AC 2012-3787: EFFECTIVE ACTIVE LEARNING APPROACHES TO TEACHING SOFTWARE VERIFICATION

Dr. Sushil Acharya, Robert Morris University

Sushil Acharya, D.Eng., Associate Professor of software engineering, joined Robert Morris University in the spring of 2005 after serving 15 years in the Software Industry. With U.S. Airways, Acharya was responsible for creating a data warehouse and using advance data mining tools for performance improvement. With i2 Technologies, he worked on i2's Data Mining product "Knowledge Discover Framework" and at CEERD (Thailand), he was the Product Manager of three energy software products (MEDEE-S/ENV, EFOM/ENV and DBA-VOID), which are in use in 26 Asian and seven European countries by both governmental and non-governmental organizations. Acharya has a M.Eng. in computer technology and a D.Eng. in computer science and information management with a concentration in knowledge discovery, both from the Asian Institute of Technology in Thailand. His teaching involvement and research interests are in the areas of software engineering and development (verification and validation) and enterprise resource planning. He also has interest in learning objectives-based education material design and development. Acharya is a co-author of "Discrete Mathematics Applications for Information Systems Professionals," 2nd Ed., Prentice Hall. He is a life member of Nepal Engineering Association and is also a member of ASEE and ACM. Acharya is a recipient of the "Mahendra Vidya Bhusak" a prestigious medal awarded by the Government of Nepal for academic excellence. He is a member of the Program Committee of WMSCI, MEI, CCCT, EEET, ISAS, AG, KGMC, and IMCIC and is also a member of the Editorial Advisory Board of the Journal of Systemics, Cybernetics, and Informatics of the International Institute of Informatics and Systemics.

Dr. Walter W. Schilling Jr., Milwaukee School of Engineering

Walter Schilling is an Assistant Professor in the Software Engineering program at the Milwaukee School of Engineering in Milwaukee, Wis. He received his B.S.E.E. from Ohio Northern University and M.S.E.S. and Ph.D. from the University of Toledo. He worked for Ford Motor Company and Visteon as an embedded software engineer for several years prior to returning for doctoral work. He has spent time at NASA Glenn Research Center in Cleveland, Ohio, and consulted for multiple embedded systems companies in the Midwest. In addition to one U.S. Patent, Schilling has numerous publications in refereed international conferences and other journals. He received the Ohio Space Grant Consortium Doctoral Fellowship, and has received awards from the IEEE Southeastern Michigan and IEEE Toledo Sections. He is a member of IEEE, IEEE Computer Society, and ASEE. At MSOE, he coordinates courses in Software Quality Assurance, Software Verication, Software Engineering Practices, Real Time Systems, and Operating Systems, as well as teaching Embedded Systems Software.

Effective Active Learning Approaches to Teaching Software Verification

Abstract

Course based exercises that map theory to practice with active learning lead to increased understanding and knowledge retention. This is vital within the field of software engineering, for software engineering is an applied discipline. Software Verification is a software engineering process that ensures the software product is being built correctly and is considered to be an industry best practice. Teaching verification to undergraduate students requires additional course material development and preparation in terms of the verification theory to be covered, the domain to be used, and the exercises to be performed, as certain verification topics are time consuming and certain codes are not suitable for undergraduate verification exercises. This paper looks into the approaches taken by two institutions to teach verification using active learning approaches. An overview of the existing literature is provided, including the pros and cons of the approaches. This is followed by an overview of the active learning approaches to software verification from two different universities, and an analysis of each approach. Also included is an assessment of student learning at each institution.

Introduction and Existing Publications

Software verification represents a major aspect of software engineering. Each and every project requires extensive verification activities to avoid software failure. The verification knowledge and skills of a software engineer are important qualities as their appropriate application will ensure confidence among end users on the reliability of the developed software. In the process of training future software engineers, it is important that all graduates receive adequate training in the discipline of software verification.

Despite the importance of testing within the software industry, it receives little attention in the undergraduate curriculum, with the coverage in Computer Science (CS) and Software Engineering (SE) courses assessed as insufficient[1,2]. Technology is rapidly changing and this implies that instructors must instill in CS and SE students the testing skills, methodologies, and knowledge required to meet the challenges of this dynamic industry.

While an understanding of verification has been deemed essential for software engineers, there have been few materials available to educators for assessing software verification at the undergraduate level. The SEI published an "Introduction to Software Verification and Validation"[3], in 1988 which provides some guidance on the curricular coverage required in software verification. The SEEK[4] and SWEBOK[5] also provide high level guidance in what material should be covered in relation to software verification. Other articles have discussed the issues students have with applying testing theory to practice[6] as well as the advantages of active and problem based learning when teaching testing[7,8] as well as integrating debugging into the teaching of testing[9].

Through this paper the authors affiliated with two different institutions share their experiences, success stories, and challenges in using an active learning approach to teaching software verification methodologies and tools.

Institutional Profiles

As this article focuses on two distinctly different institutions and their different approaches to teaching software verification, it is important to understand the differences in the institutions' profiles' and activities. This section provides background information on the two programs' profiles in this article.

Milwaukee School of Engineering (MSOE)

The Milwaukee School of Engineering offers an accredited Bachelors of Science degree in software engineering, and has been accredited since 2002. As an institution, there is a strong emphasis on small class sizes (14:1 student to faculty ratio) and extensive laboratory experience. Students graduating from MSOE spend on average 600 hours in laboratories related to their major. Institutionally, there is more square footage devoted to lab space than lecture hall space. All engineering students are required to complete a three course capstone experience. While the majority of students on campus are in the engineering fields, the school also offers a nursing program, a technical communication program, and several business programs.

MSOE prides itself in having very few traditional computer labs on campus. Instead, all students enrolled in the university are issued a laptop as part of a technology package which includes the laptop and all relevant software needed for the program the student is enrolled in.

The software engineering program offers students several unique learning opportunities. One part of the program is a 9 credit Software Development Laboratory experience where students work on large-scale, industry-sponsored projects. Students are also required to take an application domain sequence of three related, specialized courses which emphasize the application of software engineering material to different domains. Most software engineering courses are offered in the 2+2 format, meaning the course meets in lecture twice for one hour and have a 2 hour associated lab period.

Robert Morris University (RMU)

Robert Morris University offers an accredited Bachelors of Science degree in engineering (Software Engineering concentration), and has been accredited since 2002. Like MSOE emphasis is on small class sizes (10:1 student to faculty ratio) and hands on experiences through class assignments, course projects, internships (150 hours mandatory), and an interdisciplinary capstone project (3 credits). RMU also offers B.S. in Engineering degrees in Mechanical, Industrial, and Biomedical Engineering concentrations and a B.S. in Manufacturing Engineering (the only one of its kind in Western Pennsylvania). A Software Engineering Lab is a part of the engineering departments' Learning Factory (lab space) and is used for software engineering class sizes of 8-10 students. As software engineering is offered as a concentration some of the required

courses and a few of the elective courses are offered through RMU's Computer Information Systems Department (for example courses on programming, data structures, databases, etc.). All software engineering majors take the same set of courses to complete 117 credits out of the required 126 credits. For the remaining 9 credits which are software engineering elective courses students are free to take courses in the areas of their interests like security, databases, etc. All software engineering courses offered by the engineering department are delivered in a 2+2 hours or 1+3 hours format for a total of 4 meeting hours per week. Even though the courses are 3 credits the extra meeting hour is used for additional hands on experiences.

Course Sequence

Milwaukee School of Engineering (MSOE)

At MSOE, the software verification skills and practices are taught very early in the curriculum, starting the fall quarter of sophomore year with a course entitled *Introduction to Software Verification*. Prior to this course, students have completed two quarters of introductory programming using Java and a one quarter introduction to Data Structures course. Students are concurrently taking a *Software Engineering Tools and Practices* course which focuses on the tools and practices necessary for successful software engineering. Topics covered in the tools and practices course include configuration management, automatic code compilation and building, the usage of commercial grade UML tools, GUI design and tools for automating the construction of GUIs, and an introduction to Object Oriented Analysis and Design.

The software Verification course outcomes are shown in Table 1. Notice that while the course is offered at the sophomore level, there are a significant number of outcomes (6, 7, and 8) with a higher Bloom level than might be expected for a sophomore course. This can be accomplished because they rely very heavily on active learning by the students in a lab environment.

Table 1: MSOE Software Verification Course Outcomes

	Bloom's Taxonomy Level	**Outcome Description**
1	Comprehension	Explain why testing is important to software development
2	Application	Compose accurate and detailed defect reports and record defects into a defect tracking system
3	Comprehension	Identify software testing aspects within the presented software development lifecycles
4	Comprehension	Compare and contrast unit testing, integration testing, and system testing
5	Application	Apply black box tests and white box tests to construct a comprehensive software testing strategy.
6	Synthesis	Given a software module description or design (a) Construct a series of unit tests to verify correct operation of the software (b) Implement unit tests to run automatically using JUnit (c) Construct mock objects for classes which may need to be simulated (d) Analyze the results of the unit tests
7	Evaluation	Evaluate the effectiveness of software test cases with mutation testing.
8	Evaluation	Critique a given piece of source code for complexity and testability.

Lecture Coverage

Lecture material for the course begins with an analysis of software failure. In this introductory lecture, students are provided with case studies of software failure and, when possible specific details (including software code) detailing the root cause for the software failure. Students are then shown how different testing approaches might have caught the software problem before a field failure occurred. While this only encompasses one lecture in class, it serves to introduce the students to the need for software verification.

Before going in depth into software testing techniques, students need an early introduction to the software development lifecycle models. At this point in time, there has been no formalized explanation of the phases within a software development project. Thus, to even provide a rudimentary understanding of the different types of testing requires an explanation of the fundamental models of software development.

With this fundamental introductory material presented, it is now possible to move into the realm of unit testing and the discussion of the design of unit tests. Students are provided with the core definitions for testing, the relationship between errors, faults, and failures, and the concept that software verification may be viewed as the only step in software development which is destructive rather than constructive. Black box versus white box terminology is also covered.

To build tests, students need some basic theoretical underpinnings to use for selecting test cases. This comes from an explanation of equivalence class testing and boundary value analysis. Both are briefly introduced in lecture through simple active learning exercises. Simultaneously, students are also exposed to test automation through an overview of the JUnit tool.

While equivalence classes and boundary value analysis are clear mechanisms that can be used for certain types of testing, many other tests require dynamic behavior which cannot easily be modeled using equivalence classes or boundary value analysis. For these problems, the concept of a formal state model and state transition testing is introduced.

While students do not have significant problems understanding the basic concepts needed for testing, they often struggle with how to implement the test cases. It is for this reason that the concept of a Mock Object is introduced following state transition testing. In class, students are shown how a Mock object can be used to test in its entirety a simple application without user intervention or a complete UI.

Up until this point, all testing techniques have been black boxed in nature. Students are introduced to white box testing, control flow graphs and cyclomatic complexity in lecture and shown how these tools can be used to assess how complex a given implementation is going to be to adequately test.

To conclude the course, other assorted topics are presented. This includes the topic of exploratory testing. Through an active learning exercise, students learn approaches to performing unscripted exploratory testing on a simple calculator application. This then leads to the concept of mutation testing, which is introduced as a mechanism to quantify the quality of automated test suites. Lastly, students are provided with lecture material on the relationship

between design, implementation and testability. This leads students to have an understanding of how complex a given implementation is going to be to test.

Lab Coverage

While lecture represents a great medium for introducing topics to students, for many students, a more active approach with a larger problem than can be solved in lecture is necessary for the students to completely comprehend the material. Thus, the verification course includes a lab dealing with testing. One of the highlights of this lab is a strong focus on reading of code written by other engineers. In the majority of the labs, students are provided with code which has been implemented by another programmer and needs to be tested. This serves two purposes. First, it forces students to read the specifications very carefully. Second, and most importantly, it teaches students how to go about debugging code developed by another engineer. Both of these are valuable skills within the software engineering field.

At the beginning of the course, students are just returning from summer break. Thus, the first lab deals with a return to software development as well as an introduction to testing. In this lab, students are given a simple problem: design and implement a program which will read a file of lines describing triangles. For each triangle, print out whether it is isosceles, scalene, equilateral, or invalid, as well as the area and perimeter of the triangle. Students are told that they should carefully define a test file which tests all necessary conditions to verify the correct operation of their program before developing the program, and these files are reviewed by the instructor in lab. In most cases, the students are overly optimistic in defining their test cases, selecting only the simplest cases which only test the valid conditions.

The second lab sequence involves a banking program and the definition of test cases based upon a set of use cases. During the first week of this two week lab sequence, students read and review a set of use case scenarios for a software system which manages a simple bank. From the use case scenarios, students then develop test cases that will be used during the second week. The students are not given any specific guidance as to how to approach this task aside from using an organized approach to defining the test cases. This is done on purpose, for the students often learn the importance of an organized approach by not using an organized approach the first time through. During the second week of lab, students are given an executable image of the program and are asked to log the defects found in a bug tracking system.

Overall, this lab serves multiple purposes. Through active learning, it forces students to think in an organized fashion. Those who do not approach the development of test cases from the use cases in a logical manner often have great difficulty executing the tests in a meaningful manner. It also teaches the students how much effort and attention to detail is required to manually execute a large test plan. And lastly, it provides the students with a mechanism to learn how to write bug reports, helping them to become more effective communicators.

With the banking program tested, and several lectures on testing theory being presented, the sequence moves into verifying the implementation of a program which aids in filing of one's taxes. In this case, students are provided with a simplified version of the 2008 US tax code and an implementation of a program which calculates the tax due and net tax rate for a filer. Using boundary value analysis and equivalence classes, students are required to write a sequence of test

cases which will verify that the code correctly calculates the tax due and net tax rate. The code itself has been injected with bugs, and the students are required to fix these bugs as they are uncovered. From this lab, students gain a practical understanding of how to automate tests with JUnit as well as the interplay between boundary values and equivalence classes when dealing with multiple variables in a calculation.

While the tax calculator is complex in that it has multiple variables being used in its calculations, it is written in a very functional manner. This is not, however, how most programs are implemented. Many software programs involve complex sequential interactions between classes which cannot be tested with this simple program. For the next lab, students move onto testing a program which monitors a stock portfolio and provides the user with feedback as to its state. To adequately test the logic of this program, students must develop scenarios which show a rise in a security, a fall in a security, and a stable security over time as well as the interactions with classes which control playing music and logging results. This requires students to use a mock object in place of a web proxy as well as in place of the audio playback module. Students use a tool to develop the mock objects based on a UML design provided for the system. As with the previous systems, bugs have been strategically injected into the source code, and the students are responsible for fixing them.

The mock object lab then leads into a lab on state based testing. In this lab, a simple light controller is tested. The design for the light controller is expressed in the form of a state machine, and students are responsible for developing adequate test cases to exercise the light controller. To further emphasize mock objects, students are also required to perform the testing using a mock object.

In lecture, the class has now reached white box testing. Thus, to experimentally explain white box testing, the lab returns to the tax calculator project. Students are asked to measure the coverage of their initial test cases on the implementation. After doing this, students are then instructed to add additional test cases until an appropriate level of code coverage is obtained. A different implementation of the tax calculator is then given to students for testing. This implementation has been optimized to show students problematic areas for code coverage. Students are to measure the coverage of their test cases on this code as well as add test cases to improve this implementation.

This leads into the third and final exercise involving the tax calculator. In this lab session, students are to use a mutation tool to perform mutation testing on the tax calculator and their test cases. A third variation of the tax calculator is provided to the students in which the implementation has been tuned to have stubborn and equivalent mutant present. Students are to add test cases until all mutants are either killed or determined to be stubborn. Students see the impact of additional test cases on their mutation score as well as the impossibility of killing off every mutation instance.

The final lab involves the analysis and synthesis of testing complexity. Students are provided with a tool which calculates the cyclomatic complexity of compiled jar files as well as other relevant metrics. From this, students are asked to estimate the complexity of testing this code.

Students are then given the code for several of the projects to confirm their suspicions. While seemingly short, this lab is designed to be completed in the final week of the quarter in lab.

Other Coverage

In addition to the software verification course, students at MSOE receive additional practice in software verification in other courses. Software Engineering Process continues the skills developed in the introductory course while teaching students to effectively estimate and manage a personal process. In this particular course, students are required to create regression test suites as part of their testing activities.

Students also practice software verification skills in the 3 course Software Development Laboratory experience. In this course sequence, students work on large scale teams to develop software solutions for external entities. In completing this course, students are also expected to demonstrate the usage of appropriate verification techniques.

Lastly, as seniors, students are required to take Software Quality Assurance. This course focuses on the broad aspect of software quality. Software verification is included in this course as well, with the focus being on determining if the verification activities conducted have been effective towards the delivery of a quality software product.

Robert Morris University (RMU)

At RMU a *Software Verification and Validation*[10] course is offered to software engineering students at the junior level. The course provides an in-depth treatment of Software Quality Assurance and introduces methods and techniques in software verification and validation including reviews, requirements engineering, configuration management, and software testing. The outcomes measured for these courses are in line with ABET outcomes listed in Table 2.

Table 2: Measured ABET Outcomes

Outcome 1: An ability to apply knowledge of mathematics, science and engineering.
Outcome 2: An ability to design and conduct experiments, as well as to analyze and interpret data.
Outcome 3: An ability to design a system, component or process to meet desired needs.
Outcome 4: An ability to function on multi-disciplinary teams.
Outcome 5: An ability to identify, formulate, and solve engineering problems.
Outcome 6: An understanding of professional and ethical responsibilities.
Outcome 7: An ability to communicate effectively.
Outcome 8: The broad education necessary to understand the impact of engineering solutions in a global and societal context.
Outcome 9: Recognition of the need for and an ability to engage in life-long learning.
Outcome 11: An ability to use the techniques, skills, and modern engineering tools necessary for engineering practice.
Outcome S1: The ability to analyze, design, verify, validate, implement, apply and maintain software systems.
Outcome S2: The ability to appropriately apply discrete mathematics, probability and statistics and relevant topics in computer science and supporting disciplines to complex software systems.

Lecture Coverage

Software engineering education starts from the fall term of the freshman year when all students are required to take an introductory core course on decision support systems. For the next three terms students take programming courses (C++ and Java), a course on discrete mathematics, and a course on computer architecture. These courses prepare the students for a course on fundamentals of software engineering offered at the fall term of the junior year. The course description of this course "*ENGR3410 – Fundamentals of Software Engineering*" is as following:

> *This course is an introduction to the discipline of Software Engineering. The role and function of the software engineer are introduced in the context of liability issues and of the software conceptualization development, verification and validation and implementation environments. Software engineering methods, techniques and algorithms are introduced. Hands-on exercises on Software development starting from inception of software projects to its implementation are delivered. The product life cycle is approached on a block-by-block basis. UML is used as the S/W engineering modeling language.*

A major achievement of this course is an understanding of software engineering fundamentals and best practices as well as management processes. At the end of this course students deliver a fully operational student initiated software product[11]. This product is a feeder to the verification component of the verification and validation course the student will take in the spring term of the junior year. *ENGR4300 – Software Verification and Validation* thrives to enhance student skills in the following areas:

i. **Communication Skills:** Students gain experience in technical communication skills through collaborative learning, role-plays (videos and exercises), and technical presentations. This is consistent with IEEE/ACM Curriculum Guideline # 8[12].
ii. **Applied Knowledge of Methods:** Students use V&V methods in the lecture and hands-on sessions. With case-studies, role-plays (videos and exercises), hands-on exercises, observing practitioners at work, and expert lecture sessions, students gain the ability to translate theory to practice. Students also create a set of test cases to perform black box testing. This is consistent with IEEE/ACM Curriculum Guideline # 4[12].
iii. **Applied Knowledge of Tools:** Students use V&V Tools. Hands-on activities give experience to students on the use of V&V tools for *requirements management, configuration management, and defect management*. This is consistent with IEEE/ACM Curriculum Guideline # 12[12].
iv. **Research Exposure:** Students learn research skills. Students submit research assignments, and participate in research discussions on V&V related topics. These skills enable students to learn how to remain current with industry best practices and to take educated decisions in using them.

The week-by-week lecture content for this course is listed in Table 3 below.

Hands-on Assignments

The 4 hour meeting time for the *Software Verification and Validation* course is roughly divided into one and a half hours lecture and 2 and half hours hands-on assignments. The hands-on assignments are where active learning takes place. The focus in the hands-on assignments is on improving student skills in the areas listed earlier. Active learning involves class discussions, watching/analyzing role play videos, collaborative learning sessions, and short exercises. The final seven weeks of the course focuses on software verification. Through these weeks students are introduced to software testing and are required to complete a project on software testing.

Table 3: RMU V&V Lecture Content by Week

Week	Topic (s)
W1	Relationship of Software V&V to software development, Historical Perspective of V&V, S/W Quality, Software Quality Assurance
W2	Quality Methods in other Industries, Overview of Software Development Lifecycle Models, Formal Proof of Correctness, Clean-room Process for Software Development
W3	Requirements, Overview of Software Verification
W4	Verification Activities, Formal Inspection Process.
W5	Applying the Formal Inspection Process
W6	Configuration Management, Defect Management
W7	Overview of Software Validation
W8	Overview of Software Validation
W10	Software Testing
W11	Software Testing
W12	Predictable Software Development
W13	Software Reliability Modeling
W14	Software Standards
W15	Miscellaneous Topics in V&V

Students work on both Black Box and White box testing. Basis Path Testing is used to perform White Box Testing. Testing is based on Cyclomatic Complexity (software metric that provides a quantitative measure of the logical complexity of a program). Students are also exposed to Control Structure Testing and Platform Testing. An application software project completed by the students in a prerequisite course (ex. at RMU: *ENGR3410: Fundamentals of Software Engineering*) is used for the project on black box testing for this course.

The week-by-week hands-on assignment for this course is listed in Table 4.

Black Box Testing Project

This is an individual project on software testing dealing with a software product students have developed as part of their courses at RMU or at other institutes. Students need to have the source code of the software product for this project. Student begin with preparing a ‘plan of action’ and a ‘project plan (Gantt chart)’. These need to be approved by the professor before students are

allowed to begin project work-. The project plan assists the students in properly using their time to meet the goals of the project with the given timelines. The deliverables for this project are a test report that consists of requirements refinement, test cases design, test cases development, and test case execution. Students will need to create 50 test cases covering a wide range of areas that they will learn as they progress through the course. At times they receive JIT (just in time) knowledge to work on their project. For the project progress evaluation students present a progress review. For the final evaluation students will make final presentation and submit a complete test report containing the following:

1. Title page
2. TOC
3. Introduction
4. Test Plan
5. Requirements
6. Refined Requirements
7. Test Outline
8. Test Outline Iterations
9. Test Execution Summary
10. Test Cases (*minimum 50*)
11. Conclusion
12. Appendices

Table 4: RMU V&V Hands-on Assignments by Week

Week	Topic (s)
W1	Assignment 1-1: Business paper analysis (Legal issues) Assignment 1-2: Business paper analysis (Consumer protection)
W2	Assignment 2-1: Research paper analysis(Bugs) Assignment 2-2: Role Play – Manager Director - Justification for budget allocation for V&V activities. Assignment 2-3: Exercise on Deming's 14 Points - System of Profound Knowledge (SoPK).
W3	Assignment 3-1: Article Read & Discuss (Case-study) Assignment 3-2: Article Read & Discuss (Requirements from the Customer Perspective) Assignment 3-3: Exercise on Requirements (Rewriting Requirements). Assignment 3-4: Article Read & Discuss (Understanding User Requirements)
W4	Assignment 4-1: Article Read & Discuss (Issues with Post Delivery Maintenance). Assignment 4-2: Exercises on Inspections. Assignment 4-3: Exercise on Pair Programming Assignment 4-4: Exercises on Reviews.
W5	Assignment 5-1: DVD- Scenes of Software Inspection (SEI production).
W6	Assignment 6-1: Team exercise on Configuration Management
W7	Assignment 7-1: Team exercise on Defect Management (WIP)
W8	Assignment 8-1: Understanding the SRS Document (Exercise WIP) Assignment 8-2: Project related exercise – Requirements Definition.
W10	Assignment 10-1: Project related exercise - Requirements Refinement. Assignment 10-2: Project Progress Presentation.
W11	Assignment 11-1: Project related exercise - Test Outline. Assignment 11-2: Project Progress Update. (email)
W12	Assignment 12-1: Project related exercise - Designing Test Cases. Assignment 12-2: Project Progress Presentation.
W13	Assignment 13-1: Project related exercise - Executing Test Cases. Assignment 13-2: Project Progress Update. (email)
W14	Assignment 14-1: Project related exercise – Preparing Test Report. Assignment 14-2: Project Progress Update (email)
W15	Assignment 15-1: Project Final Presentation

Other Coverage

In their senior year besides the three electives of their choice the students are required to take a course on distributed systems and an interdisciplinary capstone course. Students are required to use their knowledge and experiences in software verification when they work on software projects in these courses.

In the *Distributed Systems* course students take the working software they developed in the *Fundamentals of Software Engineering* course and make changes so that the software can function in a distributed environment. Once the project is complete what was earlier a stand-alone project now becomes a client server or web based system. Before the students incorporate the changes to their code students define benchmarks for performance, reliability, and real-time and test these against the completed software. During the code change students prepare unit test cases and execute them against completed code components thus effectively utilizing the knowledge gained in the software verification and validation course.

In the *Integrated Engineering Design* (interdisciplinary capstone) course students work with students from manufacturing, mechanical, industrial, and biomedical disciplines to solve engineering problems presented by real customers. In these problems software engineering students' work on the software component where they are expected to effectively use knowledge gained from their software engineering core and elective courses. Use of verification knowledge is very important in this course. This course is where the students are able to showcase their mastery of software engineering knowledge to the customers which has in multiple situations led to full time employment for the students. Students have successfully implemented solutions for companies like FedEx Ground and Data Dimension International.

Assessment of Effectiveness

Milwaukee School of Engineering (MSOE)

There are many ways to assess teaching effectiveness. For the software verification course, two different mechanisms of assessment have been used to track educational effectiveness of the lab sequence. The first mechanism is a course survey given to all students at the end of the course. This survey asks students for each lab if the lab was an excellent tool for teaching the level of testing needed at that time in the course, whether the student learned a lot from completing the lab, and if the lab should be continued for future students. These questions are answered using a 5 point Likert scale, where 5 represents "Strongly Agree", 4 represents "Agree", 3 represents "Neutral", 2 represents "disagree" and 1 represents "Strongly disagree".

Table 5 depicts the survey results for the labs. In general, we see from the results, that the students feel the lab exercises they have completed should be continued for future students. In general, the survey also reflects that the students believed the lab was an excellent tool for teaching the material given the level of expertise shown at that time.

A second and possibly more important mechanism for assessing student's achievement in the area of testing is to look at targeted assessments on the final exam for the course. While there are many different types of questions given on the final exam and questions addressing different

levels of Bloom's taxonomy, each exam has always provided the students with an open ended problem dealing with the construction of verification tests. Based on the problem description, students are tasked with designing and implementing a JUnit test suite which fully tests the given problem. This problem is consistently graded using a 20 point grading rubric and is used for performance assessment. Based on student's achievement on this question, students are ranked as either Exceptional, competent, marginal, or inadequate. Table 6 provides assessment data for each of the 3 years that this type of comprehensive question has been asked on the final exam. Overall, in each case, the majority of students are capable of performing the given task.

Table 5: Survey Assessment Results for Labs

		2008				2009				2010				2011						
		Average	Median	Stdev	4/5 percent	Average	Median	Stdev	4/5 percent	Average	Median	Stdev	4/5 percent	Average	Median	Stdev	4/5 percent	Overall Average	Overall Median	Overall 4/5 percentage
	Sample Size	10				12				33				36				91		
Triangle Analysis	This lab was an excellent tool fo teeaching the level of testing I needed at the time.	4.30	4.00	0.48	100%	3.50	4.00	1.09	67%	3.82	4.00	0.85	0.67	4.11	4.00	0.89	86%	3.95	4.00	78%
	I learned a lot from this lab.	3.70	4.00	0.95	60%	2.75	3.00	1.06	25%	3.33	3.00	0.99	45%	3.83	4.00	1.13	64%	3.49	3.51	52%
	This lab should be continued for future students	4.10	4.00	0.32	100%	3.83	4.00	0.83	75%	3.91	4.00	0.77	73%	4.25	4.00	0.91	83%	4.05	4.00	80%
Testing a Banking System	This lab was an excellent tool fo teeaching the level of testing I needed at the time.	4.20	4.00	0.42	100%	3.75	4.00	0.62	67%	4.24	4.00	0.61	91%	4.28	4.00	0.70	92%	4.19	4.00	89%
	I learned a lot from this lab.	3.80	4.00	0.79	80%	3.42	3.50	0.90	50%	3.91	4.00	0.95	73%	4.11	4.00	0.85	75%	3.91	3.93	71%
	This lab should be continued for future students	3.70	4.00	0.67	60%	3.67	4.00	0.78	67%	4.12	4.00	0.65	85%	4.25	4.00	0.73	89%	4.07	4.00	81%
The Tax Code	This lab was an excellent tool fo teeaching the level of testing I needed at the time.					4.25	4.00	0.62	92%	4.33	4.00	0.60	94%	4.42	4.00	0.65	97%	4.36	4.00	85%
	I learned a lot from this lab.					4.00	4.00	0.74	75%	4.24	4.00	0.66	88%	4.33	5.00	0.93	89%	4.25	4.44	77%
	This lab should be continued for future students					4.25	4.00	0.45	100%	4.21	4.00	0.65	94%	4.39	4.50	0.80	94%	4.30	4.22	85%
Mock Objects	This lab was an excellent tool fo teeaching the level of testing I needed at the time.	3.60	4.00	1.07	60%	3.17	3.50	1.27	50%	3.52	4.00	1.15	55%	4.00	4.00	1.00	72%	3.67	3.93	62%
	I learned a lot from this lab.	4.00	4.00	0.82	70%	3.17	3.00	1.11	42%	3.70	4.00	0.92	58%	4.17	4.00	0.86	75%	3.85	3.87	64%
	This lab should be continued for future students	4.00	4.00	0.82	70%	3.08	3.50	1.16	50%	3.58	4.00	1.06	55%	4.09	4.00	0.89	69%	3.76	3.93	62%
State Based Testing	This lab was an excellent tool fo teeaching the level of testing I needed at the time.													4.00	4.00	1.00	72%	4.00	4.00	72%
	I learned a lot from this lab.													4.17	4.00	0.86	75%	4.17	4.00	75%
	This lab should be continued for future students													4.09	4.00	0.89	69%	4.09	4.00	69%
Code Coverage	This lab was an excellent tool fo teeaching the level of testing I needed at the time.	4.30	4.50	0.82	80%	3.75	4.00	0.75	75%	4.30	4.00	0.73	85%	4.31	4.00	0.76	86%	4.23	4.05	84%
	I learned a lot from this lab.	4.10	4.00	0.74	80%	3.83	4.00	1.11	75%	4.00	4.00	1.00	73%	4.09	4.00	0.85	81%	4.02	4.00	77%
	This lab should be continued for future students	4.50	4.50	0.53	100%	3.83	4.00	0.72	83%	4.21	4.00	0.65	88%	4.23	4.00	0.81	81%	4.20	4.05	86%
Mutation Testing	This lab was an excellent tool fo teeaching the level of testing I needed at the time.					2.92	3.00	1.00	33%	4.21	4.00	0.65	88%	3.94	4.00	0.97	72%	3.90	3.85	65%
	I learned a lot from this lab.					3.00	3.00	0.74	25%	4.06	4.00	0.90	76%	4.00	4.00	1.11	78%	3.88	3.85	62%
	This lab should be continued for future students					2.83	3.00	1.11	33%	4.09	4.00	0.80	79%	4.09	4.00	0.92	78%	3.90	3.85	64%
Design Complexity	This lab was an excellent tool fo teeaching the level of testing I needed at the time.	4.20	4.00	0.79	80%	4.08	4.00	1.08	83%					3.91	4.00	0.98	67%	4.00	4.00	46%
	I learned a lot from this lab.	4.20	4.00	0.79	80%	3.83	4.00	0.94	67%					3.71	4.00	0.96	56%	3.82	4.00	40%
	This lab should be continued for future students	4.20	4.00	0.63	90%	4.50	5.00	0.67	92%					3.94	4.00	0.76	67%	4.10	4.21	48%

Note: Not all labs all labs were conducted in each year the course was taught. Empty entries either reflect that the specific lab did not occur or that the lab occurred after the survey data was collected.

Table 6: Performance Assessment for students Designing and Implementing a Test Solution on the Final Exam

Categorization	Definition	2011	2010	2009
Exceptional	Score of 19 points of higher on the problem.	0	17%	27%
Competent	Score of 15 points or higher on the problem.	84%	75%	53%
Marginal	Score of 11 points or higher on the problem.	0	4%	13%
Inadequate	Score of less than 11 points on the problem.	16%	4%	7%

Robert Morris University (RMU)

The *Software Verification and Validation* course is focused on improving the knowledge and skills of undergraduate students (i.e. student retention of V&V knowledge and skills*)*. For this course both learning outcome assessments and course evaluations are carried out.

The following assessments are used to assess educational effectiveness (learning outcomes):

1. **Student Confidence in Methods and Tools:** Students demonstrate V&V knowledge and skills by an improvement in their ability to perform hands-on exercises. To assess student communication skills students are required to make 2 project presentations for which they are graded in the following six areas: *content (50%), organization/structure (20%), style/presentation/appearance (8%), use of visual aids (10%), audience participation (10%)* and *adherence to time limit (2%).* To assess applied knowledge of methods and applied knowledge of tools, hands-on laboratory exercises are graded. To assess research exposure, research assignments and the use of IEEE document standards in project deliverables are assessed.

2. **Test Performance:** Students demonstrate V&V knowledge and skills by an improvement on the pre-test/post-test instrument. The mid-term and final exams performance also supplements this measurement. The mid-term exam accounts for 15% and the final exam accounts for 15% of the course grade. Both exams have two sections: a set of multiple choice questions and a set of essay type questions. The exams are administered using a closed-book format for the former and open-book format for the latter. More than 50% of the exam questions are in the areas covered by hands-on assignments. Figure 1 depicts student grades for Year 1 and Year 2. In Year 1 25% of the students received an A grade whereas in Year 2 43% of the students received an A grade. It was observed that majority of the answers were better formulated and justified indicating good understanding of the subject matter.

The following assessments are used to evaluate course contents and delivery effectiveness:

1. **Faculty Course Analysis Report (FCAR):** For the Faculty Course Analysis Report (FCAR), ABET Derived Outcomes: 1, 2, 4, 5, 6, 7, 8, 9, 10, 11 and ABET Software Engineering Track Outcome: S1 (refer to Table 2) are assessed after the completion of the course. Student outcome assessments are based on *analysis of examination questions,*

research paper reviews, classroom discussions, hands-on exercises, and *project deliverables.*

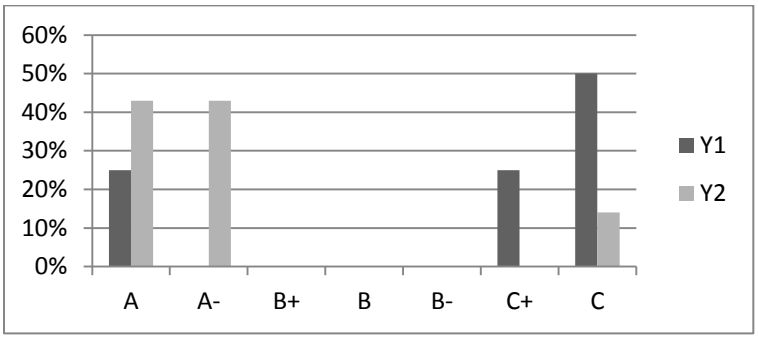

Figure 1: ENGR3400 Student Final grade distribution for Spring Terms Year 1 (Y1) and Year 2 (Y2)

2. **Feedback from Students:** Student feedbacks through course assessment and course reflection have been crucial to making enhancements to this course. End-of-term student course and instruction assessment are conducted using ***Student Instructional Report (SIR)*** **survey mechanism**. At the end of course delivery students are asked to reflect upon the course in relation to the text used, course materials provided, projects assigned, quality of lab/lecture delivery, allocation for contents, and areas of improvement. Table 7 depicts the mean score for selected assessment items for the spring term of two different years for ENGR3400 taught by the same instructor.

Table 7: Selected Assessment Items from SIR Mechanism

Item	Spring Term Year 1	Spring Term Year 2	Comparative Mean for 4 Year Institutions
Course Organization and Planning	3.54	4.34	4.31
Communication	3.30	4.06	4.37
Faculty/Student Interaction	3.18	4.26	4.37
Effectiveness of Student Assessment Tasks	3.14	3.88	4.17
Course Outcomes	3.17	3.63	3.75
Overall Evaluation	3.29	3.86	4.01

3. **Feedback from Employers:** Issues in software development are every developer's nightmare. Customers demand bug free, on-time, and within budget delivery of software. This is only possible through employees that are well versed in V&V. Prospective employers are pleased to know that a course on Software Verification and Validation is offered at Robert Morris University. Periodically the course outline is shared with prospective employers who provide constructive comments. Since the offering of this

course almost 75% of the students have been employed as interns and 85% of graduates have been employed as full time employees based on them having taken this course.

Challenges

The two institutions being studied have a common goal but use different approaches to meet these goals. Hence they have their own challenges.

Milwaukee School of Engineering (MSOE)

While this approach to teaching software verification is effective, there are some distinct challenges to teaching in this manner. The first problem involves scalability. To prevent academic dishonesty, each year, a new set of unique defects is injected into each code base for each section of the course that is offered. That poses a significant challenge to determine valid defects that are appropriate to the types of testing being done by the students. There also is the inevitable problem that students in different lab sections will simply perform a difference analysis between the two code bases and determine where the defects have been injected, mitigating the purpose for the lab.

A second major challenge is the very technology program offered by MSOE. Each year, each project and each tool must be retested for compatibility with the image issued to students, as the image on the student's machines does not match the image given to faculty members. Occasionally, there are significant compatibility problems with the software being used in lab and the machines upon which it is to run. This, of course, is premised on the fact that the students will not change their machines away from the provided image, which rarely is the case. For example, in one class, three different students saw three slightly different incorrect behaviors when the banking program executed. One of the three students had the correct image, while one had upgraded to a 64 bit operating system and another student had replaced Windows with Linux and then executed Windows inside of a virtual machine. These technical issues can really distract students from the core focus of the class.

The 2-2-3 (2 hours lecture, 2 hours lab for 10 weeks) format of the class also offers a distinct challenge depending on where the lab is placed in the week. Given that the course falls during the fall quarter, Monday classes are often missed due to the Labor Day holiday. Depending on how the holiday falls, often the first meeting of the class is a lab session. This format also makes it very challenging to keep on schedule, for even one day failing to meet all of its detailed outcomes will impact the rest of the week.

Robert Morris University (RMU)

In RMU in spite of all the challenges listed below active learning in software verification and validation has been effective. The following are current challenges in making the approach more effective.

a. **Working Software:** Initially large working software developed by the author was used for the project. However using this year after year was not practical. Due to propriety and platform reasons large working software could not be obtained. Hence it was decided that

student projects from prior courses would be used for the project in this course. A positive in this approach is the ownership of the program. As students initiate the idea for their programs they have passion and drive for making sure the program works. As a result students are serious about adequately testing the product.

b. **Student Knowledge Levels and Software Programs:** Three categories of students enroll in the software engineering program: students who are taking programming classes at RMU, students who have prior programming experience in high school, and transfer students with heavy focus on software courses predominantly from community colleges. These categories of students have different level of understanding of software programming which is directly reflected on the programming projects that they complete in a prior course and use in this course for a verification project. The project requires preparation of a minimum of 50 test cases. For some projects due to the quality of the project adequate test cases cannot be generated. Hence the project assignment has to be tuned to cater for these categories of students. An approach used is to reuse past student projects by deliberately adding a few bugs.

c. **Transfer Students and Operating Software Programs:** When student transfer from community colleges they transfer the credits for the Fundamentals of Software Engineering course minus an operating software program to be used for the Software Verification and Validation course. Such students are either given past student projects or are asked to bring to class a working program they have developed. In two particular cases in the past 5 years students were asked to perform a black box testing on Microsoft Excel and Access.

d. **Individual Project and Academic Honesty:** The project in the Software Verification and Validation course is an individual project. Students who have worked as a team in a prior course on a working software program are asked to make a copy of the code and use it to perform the project for this course. At times academic dishonesty is an issue but repetitive reminder of academic policy ensures students are doing their own work. Small class size is a huge plus in enforcing academic honesty.

e. **Lab Space:** Certainly more can be achieved in delivering verification contents like automated testing, permanent testing environments, etc. However as the Software Engineering program has to share available lab spaces with other engineering programs additional lab space is a constraint for expanding hands-on assignments focusing on additional verification tools.

Besides the students, working programs, and lab constraint discussed above another challenging area is expansion of V&V content either through additional courses or through inclusion in existing courses. Verification contents both theoretical and tools are taught throughout the software engineering curriculum. However the percentage of content taught and the percentage of hands-on assignments carried-out differ in different courses. For example for the course Fundamentals of Software Engineering 10% of the theoretical contents and 10% of the hands-on assignments focuses on verification. Though it is desired that more courses focusing on additional verification contents be offered the structure of the software engineering program

(concentration) makes it difficult to introduce additional courses as there are multiple constraints to be met to keep the course balanced and within 126 credits. Likewise it is not possible to add more contents to existing software engineering courses as these courses have other coverage areas. The table below lists percentage of verification content and % of hands-on assignments for four courses.

Table 8: Coverage of Verification in Different Software Engineering Courses

Course	Verification Content	Verification Hands-on Assignments
Fundamentals of Software Engineering	10%	10%
Software Verification and Validation	45%	45%
Distributed Systems		15%
Capstone Course		20%

Conclusions and Recommendations

This article has shown in detail two different approaches that have been taken towards the teaching of software verification. Both approaches strongly use active learning to promote students understanding of software verification. Through different means, both approaches have also been shown to be effective at teaching students the important task of verifying the correct operation of software systems. Both approaches have been successful but both have challenges which either have a work-around or limit expansion.

Bibliography

[1] Zakaria, Z. *A State of the Practice on teaching Software Verification and Validation.* Proceedings of the 2009 ASEE Annual Conference, 2009.

[2] Kaner, C., *Inefficiency and Ineffectiveness of Software Testing: A Key Problem in Software Engineering*, National Defense Industrial Association Workshop: Top 5 Software Engineering Problems or Issues Prevalent Within the Defense Industry, Washington DC, August, 2006.

[3] Collofello, J. S., *Introduction to Software Verification and Validation.* SEI Curriculum Module SEI-CM-13-1.1 December, 1988.

[4] Software Engineering 2004: Curriculum Guidelines for Undergraduate Degree Programs in Software Engineering, http://sites.computer.org/ccse/SE2004Volume.pdf

[5] Software Engineering: Body of Knowledge 2004 Version. IEEE Computer Society, 2004.

[6] Kaner, C., & Padmanabhan, S., *Practice and transfer of learning in the teaching of software testing,* SLIDES Conference on Software Engineering Education & Training, Dublin, July 2007.

[7] Williams, L., *Teaching an Active Participation Course in Software Reliability and Testing, How Should Software Reliability Engineering Be Taught"*?, Panel Paper at the IEEE International Symposium of Software Reliability Engineering 2005, Chicago, IL.

[8] Ludi, S., *Teaching Software Testing as a Problem-Based Learning Course*, Third Annual Workshop on the Teaching of Software Testing (WTST 3) February 6 – 8, 2004, Melbourne, Florida.

[9] Kaplan, R., *An Argument for Incorporating Debugging into the Teaching of Programming*, 8th Workshop on Teaching Software Testing (WTST 2009) January 30 – February 1, 2009

[10] Acharya , S., *Enhancing the Software Verification and Validation Course through 11 Laboratory Sessions,* 2008 ASEE Annual Conference & Exposition – Software Engineering Constituent Committee, June 22 - 25 - Pittsburgh, PA

[11] Acharya, S., et. al., *Using Student Incepted Projects to Retain Student Interest in Software Engineering*, Technology Interface Journal, Volume 9 No.2, Spring 2009, ISSN# 1523-9926

[12] Software Engineering 2004, *Curriculum Guidelines for Undergraduate Degree Programs in Software Engineering*, A volume of the Computing Curricula Series August 23, 2004, The Joint Task Force on Computing Curricula, IEEE Computer Society, Association for Computing Machinery.