Tool Intensive Software Development: New Challenges for Verification, Validation, and Certification

Mats P.E. Heimdahl
Department of Computer Science and Engineering, University of Minnesota

As we are moving from a traditional software development process to a new development paradigm where the process it largely driven by tools and automation, new challenges for verification and validation (V&V) emerge. Productivity improvements will in this new paradigm be achieved through reduced emphasis on unit testing of code, increased reliance on automated analysis tools applied in the specification domain, and trustworthy code generation. The V&V effort will now be largely focused on assuring that the *formal specifications are correct* and that the *tools are trustworthy* so we can rely on the results of the analysis and code generation without extensive additional testing of the resulting implementation.

Note here that, in our opinion, the possibility of reducing or fully automating the costly unit-testing efforts are key to the success of this new development paradigm. We have found little support for this type of development if modeling and analysis are to be performed in *addition* to what is currently done—these new techniques must either make current efforts more efficient or replace some currently required V&V activity. In either case, our increased reliance on tools requires that they can be trusted—this poses new challenges for V&V and certification.

Unfortunately, specification execution environments, code generators, and analysis tools are not simple pieces of software and, in our opinion, *it is highly unlikely that we will ever be able to provide the level of confidence necessary to trust a specific tool as a development tool in critical systems development*. Also, consider continual tool evolution and the situation looks increasingly grim. What we need is additional research into alternate means of trusting the results from our tools.

If our conjecture is correct—we will be hard pressed to ever fully trust any individual tool—what other avenues are available? From our experiences there are two directions that seem feasible and acceptable in development practice; (1) redundant prof paths and (2) automated test case generation.

Redundant Proof Paths: The notion of N-version programming has been suggested as one technique to increase our confidence in a software system. The idea is to execute N diverse implementations of the same system in parallel and vote on the results. If the results are not identical, it is assumed that the majority is correct. By applying a similar idea in verification, we *may* be able to raise the confidence in verification results. For example, if we verify a property using the symbolic model checker in NuSMV, the explicit state model checker SPIN, and the theorem prover PVS we may be able to rule out the possibility of erroneously failing to catch a fault. Knight and Leveson have, however, shown that the assumption underlying N-version programming—that the N versions fail independently—is false. Therefore, we do not know how much more confidence we can put in a N versions as opposed to one. In addition, current analysis techniques are not cheap nor easy to use, and applying several redundant techniques will not make it any cheaper.

Clearly, there are challenges in making redundant proof paths a practical reality but it still seems like the most practical solution to raising the level of trust in our analysis to the level required in critical systems development.

Automated Test Case Generation: By using the specification running in its execution environment and the generated code running in its host environment for "back-to-back" testing, we may be able to increase our confidence in both the execution environment as well as the code generator. We would automatically generate test-cases from the specification, execute the test cases on both the specification and the generated implementation, and compare the results. Any discrepancy between the executions indicates a problem with either (1) the specification execution environment, (2) the code generator, or (3) the implementation runtime environment. Although appealing, there are still open questions that need to be addressed before this approach can be adopted. For example: "How do we know when we have tested enough?" Specification and/or code test coverage adequacy criteria suitable for this type of testing must be developed and validated. "How do we generate the (presumably) vast number of tests needed?" Automated techniques to generate test cases to these new criteria must be developed. Nevertheless, recent developments in test-case generation from formal specifications leads us to believe that this will be a feasible solution to part of our V&V and certification problem in the near future.

To summarize, our expected reliance on tools puts increased demands on the V&V of the tools themselves. Here we pointed out the problem and discussed some possible directions—redundant proof paths and automatic test-case generation—that may help us address the problems of V&V and certification of our development tools.

Finally, as we ponder new analysis techniques and development tools, we cannot loose track of one important fact—*although perfection is the goal, perfection is not necessary for deployment.* After all, our aim is to replace costly, time consuming, and error prone manual tasks such as inspections and testing, and all that is really necessary to deploy our tools is that they are *better* than the manual tasks they replace. Unfortunately, we do not know much about how error prone our manual processes really are, nor do we know how to compare the effectiveness of an automated process to a manual process—much important analytical and empirical research is needed to address this issue.