] Gargantini, A. and C. Heitmeyer, *Using model checking to generate tests from requirements specifications*, Proceedings, ACM 7th European Software Engineering Conference/7th ACM SIGSOFT Symposium on Foundations of Software Engineering, Sept. 1999, Toulouse, FR, LNCS **1687** (1999), 146–162.

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

[11] Hatton, L., "Safer C: Developing software for high-integrity and safety-critical systems," McGraw-Hill, New York, N. Y., 1995.

[12] Heimdahl, M. P. E. and N. G. Leveson, *Completeness and consistency in hierarchical state-based requirements*, IEEE Transactions on Software Engineering **22** (1996), 363–377.

[13] Heitmeyer, C., *On the need for* p*ractical formal methods*, Proceedings, Formal Techniques in Real-Time and Fault-Tolerant Systems, 5th International Symposium, Lyngby, Denmark, Sept. 1998, LNCS **1486** (1998), 18–26.

[14] Heitmeyer, C. L., *Software Cost Reduction*, Encyclopedia of Software Engineering, J. J. Marciniak, ed., 2nd Ed., John Wiley & Sons, Inc., New York, N. Y. (2002), 1374–1380.

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

[16] Heitmeyer, C., J. Kirby, Jr., B. Labaw, M. Archer, and R. Bharadwaj, *Using abstraction and model checking to detect safety violations in requirements specifications*, IEEE Transactions on Software Engineering **24** (1998), 927–948.

[17] Heitmeyer, C., J. Kirby, Jr., B. Labaw, and R. Bharadwaj, *SCR*: A toolset for specifying and analyzing software requirements*, Proceedings, Computer-Aided Verification, 10th Annual Conference (CAV '98), Vancouver, B.C., Canada, June 28 - July 2, 1998, LNCS **1427** (1998), 526–531.

[18] Heitmeyer, C., J. Kirby, Jr., and B. Labaw, *Tools for formal specification, verification, and validation of requirements*, Proceedings, 12th Annual Conference on Computer Assurance (COMPASS '97), June 1997, Gaithersburg, MD (1997).

[19] Jeffords, R. D. and C. L. Heitmeyer, *Automatic generation of state invariants from requirements specifications*, Proceedings, Sixth ACM SIGSOFT International Symposium on Foundations of Software Engineering, Nov. 3-5, 1998, Lake Buena Vista, FL (1998), 56–69.

[20] Jeffords, R.D. and C. L. Heitmeyer, *An algorithm for strengthening state invariants generated from requirements specifications*, Proceedings, Fifth IEEE International Symposium on Requirements Engineering (RE 2001), 27-31 August 2001, Toronto, Canada (2001), 182–193.

[21] Kirby, J., M. Archer, and C. Heitmeyer, *SCR: A practical approach to building a high assurance COMSEC system*, Proceedings, 15th Annual Computer Security Applications Conference, IEEE Computer Society (1999), 109–118.

[22] Miller, S. P., *Specifying the mode logic of a flight guidance system in CoRE and SCR*, Proceedings, 2nd Workshop on Formal Methods in Software Practice (FMSP'98), Clearwater Beach, FL, ACM Press (1998), 44–53.

[23] Peled, D., *A toolset for message sequence charts*, Proceedings, Computer-Aided Verification, 10th Annual Conference (CAV '98), Vancouver, B.C., Canada, June 28 - July 2, 1998, LNCS **1427** (1998), 532–536.

[24] Rushby, J. M., *Design and verification of secure systems*, Proceedings, Eighth Symposium on Operating Systems Principles, 14-16 Dec. 1981, Pacific Grove, CA, Operating System Review **15** (1981), 12–21.

[25] Shankar, N., S. Owre, and J. Rushby, *The PVS Proof Checker: A reference manual*, Computer Science Laboratory, SRI International, Menlo Park, CA (1993).

[26] Uchitel, S., R. Chatley, J. Kramer, and J. Magee. *LTSA-MSC: Tool Support for Behaviour Model Elaboration Using Implied Scenarios*, Proceedings, Ninth International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), Warsaw, April 2003.