Continuous Self-Evaluation for the Self-Improvement of Software

Lori A. Clarke and Leon J. Osterweil*
Department of Computer Science

University of Massachusetts
Amherst, MA 01003
USA

*This work is being carried out as a collaboration among Leon J. Osterwe Clarke and George Avrunin at the University of Massachusetts, Debra Rich University of California at Irvine, and Michael Young at the University

1.0INTRODUCTION:

The purpose of evaluation is to determine the extent to which a softwa meeting its requirements, and to suggest, as much as possible, the sor modifications that should be expected to help improve its ability to m requirements. Our interest is in improving software. Thus in this pap self-evaluation approach, called perpetual testing and analysis, that larger activity of self-improvement.

In a manual, or human-driven, software improvement process humans take t carrying out the testing and evaluation of software, humans infer the ch be made, humans effect the changes, and humans reinstall the modified so which point the improvement cycle begins again. Software self-improvem that some or all of these activities are to be assisted, or entirely car by the software itself, rather than being done solely by humans. There interesting research required in order to reduce the dependence upon hum these activities. In this paper we describe how we propose to transitio responsibility for testing and evaluation of software from humans and on tools and processes.

The essence of our idea is that deployed software should be under contin analysis, and evaluation, drawing heavily upon the computational resourc utilization patterns found in the deployment environment. As the amount possible testing and analysis are virtually limitless, there is value in activities perpetually. It is important, however, that the activities n undirected. We propose that the process of perpetual testing and analys incremental results and findings so as to produce increasingly sharp and It is suggested that these results will inevitably lead to increasingly suggestions about what modifications to the software seem most likely to improvements. Thus, our perpetual testing and analysis proposal is aime

reducing the need for human involvement in evaluation, but also at assist
in proposing improvements.

There are substantial technical challenges in doing this. We propose tha
testing and static analysis go on essentially endlessly, but that results
activities be used to focus and sharpen the other s activity. Research i
make these two complementary evaluation approaches most synergistic is ne
envision orchestrating this synergy through the use of precisely defined
there is considerable research to be done in this area as well.

2.0 APPROACH

While self-modifying code has existed since the earliest days of computir
ample evidence that it poses clear dangers. Of most concern to us is the
modifying code is generally difficult or impossible to analyze, and there
its possible behaviors is difficult or impossible to bound. For this rea
in which software can modify itself, without necessitating the need for c
itself. Our approach to this problem is to view a software product as a
diverse types of software artifacts, including such types of components a
specification, an architecture, low level design specifications, code, te
analysis results. We also suggest that the software product also include
the process by which various components of the software product (code, ir
to be modified. By doing so we can assure that no component of the softw
need modify itself. Rather, a separate component, the modification proce
responsible for modifying other components, such as the code.

Going further, we suggest that the software product is also characterized
constraints that specify the way in which the various components should b
each other. The constraints are particularly important because they are
determine when and how the components of the product need to be modified.
example, when test results derived through execution of testcases are inc
the requirements to which they should be related, this is a signal that p
modification is needed.

The foregoing suggests that a collection of tools should also be consider
the software product. These tools are the devices that are used to help
components of the product and to determine the degree to which the produc
consistent. Thus compilers, design tools, testcase monitors, and static
examples of tools that should be considered to be part of the overall sof

In classical software development, the tools are applied in order to buil
product and its components right up until deployment. But at deployment
code is separated from the rest of the components, and from its constrair
is placed in the deployed environment. This complicates the modificatior
substantially.

We propose that deployed software product code remain tethered to the otl components (including the software product modification process), as well constraint and tool sets that comprise a complete product. By doing this possible for tools to continue to evaluate the consistency of the code wi and to effect modifications. This is the more precise sense in which c suggests how to effect the self-modification of software, without necessi modification of the code component.

Thus the perpetual testing approach implies that software code be perpetu by access to an environment that supports its evaluation and improvement to this as the development environment, even though it will persist past development), and takes a pro-active role in assuring that evaluation cor positive contributions to the improvement of the software. Coordination numerous types of testing and analysis artifacts that exist in the develc deployed environments is a daunting job that is prohibitively expensive a carried out manually. Instead, a highly automated testing and analysis p testing and analysis tools and artifacts, using the available computing r be acquired at any given time. Although this process is considered to be of the product, it is not required to be resident with the code in the de

3.0 TECHNICAL AND RESEARCH CHALLENGES

3.1 DEPLOYMENT CONSIDERATIONS

It is important to emphasize that the nature of the interconnection between the development environment and the deployed code that is (p being testing perpetually will vary considerably, depending upon the variation in deployment situations. Thus, for example, it may be qu to suggest that non-critical prototype research software deployed in research setting may be under continuous evaluation, and continual i with its development environment. On the other hand, it is unthinka that mission critical realtime software code deployed in a secure mi environment will have direct contact with its original development e while it is deployed and in service. There is a large spectrum of c situations between these two extreme situations. It is our belief, can fashion a corresponding spectrum of approaches to supporting the of the benefits of perpetual testing. Certainly the degree to which connect deployed code to the rest of the product will dictate the de we are likely to have with self-evaluation and self-improvement.

Clearly the furnishing of parameterized monitoring capabilities, cor communication between deployment and development environments, and rehosting of major portions of the development environment all pose technical challenges.

3.2 ANALYSIS AND TESTING RESEARCH

3.2.1 INCREMENTAL RETESTING AND REANALYSIS

Testing and analysis techniques should be applicable continuously initial software development, where it should be applied to incom systems, and continuously as software modifications are considere carried out. As modifications are being considered, it is highly to repeat analysis and testing for unchanged aspects of the softw This entails the ability to carry out careful analysis of the imp modifications, determination of which past analytic results remai determination of the most pressing retesting/reanalysis needs, an optimized process for addressing the most pressing needs first.

Incremental reanalysis and testing will require technologies for web of summary information about the code and its various subcomp deciding what parts are partly or fully reusable, and optimizing portions that changes have rendered invalid. Incremental approac been developed for compilation, based on interprocedural data-flo techniques. Incremental techniques developed for testing and ana conceptually similar, but will involve a much more complex set of and relations among multiple sources and kinds of information. T these relations is indicated briefly below.

3.2.2 INTEGRATION OF TECHNIQUES

The prospect of Perpetual Testing suggests that diverse analysis techniques will have to be synergistically integrated and reinteg dynamically and in diverse ways. Previous research has described different analysis techniques could complement one another, but h proposed fixed integration strategies consistent with the assumpt testing was a phase of fixed duration. These activities have tea expensive, but usually less precise, analysis techniques with mor but very precise, techniques, often greatly reducing computation human intervention. For example, static dataflow analysis scans and sharpen dynamic testing regimens, and can be used to iterativ and solve successions of dataflow analyses, that iteratively shar results. These prototype integrations indicate many opportuniti synergy, and the clear feasibility of effecting new integrations

Verification systems have traditionally maintained constraint rel lemmas and theorems, but have had an overly simplistic view that is either proved (when all its relations are proved), or is compl support. Testing systems have maintained records of thoroughness notions of coverage), as well as sets of properties in the form o but have not related levels of assurance to particular properties analysis tools have been limited to a distinction between verifie (must results) and inconclusive (may) results.

Integration of analysis and testing techniques, and particularly post-deployment usage information, provides an opportunity for a set of constraints among properties, techniques, and levels of as example, a property may be verified by a static dataflow analy dependent on the absence of an aliasing relation that is monitore Each asserted property of a software system can be supported by a of analyses and assurances of differing strengths, which are prop through the web of relations and constraints. Monitoring of depl software is critical in this regard, and explicit constraint link monitored in the deployed software provide a way to calibrate and assurances established in the development environment.

3.2.3 SPECIFICATION-BASED ANALYSIS

Testing is a human intensive process. Testers must develop test c test cases and then evaluate the results. The latter task can be consuming, tedious, and error prone. Specification-based testing are being developed so that humans no longer have to play the rol The use of specifications needs to be further developed, not only but for a range of analysis techniques.

Specifications are the driving impetus for a number of analysis a Some modern dataflow analyzers use a quantified form of regular e for expressing properties that are to be validated, others use te specifications, while still others use graphical interval logic. that is actually done and the validity of the system being evalua the specification. Needless to say the quality of the analysis a the quality of the specification. For example, data flow analysis determine the validity of a property unless that property is capt specification.

An advantage of specification-based analysis and testing is that specification base can grow. Over time, as users have more experi the software product, new specifications can be formulated. Ever discovered after deployment should be captured by a new specifica intended behavior so that future modifications of the software pr evaluated against this specification as well. As the specificatio more and more of the reanalysis and retesting will be driven by t should improve the quality of the testing and analysis, but shoul improve the quality of the software product, reduce the amount of intervention, and greatly reduce the time devoted to testing and

3.2.4 PREDICTIVE-DRIVEN ANALYSIS AND TESTING

A perpetual testing framework provides a unique opportunity to ga about the sequence of modifications to the software product and t metrics to predict the most appropriate analysis or testing techn evaluation of subsequent modifications. For example, a code subc that has a history of containing faults might be given a high pri reanalysis if it is modified. Information about the kinds of fau discovered in the past and metrics about the component itself, su uses concurrency or has complicated data structures, would impact of analysis approach to employ. Also, past execution costs could predict computing resources that would be needed to complete the a timely fashion. Finally, if one technique proved to require sub interaction in the past for problems with a similar metric footpr consumptive techniques would be considered.

Recent work on testing and analysis has just started to address u to drive the choice of analysis techniques. With perpetual testi be able to be more complete and effective in gathering informatio software being evaluated, the kinds of faults discovered (or prop verified) by the analysis and testing techniques, the computing r and the amount of human intervention required. On the basis of th information, we should be able to develop a predictive model of w to be the best testing and analysis process. This model would its subject of evaluation and would continue to be modified as we gat experimental evidence. This should lead to a predictive meta-mod incorporated as part of the overall software product. As more inf learned about a particular software product and its modification information would be used by the meta model to evolve a predictiv drive the testing and analysis process for that software.

3.2.5 PERPETUAL TESTING PROCESS DEVELOPMENT

Analysis and testing activities should be viewed as processes to as software is, from requirements through coding, evaluation, and This is particularly important for perpetual testing processes. B to be indefinitely ongoing processes, it is essential that clear beforehand, so that progress towards those goals be continually m and so that revisions to either goals, of processes, or both can continually. Thus, perpetual testing can reasonably be viewed as integration of sequences of applications of both testing and anal demonstrable support of precisely articulated analysis and testin propose to demonstrate the value of articulating requirements for testing processes, and then also demonstrate perpetual testing pr architectural designs.

It seems particularly appropriate to consider, in addition, the d code for analysis and testing processes. For example, dynamic reg testing entails iteration that is comfortably expressed with trad Deciding when and whether to follow coarse-grained, static datafl with more precise, sharpened dataflow analyses is expressible wit case and if-then-else constructs. In both cases, the fine scale products must also be specified.

The testing and analysis process we envisage will also require re mechanisms. It is often important to program immediate reactive to such testing failures as incorrect results and system crashes. analysis results may need to trigger fixed standard subsequent re reporting of analysis results obtained at deployment sites might by timer events, or by the very action of their being completed.

All of this argues for the specification of actual executable pro guide the execution of these perpetual testing processes, and the continuous self-evaluation of software products. In our work we demonstrated such perpetual testing process code, and now propose used as the engine for driving software product self-improvement.

4.0 SUMMARY

The perpetual testing and analysis approach promises to enable the c self-evaluation of software throughout the sequence of modifications during its entire lifetime, and thereby to enable software self-impr can be measured and evaluated. Key to doing this is to perpetually software code to the rest of the overall software product s componen constraints, tools, and the perpetual testing process itself. This increase the confidence that people will have in their software proc software products become larger and more complex, they will become e difficult to evaluate and improve, trust, and predict unless an appr perpetual testing is explored.

5.0 ACKNOWLEDGEMENTS

This research was partially supported by the Air Force Research Labo and the Defense Advanced Research Projects Agency under Contract F3(2-0032, and by U.S. Department of Defense/Army and the Defense Advan Research Projects Agency under Contract DAAH01-00-C-R231, The U.S. Government is authorized to reproduce and distribute reprints for Go purposes notwithstanding any copyright annotation thereon.

The views and conclusions contained herein are those of the authors not be interpreted as necessarily representing the official policies endorsements, either expressed or implied of the Defense Advanced Re Projects Agency, the Air Force Research Laboratory/IFTD, the U.S. De Defense, the U. S. Army, the U.S. Government, the National Science F or of IBM.