Reliable Software Development: Analysis-Aware Design

Gerard J. Holzmann

Laboratory for Reliable Software,
Jet Propulsion Laboratory, California Institute of Technology,
Pasadena, CA 91109, USA
http://lars-lab.jpl.nasa.gov/

Abstract. The application of formal methods in software development does not have to be an *all-or-nothing* proposition. Progress can be made with the introduction of relatively unobtrusive techniques that simplify analysis. This approach is meant replace traditional *analysis-agnostic* coding with an *analysis-aware* style of software development.

Software verification efforts are often stumped by the complexity of not just the analysis itself, but also the preparations that have to be made to enable it. This holds especially if the code base is large, and written in a more traditional programming language with limited builtin protection. When asked to analyze, for instance, the embedded software of an automobile in a high-profile study of the potential software causes for unintended acceleration incidents, our first challenge is generally in the preparations phase. Very similar challenges can exist in the analysis of mission-critical flight software for space missions. Some of the more time-consuming obstacles in these efforts can be avoided, though, if code is designed explicitly with the possibility of independent analysis in mind.

To give a, perhaps overly simple, example of the wisdom or restricting access to shared data in a multi-tasking system: if valuables are stored in the open in a yard, and some are damaged or missing, the analyst will generally have a difficult problem finding out what happened. If they are stored in a locked room and the same thing happens, the job of finding out what happened is reduced to finding out who had access to the key. The analogy to software will be clear: taking even simple precautions can have a large effect.

The adoption of somewhat stronger analysis-aware coding principles can make a notable difference in the types of guarantees that one can give about a large software system, especially when formal methods related tools are used such as static analyzers [10], logic model checkers [6], or provers such as VCC [3]. As one example, reconstructing where state information is stored in a complex system can be one of the hardest obstacles in the application of model-driven verification techniques [7]. The task becomes almost trivial if the global state information is co-located in memory, or placed in a single data structure.

There are many other relatively benign tactics that can be adopted to make code safer and more thoroughly verifiable. As one other example, the integration of complex code with simple aspect-oriented annotations [1] can support

the mechanical generation of code instrumentations that can help a verifier extract design information, or build visualizations of dynamic data structures and message flows that can guide a verification effort.

The benefit of assertions is also well-known [2, 9]. A richer set of assertions can be used though [5, 4]. As an example, two types of inline *temporal assertions* can be used in combination with the FeaVer model extractor [5]: response and precedence assertions. A response assertion, $assert\_r(e)$, expresses the LTL property $\Diamond e$), stating that a condition e must hold within a finite number of steps from the point in the code where the assertion is placed. A precedence assertion, $assert\_p(e1, e2)$, expresses the LTL property ($e1\ U\ e2$), which states that condition $e1$ must hold at least until, within a finite number of steps, condition $e2$ also holds. Temporal assertions can be used to derive property automata for model checkers, as in the FeaVer system [5], or to generate runtime monitors for use in runtime verification, as in the TimeRover system [4].

Finally, in the analysis of complex systems, exhaustive proof will often be beyond reach. This is not necessarily fatal. As Sanjoy Mahajan, a theoretical physicist at Caltech, noted, the attempt to solve complex problems with rigorous methods can lead to *rigor mortis*, which can only be avoided by breaking some of the rules. In our case this can mean using randomized proof techniques and massively-parallel search techniques [8], which can be remarkably effective.

Acknowledgments. The research described in this paper was carried out at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration.

References

1. `http://en.wikipedia.org/wiki/Aspect-oriented_programming`
2. Clarke, L.A., Rosenblum, D., A historical perspective on runtime assertion checking in software development. ACM SIGSOFT Software Eng. Notes, 31:3, May 2006.
3. Cohen, E., Dahlweid, M., et al., VCC: A Practical System for Verifying Concurrent C. Proc. 22nd Int. Conf. on Theorem Proving in Higher Order Logics, 2009, LNCS 5674, Springer Verlag.
4. Drusinsky, D., Temporal Rover. Proc. 7th Spin Workshop, Stanford Univ., LNCS 1885, Springer-Verlag, pp. 323 – 330.
5. Holzmann, G.J., and Smith, M.H., FeaVer 1.0 User Guide, Bell Laboratories Technical Report, 2000, 64 pgs. `http://cm.bell-labs.com/cm/cs/what/modex/`.
6. Holzmann, G.J.: The Spin Model Checker: Primer and Reference Manual. Addison-Wesley, 2004.
7. Holzmann, G.J., Joshi, R., Groce, A., Model driven code checking. Automated Software Eng. Journal, Vol. 15, Nr. 3-4, Dec. 2008, pp. 283-297.
8. Holzmann, G.J., Joshi, R., Groce, A., Swarm Verification Techniques , IEEE Trans. on Software Eng., to appear 2011.
9. Kudrjavets, G., Nagappan, N., Ball, T.,: Assessing the relationship between software assertions and code quality: an empirical investigation. Microsoft Technical Report, `MSR_TR-2006-54` 2006, 17 pgs.
10. `http://spinroot.com/static/`