Workshop on Verification, Validation and Testing of Learning Systems in conjunction with the Eighteenth Annual Conference on Neural Information Processing Systems Conference, Whistler, BC, CA, Dec. 16-18, 2004.

Characterizing Software Quality Assurance Methods: Impact on the Verification of Learning Systems

Frederick Sheldon, CSE, ORNL*
Oak Ridge, TN 37831
SheldonFT@ornl.gov

Ali Mili, CCS, NJIT
Newark, NJ 07102
mili@cis.njit.edu

October 20, 2004

1 Introduction: Taking Stock

While learning systems offer great promise in reducing cost and improving quality of control applications [12, 11], they also raise thorny issues in terms of the mismatch between the quality standards that these systems must achieve [10] and the available technology. There is widespread agreement [7, 8, 9, 12, 11] that current verification technology does not apply to online learning (i.e., adaptive) systems, whose function evolves over time, and cannot be inferred from static analysis.

Yet, we claim that one can still use insights from traditional verification technology to better develop verification techniques for adaptive systems. In this short paper, we wish to explore this possibility by:

- Characterizing traditional verification techniques, and using further dimensions of such to propose a classification scheme for assurance (i.e., verification) methods.
- Using the proposed classification scheme to characterize methods that have been or are being developed for online learning systems.
- Perhaps most importantly, showing how the classification/ characterization scheme can be used as a tool to formulate coherent conclusions from an eclectic verification effort (i.e. a verification effort that uses more than one method (e.g., see [5]).

2 A Classification Scheme

We submit the premise that traditional verification methods can be characterized by the following features:

- **Goal**. Property that the method attempts to prove: this can be correctness (e.g., completeness, consistency), but can also be recoverability preservation (an implicit assumption for fault tolerance), or non-functional properties (e.g. reliability, response time).
- **Reference**. Specification against which we are trying to prove the property; very often, we find that what is thought of as different properties are really the same property (correctness) proven against *different* specifications.
- **Assumption**. The condition assumed by the verification method; all verification methods are based on a set of assumptions, and are valid only to the extent that these assumptions hold. For example, testing techniques assume that the testing environment includes (i.e., subsumes) the operating environment; proving techniques assume that the operating environment includes (i.e., subsumes) the assumed semantics of the verification method; fault tolerance techniques assume that the error detection and error recovery logic is fault free, etc.
- **Certainty**. Probability of the claim that is made. Correctness verification claims correctness with probability one; certification testing claims reliability or safety with some lower probability [6].
- **Method**. This dimension includes the well known classification into static (e.g., deductive reasoning, and state-based approaches such as Z, and model checking) versus dynamic (e.g., approaches based on simulation, testing and run-time monitoring); it is possible to imagine others.

This classification is illustrated in table 1. In this table, we show the dimensions of *Goal* and *Reference*; the other dimensions can be used to fill out the entries of the table for each method. The goals are ranked in order of logical implication. The specifications are ranked in partial refinement order: the strongest specification to which we may match a program is the function that it is expected to compute, according to how it is written; the next level is the specification that it was written to satisfy, which is typically much less refined than the expected function; the next level down is a specification of a safety property, typically a minimal condition that we want the program to satisfy even if it is not correct. A test oracle is typically less refined than the requirements specification, and represents the specification against which the program is matched in a test; the sub-test oracle is the restriction of the test oracle to the test data on which the program ran successfully.

*This manuscript has been co-authored by a contractor of the U.S. Government (USG) under DOE Contract DE-AC05-00OR22725. The USG retains a non-exclusive, royalty-free license to publish or reproduce the published form of this contribution, or allow others to do so, for USG Purposes.

Table 1: Two Dimensions: Goals and Specification

Goals / Specifications	Correctness	Recoverability Preservation
Expected Function		
Requirements Specification		
Safety Property		
Test Oracle		
Sub- (Test Oracle)		

To illustrate this table, we consider some well known verification methods or properties and discuss how we see them fit in the proposed classification.

- *Certification Testing*. If we submit program P to a test involving test oracle Ω and test data T and we let Ω' be the restriction of Ω to the test data on which the program execution was successful, the certification test establishes the correctness of the program to the (usually very small) specification Ω'. This conclusion is conditional on the test environment subsuming (being as demanding as or more demanding) than the operating environment.

  Perhaps more significantly, we have also established the probable correctness of the program with respect to specification Ω, conditional on the test data being representative of the overall input domain (usually a doubtful proposition).

- *Static Verification*. This establishes the correctness of the program with respect to the requirements specification, with probability one (at least in principle), conditional on the operating environment subsuming the semantics inherent in the verification method.

- *Reliability*. Establishes the correctness of the program with respect to the requirements specification with some probability p (over some period of time), which can be used to compute such metrics as MTTF by integrating factors such as frequency of invocation.

- *Safety*. Safety is nothing more than correctness with respect to a given safety property (e.g., safety critical programs have the requirement, that no single point of failure will cause (the control program) loss of vehicle/sability).

- *Fault Tolerance*. Fault tolerance is a system's ability to avoid failure after faults have caused errors; it is based on the premise that errors are detected before failure occurs, and is possible only to the extent that recoverability is preserved. Fault tolerance achieves failure avoidance with probability 1, assuming that the fault tolerance infrastructure (code for error detection, error recovery, etc) is trustworthy/ correct.

- *Fail Safety*. Fail safety provides that the program can preserve a safe behavior even when it fails; this requires recoverability preservation with respect to the safety property at hand.

- *Symbolic Execution*. This technique establishes the correctness of a program with respect to its expected function, with probability one, under the same condition as correctness verification.

3 A Calculus of Verification Results

The classification presented above allows us to cast verification results in a uniform manner, by providing the goal, reference, assumption and certainty of each result. If the goal is represented by predicate $\geq$, the reference is represented by specification R, the assumption is represented by predicate A and the certainty is represented by probability p, then the result has the form

$$P(S \geq R|A) = p.$$

We introduce two forms for $\geq$, for the purposes of this discussion:

- Correctness: $S \sqsupseteq R$, i.e. S refines R in the sense of programming calculi [1, 2, 3, 4, 13, 14].

- Recoverability Preservation: $S \supseteq R$, i.e. S preserves recoverability with respect to R.

By introducing this notation we can now derive a wide range of identities, that can be used to compose verification results obtained from heterogeneous methods, with respect to distinct specifications, established with distinct probabilities, etc. These identities are cast in terms of our notation results that stem from probability calculus, programming calculi, refinement semantics, etc. A discussion of these identities is beyond the scope of this paper, though we will present one for the sake of illustration:

$$P(S \sqsupseteq (R_1 \sqcup R_2)|A) = P(S \sqsupseteq R_1|A) \times P(S \sqsupseteq R_2|A),$$

where $(R_1 \sqcup R_2)$ is the join of R_1 and R_2 in the refinement calculus, and we assume that the events $(S \sqsupseteq R_1)$ and $(S \sqsupseteq R_2)$ are statistically independent.

Example of application of this calculus: We have tested a program on test data T using oracle Ω and have established its safety with respect to specification Σ and have established that it preserves recoverability with respect to specification V. Given these preliminary representations, what can we infer overall about the correctness/safety/or other properties of adaptive programs?

4 Verification of Adaptive Systems

The methods available for verifying a program depend on whether the program is deterministic (i.e., traditional) or adaptive. Furthermore, the interpretation of the verification results using such methods, in terms of the notation presented above, depend on the same distinction. However, the identities of the verification results apply equally to any type of program. Consequently, we expect to analyze a range of existing methods for verifying adaptive systems and cast them in terms of our representation to determine if and how our calculus can be used to:

- Cumulate verification results obtained from distinct methods,
- Identify redundancies and complementariness between distinct methods, and
- Infer new verification results from existing results.

References

[1] R.J. Back and J. von Wright. *Refinement Calculus: A Systematic Introduction*. Graduate Texts in Computer Science. Springer Verlag, 1998.

[2] P. Gardiner and C.C. Morgan. Data refinement of predicate transformers. *Theoretical Computer Science*, 87:143–162, 1991.

[3] E.C.R. Hehner. *A Practical Theory of Programming*. Springer-Verlag, 1993.

[4] C.A.R. Hoare and et al. Laws of programming. *Communications of the ACM*, 30(8):672–686, 1987.

[5] H.Y. Kim and F. Sheldon. Testing Software Requirements with Zed and Statecharts Applied to an Embedded Control System, Software Quality Journal, Kluwer, Dordrecht Netherlands, pp. 232-266, Vol.12, Issue 3, Aug. 2004.

[6] A. Mili, F. Sheldon, F. Mili, M. Shereshevsky and J. Desharnais. Perspectives on Redundancy: Applications to Software Certification, IEEE Proc. HICSS, (Testing and Certification of Trustworthy Systems Minitrack), Big Island, HI, Jan. 3-6, 2005.

[7] A. Mili, B. Cukic, Y. Liu, and R. Ben Ayed. Towards the verification and validation of online learning adaptive systems. In Taghi Khoshgoftaar, editor, *Computational Methods in Software Engineering*. Kluwer Scientific Publishing, 2003.

[8] Ali Mili, GuanJie Jiang, Bojan Cukic, Yan Liu, and Rahma Ben Ayed. Towards the verification and validation of online learning systems: General framework and pllications. In *Proceedings, Hawaii International Conference on Systems Sciences*, Big Island, HI, 2004.

[9] Orna Raz. Validation of online artificial neural networks —an informal classification of related approaches. Technical report, NASA Ames Research Ctr, Moffet Field, CA, 2000.

[10] SC-167. Do-178b: Software considerations in airborne systems and equipment cerification. Technical report, Radio Technical Commission for Aeronautics, 1992.

[11] J. Schumann. Vericonn: Verification of controllers based on adaptive neural networks. Technical report, NASA Ames Research Center, Automated Software Engineering Group, 2002.

[12] J. Schumann, P. Gupta, and S. Nelson. On verification and validation of neural network based controllers. In *Proceedings, International Conference on Engineering Applications of Neural Networks*, 2003.

[13] E. Sekerinski. A calculus for predicative programming. In R.S. Bird, C.C. Morgan, and J.C.P. Woodcock, editors, *Mathematics of program construction : second international conference*, number 669 in Lecture Notes in Computer Science, Oxford, UK, 1992. Springer-Verlag.

[14] J. Von Wright. A lattice theoretical basis for program refinement. Technical report, Dept. of Computer Science, Åbo Akademi, Finland, 1990.