Integrating Model-Based Verification into Software Design Education

Levent Yilmaz and Shuo Wang
Auburn University

Abstract

Proper design analysis is indispensable to assure quality and reduce emergent costs due to faulty software. Teaching proper design verification skills early during pedagogical development is crucial, as such analysis is the only tractable way of resolving software problems early when they are easy to fix. The premise of the presented strategy is based on the observation that fundamental component of any engineering curriculum is the use of formal and sound techniques that facilitate analysis of artifacts produced by students. Yet, fundamental roadblocks exist in bringing the state of the art in design analysis to the classroom due to the steep learning curve and quagmire of theoretical details involved in formal methods. This paper suggests a strategy and tool support that promotes the attainment of design evaluation skills. We also discuss how selective and pragmatic application of formal methods can be used in software design education.

Introduction

There exist efforts in making software testing an integral part of programming courses (Barbosa et al. 2003; Edwards 2001; Edwards 2003a). Such initiatives do not only increase awareness among students about the role and significance of testing, but also help them attain significant program analysis experience. Yet, there are very few suggestions for the improvement of critical analysis skills as part of software design courses (Garlan 1994). The emergent trend toward model-driven development (Gluch and Weinstock 1998) and model-driven architecture (MDA 2004) implicates the necessity of integrating model-based verification methodologies into software design education. Analyzing design artifacts requires educated use of formal methodologies. Unfortunately, current state of the practice relies on informal procedures and previous experience in building similar systems. Lack of proper training and poor management skills are major causes of delayed, bug-riddled, and incomplete software. Teaching proper design analysis skills early in the pedagogical development is crucial, as such analysis is the only tractable way of resolving problems early when they are easy to fix. Given these observations, we explore the following: How can we integrate formal methods seamlessly into introductory software design courses, so that students can (1) appreciate pragmatic utility and use of formal methods without getting into the quagmire of theoretical details, (2) avoid steep learning curves about the syntax of a specific formal method by using alternative abstract templates to critically analyze designs, and (3) discover inconsistencies and design conflicts within the realm of the actual software development process.

Related Work in Bringing Formal Analysis Methods into the Classroom

Current approaches in integrating formal methods into software engineering education fall into three main categories. The first approach is to avoid formal methods. While this strategy is observed in most continuing education programs, its appropriateness for general software engineering education is debatable. The second approach is to devote a specific course with emphasis on formal verification of source code. The advantage of such a course is that students are exposed to a wide variety of formal methods such as Z and VDM. The broad coverage of formal methods provides flexibility to tailor a course to make it relevant to certain software engineering skills. However, broad coverage of formal methods may not enable a student to be proficient on a specific formal approach. Furthermore, the methods are taught in an isolated manner with an emphasis on the notations rather than the underlying principles. This isolated exposure to formal methods prohibits students to apply such approaches to software engineering practices. The third approach is to redesign the entire program so that formal methods are integrated throughout the curriculum. A widely known example is the CMU strategy (Garlan et al. 1992; Garlan 1994), where the graduate program in software engineering is redesigned to facilitate exposure to formal models of software systems. While the CMU strategy presents a novel strategy for comprehensive treatment of formal methods, the curriculum is formulated for graduate students. As such, the strategy presumes familiarity and exposition to advanced logic and discrete math. In the second strategy outlined above, formal methods courses are taught at the undergraduate level following preliminary exposition to discrete math or mathematical logic courses, yet they are treated in an isolated manner on toy source code samples for illustrative purposes. Use of formal methods within the context of software design and modeling is not yet common. The fundamental reason why formal methods are not utilized effectively in undergraduate software design courses can be attributed at the least to the following two reasons: (1) the impedance mismatch between the underlying mathematical underpinning of formal methods and student's semi-formal, if not informal, view of the design problem and (2) lack of tool support for seamless integration of formal methods into software design education.

A Strategy for Integrating Model-Based Verification into Software Design Education

In this section a strategy for the integration of model-based verification (Gluch and Weinstock 1998) into undergraduate software design education is discussed. The Web-based computer aided evaluation center, the components of which are shown in Figure 1, plays a critical role in the proposed strategy.

The system is used within the context of COMP3700, which is a Software Design and Modeling course (COMP3700) at the department of Computer Science and Software Engineering. It is a junior-level course with an average enrollment of 35 students. As a prerequisite, students are expected to be familiar with software construction. This includes experience in topics such as testing, debugging, configuration management, low-level file and device I/O, and systems and event-driven programming. Based on this foundation, COMP3700 presents an integrated set of techniques for software analysis and design based on object-oriented concepts using the UML notation. Introduction to object concepts, fundamentals of an object oriented analysis and design process, use-case analysis, object modeling using behavioral techniques, and design patterns constitute the fundamental topics covered in COMP3700. The structure of the course is recently redesigned to cover software design quality assessment. While the traditional lecture style is preserved, it is decided that analysis and design skills are best acquired in terms of (1) learning by doing (Edwards 2001), (2) critical analysis (Stice 1987), and (3) collaborative problem solving. As such, COMP3700 offers two substantial group projects. The first project entails the development of design models based on the given problem definition. Students develop UML (Booch et al. 1999) models to visualize design artifacts. The second project involves the peer evaluation of a design produced by another group using various quality assessment procedures. To support wide applicability in various course settings, models are translated into an interoperable format called XMI before submission into the central repository. The second phase involves collecting design metrics and performing structural consistency checks.

Figure 2 illustrates the outcome of a structural analysis session. For violated consistency conditions, the rule definition and its category (type) are indicated along with a detailed description. The rules include intra-consistency checks within a single diagram (finite state chart), as well as inter-consistency constraints among distinct model types (i.e., sequence diagrams vs. state charts). Such checks are reported to be useful in group projects as different students are responsible for assuring design coherency and consistency before final project submission. The rules depicted in Figure 2 represent some of the completeness and consistency rule violations. Along with intra-consistency

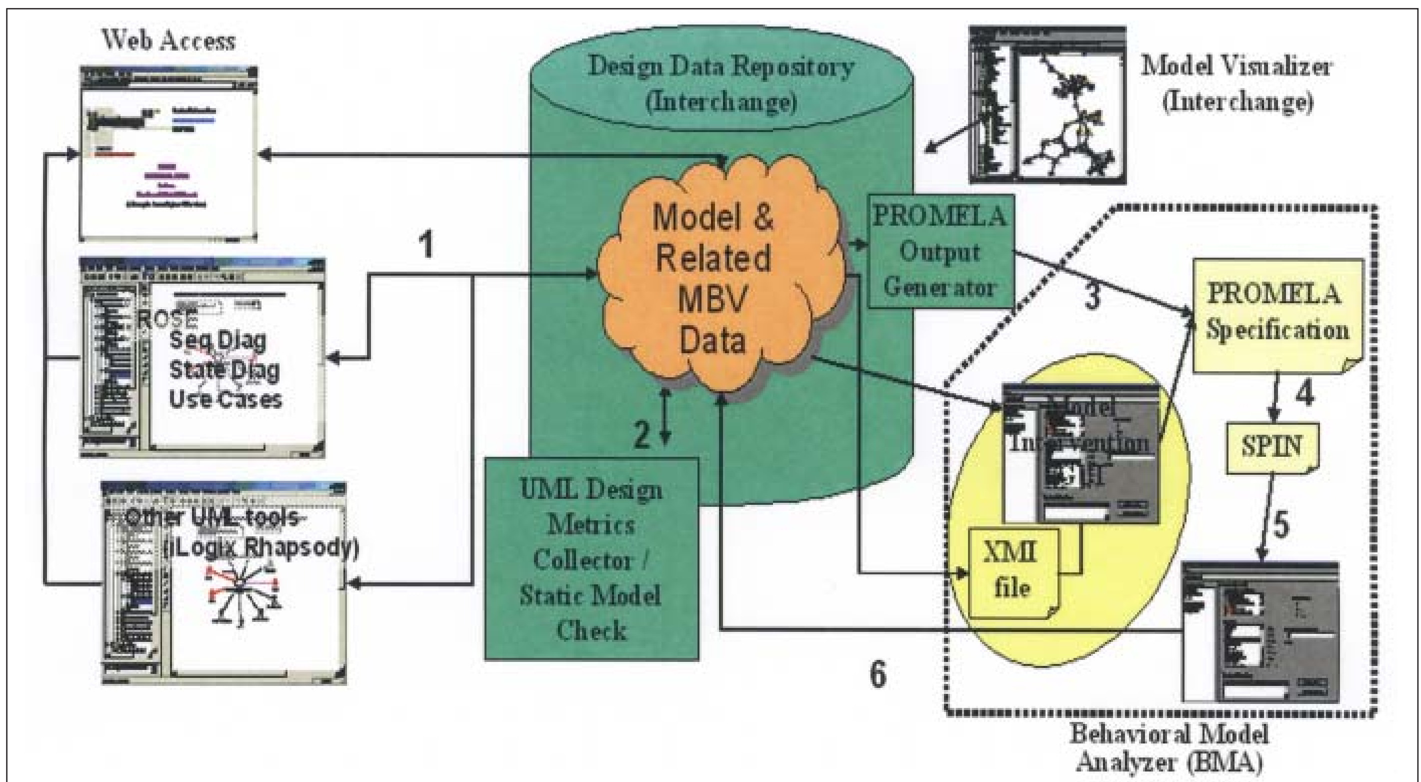


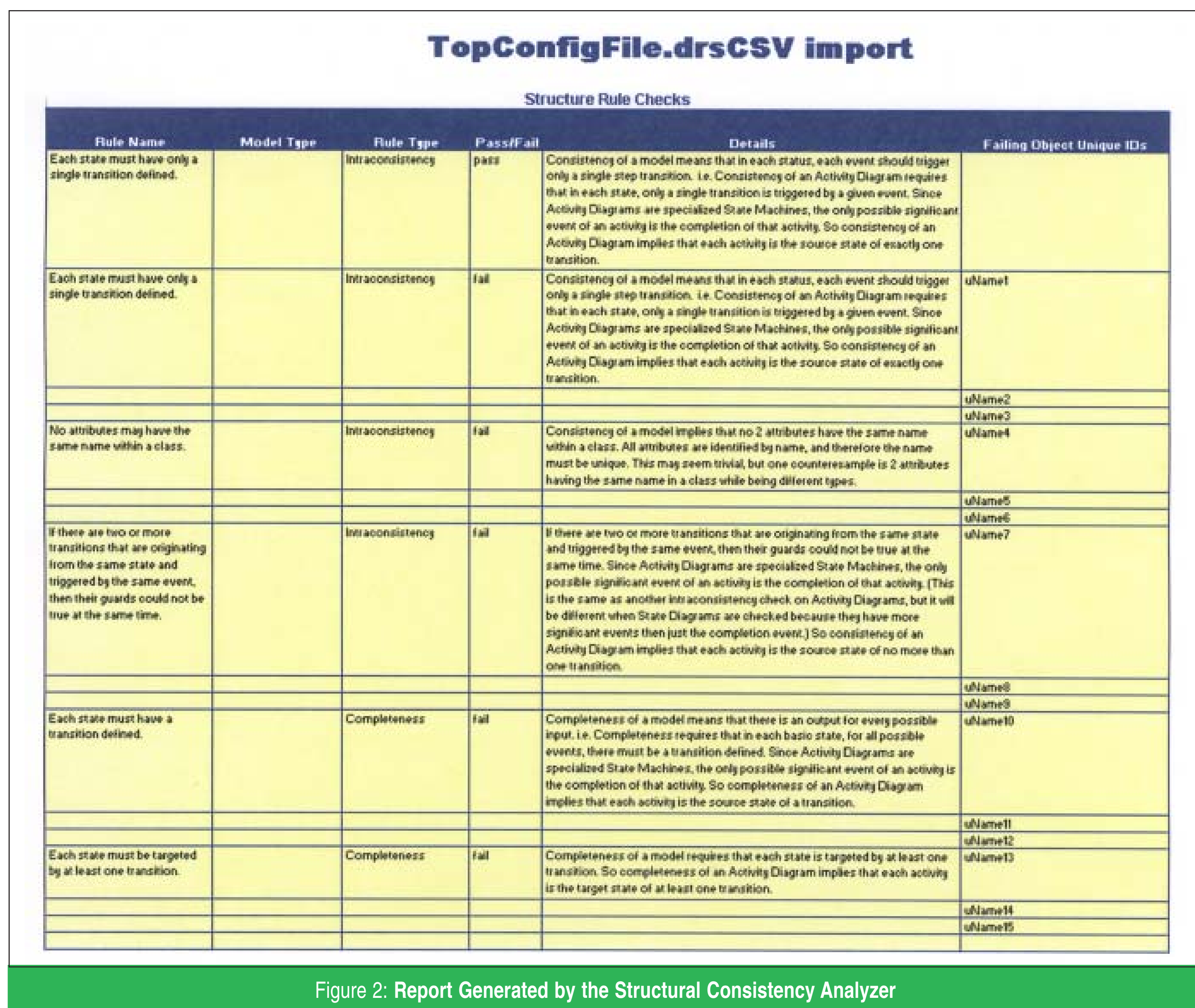
Figure 1: Web-Based Computer-Aided Verification Environment – Web-CAVE

rules, state transitions in finite state machine designs of objects are traced back to messages in sequence diagrams for completeness. Unaccounted state transitions in finite state machines of design objects are common mistakes among inconsistencies between design diagrams. Besides structural consistency analysis, Web-CAVE provides facilities for students to collect various design metrics to relate them to identified high level quality objectives such as maintainability, testability, and reusability.

Structural consistency analyzer component of Web-CAVE is also used by students as a peer-review tool. COMP3700 project guidelines require routine inspections and reviews of design artifacts as part of the new emphasis on design quality assurance. Students understand that design is a process of experimentation involving the continuous discovery of technical information associated with the function, form, and fit of the product. Inspections and peer-reviews are an integral practice in the process of experimentation in COMP3700. Before the development of Web-CAVE, students were required to identify a set of critical questions to overview their designs (hard copies of design charts) and check for violation of design integrity and consistency. Web-CAVE provides a significant number of online services including design consistency rules (see Figure 2) and metrics collectors to facilitate performing efficient and effective reviews.

Behavioral Model Analyzer

Structural analysis significantly reduces errors, but it does not guarantee behavioral correctness. In COMP3700 students are required to develop UML state charts to depict behavioral aspects of design objects. While model checking (Clarke et al. 1986, 1994, 2000) is a state of the art analysis method, its use requires understanding of temporal logic in developing formal specifications. Note, however, students taking COMP3700 lack such for-


TopConfigFile.drsCSV import

Structure Rule Checks

Rule Name	Model Type	Rule Type	Pass/Fail	Details	Failing Object Unique IDs
Each state must have only a single transition defined.		Intraconsistency	pass	Consistency of a model means that in each status, each event should trigger only a single step transition. i.e. Consistency of an Activity Diagram requires that in each state, only a single transition is triggered by a given event. Since Activity Diagrams are specialized State Machines, the only possible significant event of an activity is the completion of that activity. So consistency of an Activity Diagram implies that each activity is the source state of exactly one transition.	
Each state must have only a single transition defined.		Intraconsistency	fail	Consistency of a model means that in each status, each event should trigger only a single step transition. i.e. Consistency of an Activity Diagram requires that in each state, only a single transition is triggered by a given event. Since Activity Diagrams are specialized State Machines, the only possible significant event of an activity is the completion of that activity. So consistency of an Activity Diagram implies that each activity is the source state of exactly one transition.	uName1
					uName2
					uName3
No attributes may have the same name within a class.		Intraconsistency	fail	Consistency of a model implies that no 2 attributes have the same name within a class. All attributes are identified by name, and therefore the name must be unique. This may seem trivial, but one counteresample is 2 attributes having the same name in a class while being different types.	uName4
					uName5
					uName6
If there are two or more transitions that are originating from the same state and triggered by the same event, then their guards could not be true at the same time.		Intraconsistency	fail	If there are two or more transitions that are originating from the same state and triggered by the same event, then their guards could not be true at the same time. Since Activity Diagrams are specialized State Machines, the only possible significant event of an activity is the completion of that activity. (This is the same as another intraconsistency check on Activity Diagrams, but it will be different when State Diagrams are checked because they have more significant events then just the completion event.) So consistency of an Activity Diagram implies that each activity is the source state of no more than one transition.	uName7
					uName8
					uName9
Each state must have a transition defined.		Completeness	fail	Completeness of a model means that there is an output for every possible input. i.e. Completeness requires that in each basic state, for all possible events, there must be a transition defined. Since Activity Diagrams are specialized State Machines, the only possible significant event of an activity is the completion of that activity. So completeness of an Activity Diagram implies that each activity is the source state of a transition.	uName10
					uName11
					uName12
Each state must be targeted by at least one transition.		Completeness	fail	Completeness of a model requires that each state is targeted by at least one transition. So completeness of an Activity Diagram implies that each activity is the target state of at least one transition.	uName13
					uName14
					uName15

Figure 2: **Report Generated by the Structural Consistency Analyzer**

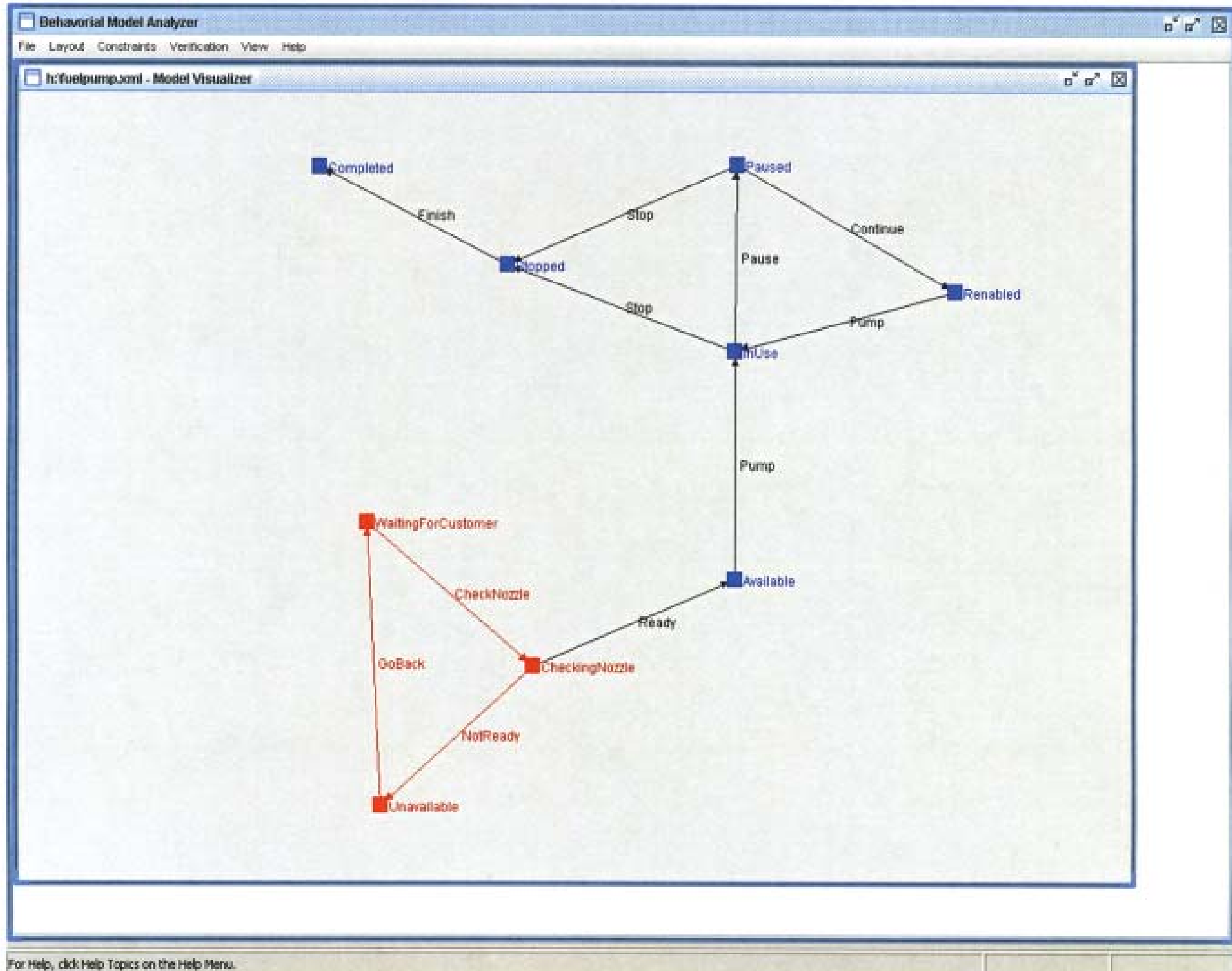


Figure 3 **– A Potential Infinite Loop**

mal methods background. As a result bringing this advanced technology, in its basic form, to improve critical design analysis skills is a challenging task.

To this end, a behavioral model analyzer (BMA) tool is developed to help students formulate abstract high-level queries for their behavioral designs. The constraints, against which the model is checked, are defined in terms of a menu of templates. The formulated pattern is then automatically mapped onto a temporal logic specification of the SPIN model checker. By using a high-level constraint composer menu, the students are then able to query their designs to reason about the properties suggested (Dwyer et al. 1999).

Potential errors detectable by the analyzer include (1) **unreachable states**, which are states that will never be reached in the state machine of the model, (2) **missing end states**, (the state machine in the model never terminates), (3) **infinite loops,** (4) **constraint violation**. As shown in Figure 3, unreachable states are redrawn as red circles; an extra end state in red label “End” will appear when there is no proper ending state found, and a loop of solid red circles with red transition arrows among them suggests an infinite loop. If after model checking is complete, no change in the visualization occurs, this indicates that the analyzer did not find any potential problem. In model checking, a complete model not only contains the specification of system behavior, defined in XMI files and eventually translated into Promela input files, but also the formalization of correctness requirements that apply to the model.

For example, let P be a behavioral property expressed in numerical logic, $X = 0$. Let it also be given that X is some integer variable in the model specification that represents certain behavior, and 0 is the only correct value X is supposed to have. It is possible to evaluate, under certain circumstances, whether this correctness property remains true or not, e.g., will X always be 0; will X eventually be 0, etc. Correctness properties are supplied to the model checker in the form of linear temporal logic (LTL). Since students are typically unfamiliar with LTL, they create an equivalent “constraint pattern” using the BMA user interface. Students still

need to supply the definition of P, Q, or R, known as a "token" in a constraint pattern, but the interface provides a useable means to do so. Table 1 below lists the implemented correctness properties within the BMA.

Evaluation and Improvement of Web-CAVE

While a comprehensive survey or field study is not yet performed to evaluate the effectiveness of the system, group project team leaders are asked to reflect upon their group design project experience. A common response among team leaders was on the convenience of Web-CAVE in locating inter-diagram inconsistencies. That is, as different members of a group develop distinct aspects and views of the same system (i.e., conceptual model, interaction diagrams, class design diagrams, and finite state designs), inconsistencies among diagrams become a significant concern. Structural consistency analyzer in Web-CAVE help members of a group recognize integrity problems earlier and coordinate their activities to assure design correctness. A common criticism is the difficulty of use of the system as well as the lack of clarity of the relevance of collected design metrics to selected quality objectives. Web-CAVE is now being extended to provide a framework by which design indicators (i.e., coupling) can be mapped to design principles (i.e., modularity) that govern the process, by which the design is produced. Adherence to such principles is expected to help achieve quality objectives such as reusability. Such a framework will provide the necessary heuristics to infer the degree to which a quality objective is achieved in a group project.

Summary and Conclusions

The steep learning curve and effort involved in applying conventional formal methods in software engineering are the primary roadblocks in their practical use. Realizing this barrier, integration of model-based verification perspective to software design and modeling course is discussed. The premise of the strategy is based on the observation that fundamental component of any engineering curriculum is a collection of formal and sound techniques that facilitate analysis of artifacts produced by students. We discuss several opportunities to facilitate the integration of model-based verification into undergraduate software design education. To this end, high-level architecture of a web-based computer aided verification system is presented to illustrate how attainment of analysis and verification skills can be promoted through an online design evaluation system. Also, the notion of abstract verification patterns is used to bridge the gap between the mathematical underpinnings of formal methods and student's semi-formal design worldview.

Correctness Property Types in BMA	System Behavior Patterns (Dwyer 1999)
always P	Invariance
eventually P	Guarantee
P implies eventually Q	Response
P implies Q until R	Precedence
always eventually P	Recurrence
eventually always P	Stability
eventually P implies eventually Q	Correlation

Table 1: **Abstract Analysis Patterns**

References

Barbosa, E. F, LeBlanc R., Guzdial M., Maldonado C. J. (2003). "The Challenge of Teaching Software Testing Earlier into Design," *Workshop on the Teaching of Software Testing (WTST).* February 7-9, 2003 Melbourne, Florida.

Booch G., J. Rumbaugh, and I. Jacobson (1999). *The Unified Modeling Language User Guide.* Addison-Wesley.

Clarke E. M., Emerson E., Sistla A. 1986. "Automatic Verification of Finite State Concurrent Systems using Temporal Logic Specifications," *ACM Transactions on Programming Languages and Systems*, vol. 8 no. 2, pp. 244-263.

Clarke E. M., Grumberg O. and Long E. D. 1994. "Model Checking and Abstraction," *ACM Transactions on Programming Languages and Systems*, vol.16 no.5, pp. 1512-1542.

Clarke E. M., Grumberg O. Peled D. 2000. *Model Checking*. MIT Press.

Dwyer B. M., Avrunin S. G., and Corbett C. J. (1999). "Patterns in Property Specifications for Finite-state Verification," in *Proceedings of the 21st International Conference on Software Engineering*, May, 1999.

Edwards H. S. (2001). "Can Quality Graduate Software Engineering Courses be Delivered Asynchronously On-Line," In *Proceedings of the ICSE'2000,* pp. 676-679.

Edwards H. S. (2003a). "Automatically Assessing Assignments That Use Test-Driven Development," *Workshop on the Teaching of Software Testing (WTST)*. February 7-9, 2003 Melbourne, Florida.

Edwards H. S. (2003b). "Web-CAT Automated Grader," http://web-cat.cs.vt.edu/grader/.

Garlan D, Shaw M., Okasaki C., Scott C., and Swonger R. (1992). "Experience with a course on architectures for software systems," In *Proceedings of the Sixth SEI Conference on Software Engineering Education*. Springer Verlag, LNCS 376, October 1992. Also available as CMU/SEI technical report, CMU/SEI-92-TR-17.

Garlan D. (1994). "Integrating Formal Methods into a Professional Master of Software Engineering Program," In *Proceedings of The 8th Z Users Meeting*, June 1994.

Gluch P. D. and Weinstock B. C. (1998). "Model-Based Verification: A Technology for Dependable System Upgrade," CMU/SEI-98-TR-009, September 1998.

MDA. (2004). MDA: "The Architecture of Choice for a Changing World," http://www.omg.org/mda/executive_overview.htm.

Stice E. J. (1987). *Developing Critical Thinking and Problem-Solving Abilities*. Jossey-Bass Inc., San Francisco, CA, 1987.

Levent Yilmaz is assistant professor of Computer Science and Software Engineering in the College of Engineering at Auburn University. Before joining to faculty in 2003 he was a senior research engineer in the Simulation and Software Division of Trident Systems Incorporated, in Fairfax, Virginia. Dr. Yilmaz earned his Ph.D. and M.S. degrees from Virginia Tech. He worked as a lead project engineer and principle investigator for advanced simulation methodology, model-based verification, and simulation reuse technology development efforts. His research interests are on advancing the theory and methodology of simulation modeling, agent-directed simulation to explore and understand dynamics of software processes and project dynamics, and education in simulation modeling. He is a member of ACM, IEEE Computer Society, Society for Computer Simulation International, and Upsilon Pi Epsilon.

Shuo Wang is graduate student of Computer Science and Software Engineering in College of Engineering at Auburn University. He earned his B.S. degree in Computer Science from Georgia Institute of Technology. His academic and research interests are software modeling and simulation, software engineering, computer networks, and human-computer interaction. He is a member of ACM and IEEE.