Int. J. Engng Ed. Vol. 22, No. 2, pp. 407–418, 2006
Printed in Great Britain.

0949-149X/91 $3.00+0.00

A Strategy and Tool Support to Motivate the Study of Formal Methods in Undergraduate Software Design and Modeling Courses*

SHUO WANG and LEVENT YILMAZ
Department of Computer Science and Software Engineering, College of Engineering, Auburn University, Auburn, AL 36849, USA. E-mail: yilmaz@auburn.edu

Proper design analysis is indispensable to assure quality and reduce emergent costs due to faulty software. Teaching proper design verification skills early during the pedagogical development of a software engineer 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 a fundamental component of any engineering discipline is the use of formal and sound techniques that facilitate analysis of produced artifacts. Yet, fundamental roadblocks exist in bringing the state of the art in formal design analysis to the undergraduate software engineering classroom due to the steep learning curve and quagmire of theoretical details involved in formal methods. This paper suggests a strategy and tool support to improve the attainment of software design verification skills. We illustrate how selective and pragmatic application of model-based verification methods can be used in software design education via tools that aim to bridge the gap between students' semi-formal design worldview and the formalism underlying formal methods.

AUTHOR QUESTIONNAIRE

1. The paper discusses materials/software for a course in Software Engineering.
2. Students are taught this course Computer Science, Software Engineering and Computer Engineering departments
3. Level of the course (year): Junior (3rd year).
4. Mode of presentation: lectures, group design project, programming assignments.
5. The material is presented in a regular course.
6. Hours required to cover the material: 3 hours/week for 15 weeks.
7. Student homework or revision hours required: 5 homeworks over 15 weeks.
8. The novel aspects presented in this paper involve integrating the formal methods transparently into software design courses, so that students can:
   - appreciate pragmatic utility and use of formal methods without getting into the quagmire of theoretical details,
   - avoid steep learning curves about the syntax of a specific formal method by using alternative abstract templates, and
   - discover inconsistencies and ask pertinent questions about designs within the realm of the actual software development process.
9. The standard textbook recommended in the course is Craig Larman, *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design,* Third Edition, Prentice-Hall (2005).
10. The material is not covered in the textbook.

* Accepted 27 September 2005.

INTRODUCTION

SOFTWARE ERRORS are prevalent and detrimental; they cost the USA economy an estimated $59.5 billion annually, or about 0.6 percent of the gross domestic product [1]. According to numerous other sources [2–6] software projects fail to meet deadlines, are suffused with defects, run over budget, and do not include many of the features present in the original specifications. The NIST study [1] indicates that while all errors cannot be removed, more than a third of these costs, or an estimated $22.2 billion, could be eliminated by an improved quality evaluation infrastructure that enables earlier and more effective identification and removal of software defects. Due to this imminent issue, concern has grown over the levels of assurance of software intensive systems [6].

Mitigating this problem requires immediate attention of educators to improve the pedagogy of software engineering by facilitating transparent integration of quality evaluation techniques into the curriculum. Fortunately, there already exist efforts in making software testing an integral part of programming courses [7–9]. 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. While teaching students how to instill confidence in software has already received significant attention, reasoning and asking pertinent question about design have not received the same level of treatment.

The emergent trend toward model-driven development [10] and the adoption of principles underlying the model-driven architecture (MDA) [11] suggests the necessity of promoting critical design analysis skills. The significance of teaching formal methods in the context of emerging trends in software development is also argued by Davies and Simpson [12] and Robinson [13]. They both agree that analyzing design artifacts requires educated use of formal methodologies. Unfortunately, current state of practice extensively relies on informal procedures [14].

Lack of proper training in formal analysis is one of the major causes of bug-riddled and inconsistent software. Teaching proper design analysis skills early in the pedagogical development is crucial [15], as such analysis is the only tractable way of resolving problems early when they are easy to fix. Besides, a fundamental component of any engineering discipline, including software engineering, is the use of formal and sound techniques that facilitate analysis of artifacts produced by students [16]. Yet, the impact of formal methods in software engineering practice [13], as well as education, is minuscule [17]. The fundamental reasons why formal methods are not effectively utilized are attributed to:

- the impedance mismatch between the underlying mathematical underpinning of formal methods and student's semi-formal, if not informal, view of the design problem [18];
- the lack of tool support for seamless integration of formal methods into software design education [19, 12].

How can we integrate formal methods transparently into software design courses? Given the above observations, we explore methods that can:

- appreciate pragmatic utility and use of formal methods without getting into the quagmire of theoretical details;
- avoid steep learning curves about the syntax of a specific formal method by using alternative abstract templates;
- discover inconsistencies and ask pertinent questions about designs within the realm of the actual software development process [20].

The proposed strategy entails the integration of the Model-Based Verification (MBV) methodology [21] into an undergraduate software design course. In this course students participate in group projects to formulate requirements and develop UML-based [22] software designs. The Behavioral Model Analyzer (BMA) is part of a comprehensive Web-based Computer Aided Verification Environment (Web-CAVE). It is used by students to verify the behavior designs to discover and locate errors before coding starts. Web-CAVE is a student-centered design evaluation center that integrates structural consistency analyzer, design metrics collector, and the BMA. In this paper, we focus on the design rationale, development, and use of the BMA in facilitating the attainment of design verification skills.

The premise of the underlying strategy is partly based on the utilization of high-level and easy-to-learn templates that do not require a background in formal mathematical logic. Our findings indicate that by using the tool, students gain valuable experience in identifying potential sources of discrepancies and faults within their behavioral models. Furthermore, the tool provides a basis for the development of an online grading system to support instructors.

This paper reviews current approaches in using formal methods within the software engineering curriculum along with the Model-Driven Architecture (MDA) trend in software development to suggest a pragmatic strategy based on selective and pragmatic application of model checking. The overview of the context within which the BMA is used is presented and development of the BMA is outlined. We demonstrate the utility of the BMA in terms of a case study and conclude by discussing potential avenues for further research to address further issues pertaining to improvement of BMA's transparency.

FORMAL METHODS IN SOFTWARE DESIGN EDUCATION

Formal methods are mathematically-based techniques for describing system properties. As such, they provide frameworks within which engineers can specify, develop, and verify systems in a systematic, rather than ad hoc, manner [23]. Recently, in meeting the challenges to reduce software errors and increase the quality of software systems, a set of formal software engineering techniques and practices for software verification, known as MBV, has been developed [21]. Seamless and transparent integration of the MBV philosophy into a software design course constitutes the foundation of the proposed strategy.

Current strategies for integrating formal methods into software design education

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 open to debate. 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 [24] and VDM [25]. 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 [19]. 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 from applying 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 [26, 27], 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.

Model-based verification

The challenges involving in the motivation of the study of formal methods in the classroom are well-documented [28]. Also, given the empirical evidence that computer science students do experience difficulty with the concepts underlying formal methods [18], strategies to overcome the, so-called, 'mental resistance' phenomena are needed. MBV and its underlying philosophy provide the requisite impetus to address this need.

The foundation of the MBV involves the selective use of different levels of abstraction and formalism in the systematic generation and analysis of 'essential' models of a system. Essential models are simplified abstract representations that capture the blueprint of a system. The current work in MBV is building and extending the foundation of various promising techniques, especially formal techniques, and adapting their underlying principles and methodologies to improve software quality. In the broader perspective, MBV can be seen as an integral technique in verification and testing practices that reduce the occurrence of errors [21].

A well-established and accepted method in MBV is model checking [29–31]. Model checking relies on building a finite model of a system and checking that a desired property holds in the model. It involves an exhaustive state space search which is guaranteed to terminate. Model checking can be applied to analyze specifications of software systems. There are two broad approaches to model checking that are widely used. The first is temporal model checking, a technique that uses specifications expressed in a temporal logic. The second approach uses specifications given as an automaton, and the system is also modeled as an automaton. The system is then compared to the specifications to determine whether its behavior is consistent with the specifications. There are tools that facilitate the checking of expected model-based system behavior with respect to safety and liveness properties of systems. Yet, these tools require advanced knowledge on using temporal logic to specify such properties.

INTEGRATING MODEL CHECKING INTO DESIGN EDUCATION

A practical strategy for bringing formal methods to undergraduate software design education needs to be based on a level of abstraction that does not require extensive exposure to the formal syntax of the specific method. This fundamental requirement influenced and guided the overall design of the Web-CAVE, which is a web-based environment that enables students to evaluate the structure and behavior of their design models. Web-CAVE and its BMA tool, which is the focus of this paper, are used in the context of an undergraduate software design and modeling course.

COMP3700—an undergraduate software design and modelling course

The Software Design and Modelling course (COMP3700) at the department of Computer Science and Software Engineering is a junior-level course with an average enrollment of 35 students. As a prerequisite, students are expected to be familiar with object-oriented (OO) software construction. Based on this foundation, COMP3700 presents an integrated set of techniques for software analysis and design using the UML notation. Introduction to object concepts, fundamentals of OO 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 revised to cover software design quality assessment. While the traditional lecture style is preserved, it is recognized that analysis and design skills are best acquired in terms of:

- learning by doing [8, 34];
- critical analysis [32];
- collaborative problem solving [33].

COMP3700 offers two substantial group projects. The first project entails the development of design models based on the given problem definition. Students develop UML [22] models to visualize

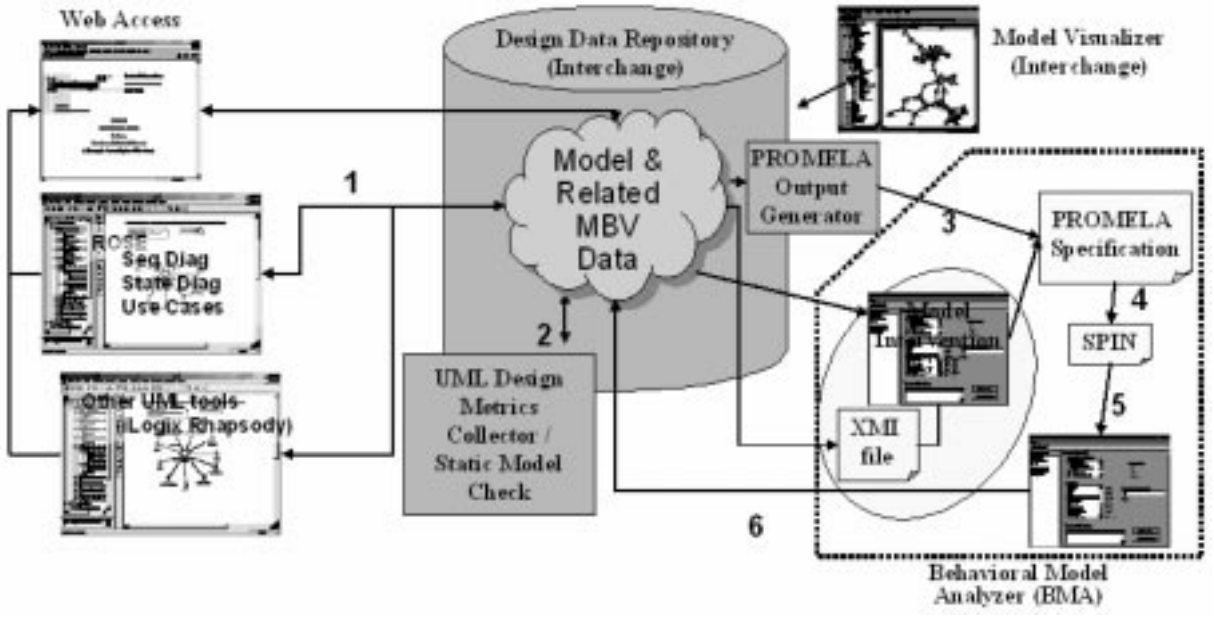

Fig. 1. Web-CAVE.

design artifacts. The second project involves the evaluation of a design produced by another group using various quality assessment procedures. The Web-based computer-aided evaluation center (see Fig. 1) plays a critical role in supporting the second group project.

The Web-CAVE environment

Web-CAVE is comprised of three components: (1) structural consistency analyzer, (2) design metrics collector, and (3) the BMA. As shown in Fig. 1, the design models can be uploaded by students onto a central repository through a web-based interface.

To facilitate interoperability across different UML design tools, the models are translated into XMI (an XML-based exchange format) before they are stored in the database. The structural model analysis component performs consistency analysis across different diagrams uploaded by the members of a group project. In addition to structural consistency analysis, Web-CAVE provides facilities for students to collect various design metrics. Inspections and peer-reviews are an integral practice in the process of experimentation. Before the development of Web-CAVE, students were required to identify a set of critical questions to overview their designs and check for violation of design integrity and consistency.

Web-CAVE provides online services including design consistency rules and metrics collectors to facilitate performing efficient and effective reviews. However, this paper focuses on the behavioral model analysis feature of Web-CAVE; so, we refer interested readers to [35] for a review of the structural analysis capabilities of the environment.

A strategy for integrating model checking into software design courses

Structural analysis significantly reduces sources of errors, but it does not guarantee behavioral correctness. In software design classes, students are often required to create UML statecharts to depict behavioral aspects of their designs. While model checking [29–31] is a state-of-the-art analysis method for analyzing and verifying state machine models, its use requires understanding temporal logic in developing formal specifications. Note, however, students taking COMP3700 lack such formal background. Furthermore, common myths on formal methods [36] inhibit the recognition of the utility of formal methods by students.

As a result, bringing this advanced technology, in its basic form, to improve critical design analysis skills, is a challenging task. The BMA aims to bridge the gap between students' semi-formal design worldview and the formalism underlying model checking. The BMA accepts a software design model and a property specifying how the model is required to behave.

The BMA then performs model checking using the model and the property to display the results back to the student in an informative form. Although this description hardly differs from the operation of any other model checking tool, the BMA places emphasis on the following features that are not present in other model checkers:

- The inputs are UML models, which are commonly used in software design, rather than a tool-specific model description language, to which students have little exposure.
- The properties defining the required behaviors of the models can either be supplied or derived from UML sequence diagrams in the form of

specification templates rather than temporal logic, thus eliminating the requisite mathematical skills involved in formal methods.
- The results of model checking are shown using graph visualization rather than text, so that students can have better intuition about the source of errors.

The features of the BMA are implemented using three incremental steps. The first step includes the translation of UML statechart models for the purpose of model checking, the construction of the specification template, and a graph visualization to show the counter example that violates the required property. The second step incorporates the capability to recognize limited forms of behavioral properties in terms of specification templates. The third step adds the capability to visualize the specification finite state machine generated from the required behavioral properties.

Types of input models used in the BMA

The Unified Modeling Language (UML) [22] is a general-purpose visual modeling language that is used to specify, visualize, construct, and document the artifacts of a software system. UML is capable of capturing information about the static structure and dynamic behavior of the system. The static structure defines the collection of discrete objects that make up the system. The dynamic behavior defines the history of objects over time and the communication among objects to accomplish goals. UML is not a highly formal language, but rather a semi-formal modeling language for discrete systems. The UML state machines and sequence diagrams are widely used to specify dynamic behavior of software components. The state machine view describes the dynamic behavior of objects over time by modeling the lifecycles of objects of each class. A state machine is a graph of states and transitions, and it describes the response of an instance of the class to events that it receives. Events represent the kinds of changes that an object can detect—the receipt of calls or explicit signals from one object to another, a change in state values, or the passage of time. A state is defined as a set of object values for a given class.

Converting UML models into PROMELA

Consistent with the philosophy of the MDA initiative [26], the BMA converts a UML statechart into a PROMELA code skeleton that can easily be augmented by students, who are not familiar with the PROMELA syntax. The PROMELA code translated from a UML statechart does not provide actions and the logic needed to execute the transitions that update the state variables. Yet, the SPIN model checker that interprets the PROMELA code needs to update the state variables to change the state of the model by executing transition logic. This requires explicit specification of transition code. Since UML statecharts do not currently provide conventions to define the transition code, the BMA tool enables students to enter the code through a graphical interface to complete the specification of the model. The process by which the PROMELA code is generated is demonstrated below. The process is reminiscent of the code generation and simulation efforts advocated in the MDA initiative, where the source code is generated from UML models to reduce the coding effort in software development. The same philosophy is applied in the BMA tool to improve verification of software designs by reducing the effort required to develop formal specifications.

Deriving constraints from specification templates

In model checking, a property (i.e., required behavior) of the model is specified using temporal logic. The steep learning curve involved in using mathematical logic is one of the core reasons why formal methods have been lacking in undergraduate design education. Yet, it is imperative for the BMA to navigate around this obstacle. As a result, the BMA uses the notion of specification templates [37] to describe the required property of the model. A specification template is a generalized description of a commonly occurring requirement on the permissible state sequences in a finite-state model of a system, and it describes the essential constraints of some aspect of the system's intended behavior. The specification templates are generalized in a hierarchical structure in terms of their scopes for formal specification and verification.

The scope of a template is the extent of program execution over which the template must hold. It is determined by specifying a starting and an ending state for the template. Therefore the scope consists for all states beginning with the starting state and up to but not including the ending state. As shown in Fig. 2, there are five different scopes:

- *Global*—the entire model execution.
- *Before*—the execution up to a given state.
- *After*—the execution after a given state.
- *Between* – any part of the execution from one given state to another.
- *After-Until*—just like *Between*, but the designated part of the execution continues even if the second state does not occur.

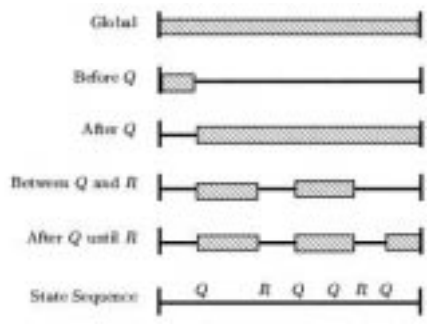

Fig. 2. Scopes of specification templates.

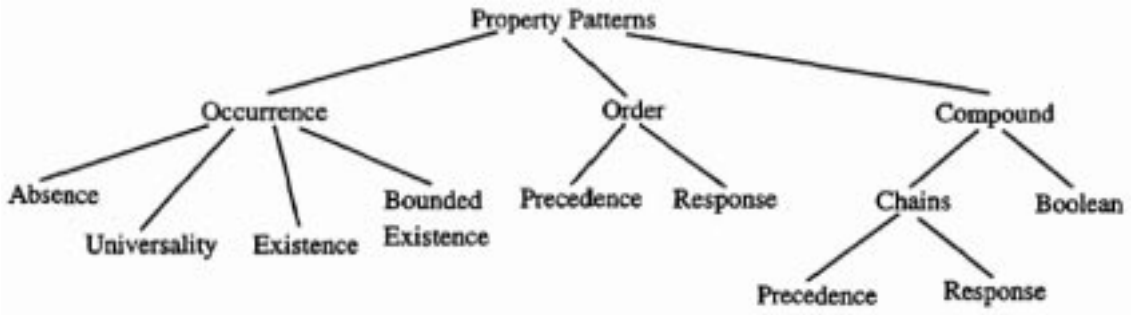

Fig. 3. Taxonomy of specification templates.

Common specification templates used in model checking are shown in Fig. 3. Each template is translated into its temporal logic formula by the BMA. When working with a specification template, only states required by the particular template need to be supplied by the student. The process by which a template is instantiated is shown below using a simple example. The semantics of the commonly used templates are defined in [37].

The *occurrence* template type includes concrete templates from which students select to specify the following types of constraints:

- *Absence*—A given state or event does not occur with a scope. This template is also known as *Never*.
- *Existence* – A given state or event must occur within a scope. This template is also known as *Future* or *Eventuality*.
- *Bounded Existence*—A given state or event must occur k times within a scope.
- *Universality*—A given state or event occurs throughout a scope. This template is also known as *Globally*, *Always* and *Henceforth*.

Interested readers can find more about the rest of the templates along with their temporal logic specifications in [37].

THE DESIGN OF THE BMA

The diagram shown in Fig. 4 provides the physical layout of the implementation components of the BMA, grouped with respect to their functions. The components in the BMA can be grouped into five subsystems. The subsystem *UI/Controller* contains the *BMADesktop*, which performs interaction with the student to facilitate communication among other components.

The *ModelChecking* performs model checking and retrieves the raw results. The *Visualization* provides the results of model checking using graph visualization. The BMA is enacted when the *DestopBMA* component initializes. This component contains functions to call other components in order to execute the request by using interactive user interface widgets. Using these widgets, a UML design model is converted into PROMELA language, supplies specification

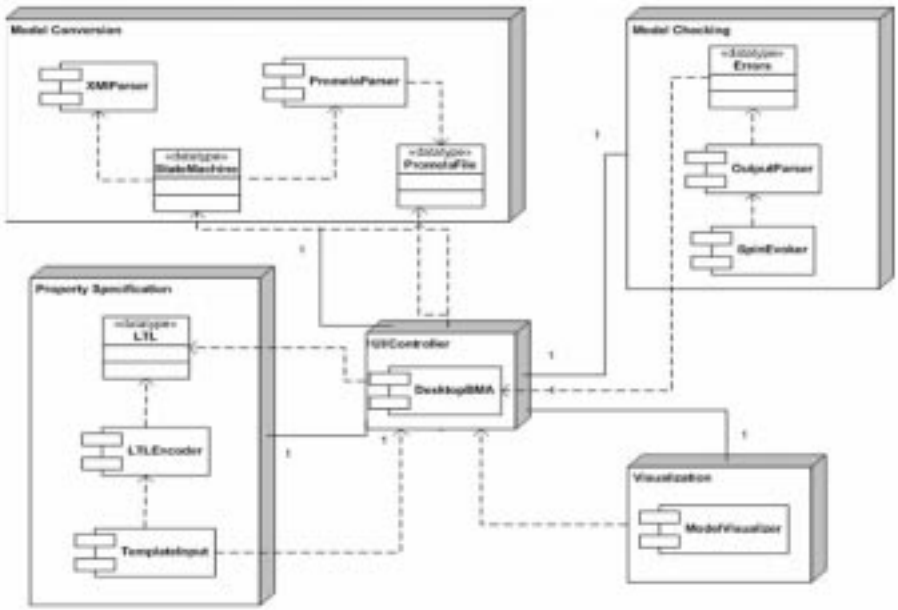

Fig. 4. The architectural design of the BMA.

templates, and launches the model checking process to find potential errors in the design.

When the student chooses a UML model as the input design model, the component *XMIParser* extracts the detailed model information from the input and stores it in a data structure called *StateMachine*. Concomittantly, the component *ModelVisualizer* generates the graph visualization and displays it to the student. *PromelaEncoder* converts the model in the *StateMachine* data structure into its corresponding PROMELA code skeleton. *DesktopBMA* initiates the *TemplateInput*, which provides the specialized user interface elements that allow the student to supply constraint templates.

The temporal logic generated from the templates is saved into a temporary text file as well. *SpinEvoker* enacts the model checker and calls *OutputParser* to extract errors present in the text-based results generated by SPIN. The errors are saved into a data structure called *Errors*. This data structure is then used by the *ModelVisualizer* to generate a colored trace of problematic states of the design to show the results of the model checking back to the student.

The BMA is developed in Java using the open source Eclipse IDE, and it is compiled using Sun's Java SDK 5.0.

All GUI components in the application are built using the Java Swing toolkit. The XMI files containing the UML diagrams are parsed using the open source Xerces XML SAX parser for Java. Graph-based visualizations in the application are generated by the Java Universal Network/Graph Toolkit (JUNG). The model checker SPIN is used to perform model checking.

THE UTILITY OF BMA IN TEACHING VERIFICATION-DRIVEN DESIGN

The BMA emulates the success of Test-Driven Assignments [9], as well as the promising proposals for improving testing skills [38] and integrating more testing into the curriculum [25]. Our strategy is to require students to submit verification queries in terms of BMA templates along with their UML statechart designs.

As part of their homework assignment, students are provided with a set of constraints their designs need to satisfy. They are asked to supply verification queries and demonstrate the correctness of their designs with respect to these queries. In evaluating submitted assignments we not only check the correctness of the design, but also students' verification performance. Verification performance is tested by running the verification queries against the instructor provided design models that are mutated to measure the fault-revealing quality of the student-supplied queries.

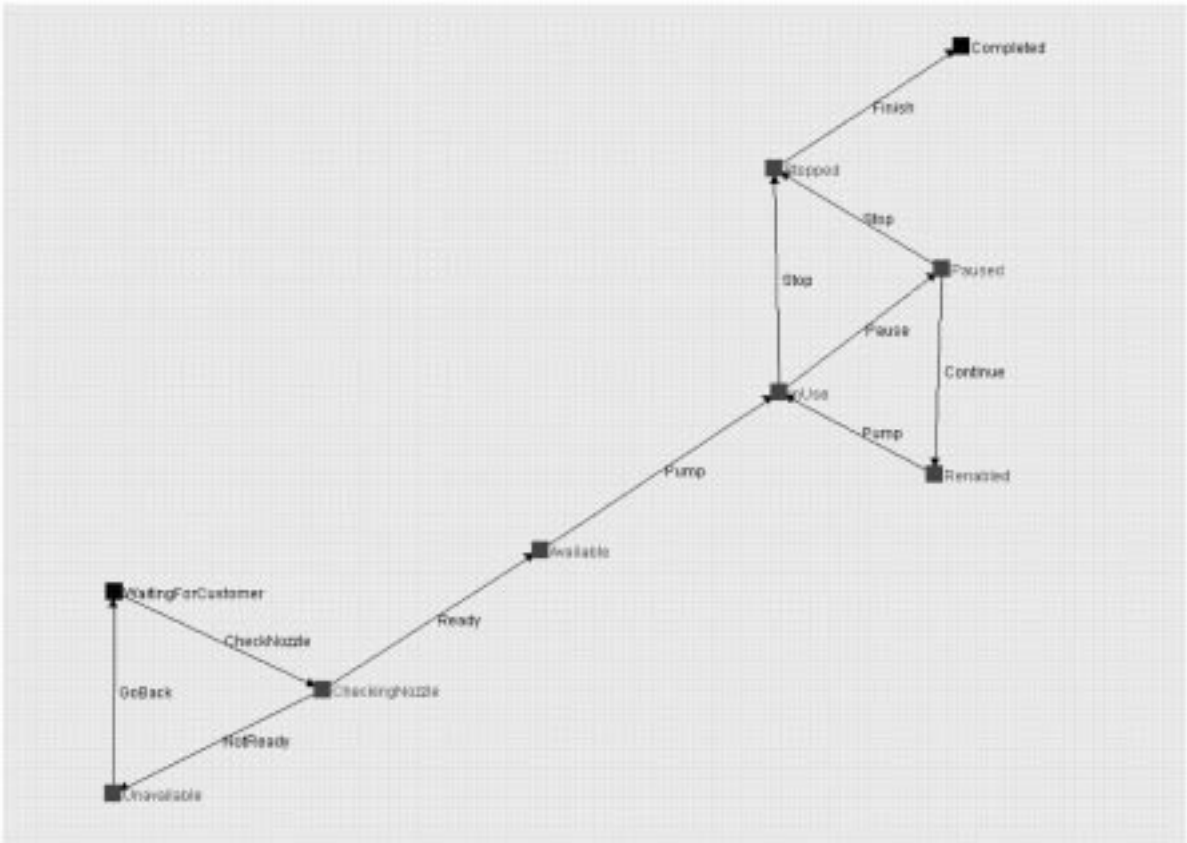

Fig. 5. BMA visualization for the gas pump statechart.

The case study

To illustrate the utility of the BMA, we use the design model of a simple gas pump that is created and submitted by a student. Figure 5 illustrates the BMA visualization of an input UML statechart.

The **gas pump model** provides a brief synopsis of the model's behavior. The gas pump starts in an idle start state waiting for a customer to interact with it. When a customer needs to use the pump by lifting a nozzle, the pump checks whether the nozzle is available to be used. If the nozzle is out of service, the pump indicates that it is unavailable and eventually goes back to the idle start state. If the nozzle is ready to be used and once the customer starts pumping gas into the car, it becomes in use. At any point during the pumping process, the customer can pause or stop the process to finish fueling the car. While the pump is paused, the customer can either continue or stop the fueling process.

One of the queries can be described as 'the *pumpstatus* will reach state *inuse* before *paused*', which considers the fact that a customer must always start the fueling process before s/he can pause the pump. This template can be divided into two logical propositions: one representing the status of the pump is currently *inuse*, the other is currently *paused*. The two propositions are joined using the scope operator *before* that is presented in the template list of the specification development interface shown in Fig. 6. In deriving the verification query, the instructor defines the two propositions as follows: The first is called *pumpinuse* with the logical expression '*pumpstatus == INUSE*' assigned to it. The second is called *pumppaused* with the logical expression '*pumpstatus == PAUSED*' assigned to it. We also specify that the scope operator of this template is *before*.

This indicates that *pumpinuse* should occur before *pumppaused*. As shown in Fig. 7, the model checking process does not yield any errors. The instructor tests the design with the rest of the verification queries to determine if the design is consistent with the constraints provided along with the problem definition. Next, the instructor uses the reference model for the problem to test the quality of the verification queries submitted by students. The model is injected with faults that violate explicitly stated problem constraints For instance, the instructor injects a fault to make sure that the transition *Pause* and the state '*Paused*' are not reachable during execution. In the scenario, the customer may complete the fueling process without pausing in the middle, thus the *pumpstatus* state variable may not be assigned to the state *paused*.

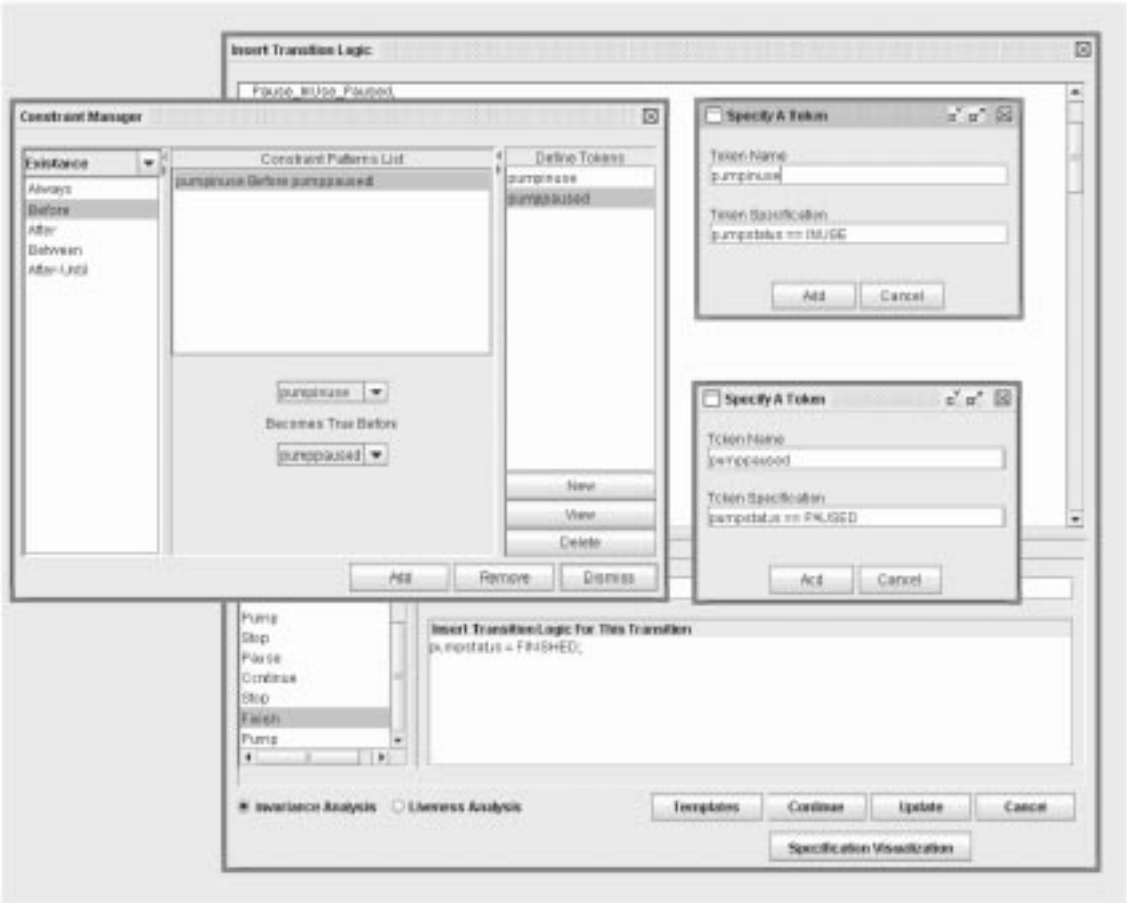

Fig. 6. Abstract constraint manager.

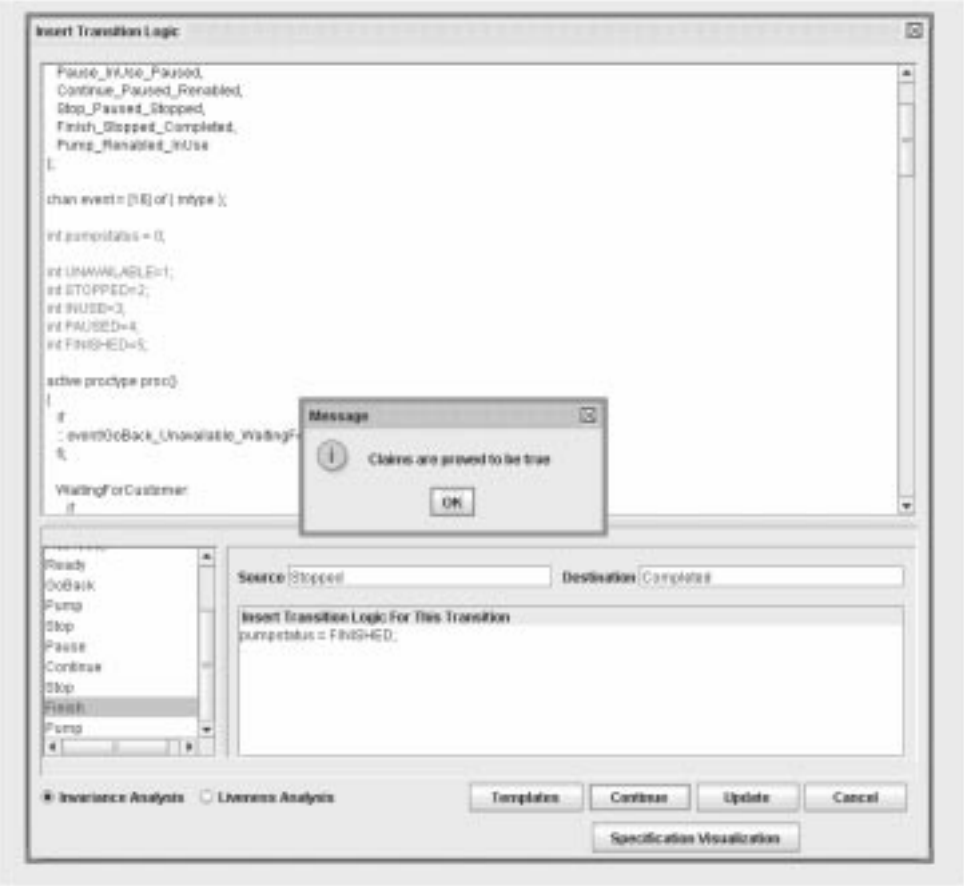

Fig. 7. Result of model checking.

To demonstrate this feature, we use the following specification template provided by the student: 'the *pumpstatus* will reach the state *paused* between the states *inuse* and *stopped*'. Creating the template in the same way as described above and performing model checking proves that the design model violates the specification and the colored error trace is shown in Fig. 8. The trace is shown in terms of sequence of states and transitions that lead to the state, which fails the specified property. The fault coverage of student queries is used to assess the quality of the verification queries they develop. This strategy is similar to the notion of fault-coverage in specification-based testing, where the quality of test cases is assessed based on the extent to which they cover injected faults in mutated software programs.

Preliminary observations on the utility of the Web-CAVE and the BMA

While a comprehensive survey or field study is not yet performed to evaluate the effectiveness of the Web-CAVE system and its BMA tool, 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 helped group members recognize integrity problems earlier to assure design coherency and correctness. A common criticism is the difficulty of the usability of the system, as well as the lack of clarity of the relevance of collected design metrics to quality objectives.

Also, the BMA tool is found by students useful in revealing discrepancies between state charts and sequence diagrams. However, the requirement for developing and submitting verification queries along with the assignments is found by students to be time consuming, since there is clearly a learning curve in using the verification pattern used by the BMA. Note, however, this effort is significantly less compared to learning to write formal specifications. On the other hand, students agree on the usefulness of the tool in asking pertinent questions about their designs, discovering and locating errors in models before submitting their assignments. Having instructor-supplied constraints and development of verification queries that aim to prove the satisfiability of these constraints improve their confidence in the grading scheme. As a side effect, the existence of the BMA tool and the inclusion of the reference

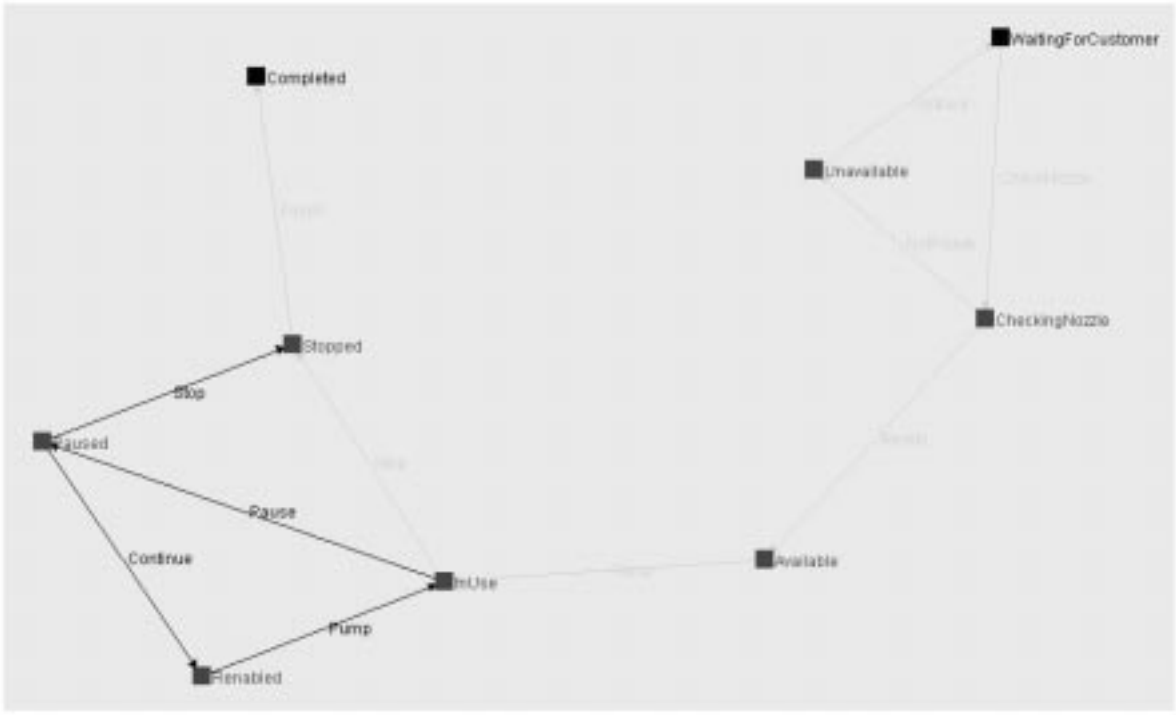

Fig. 8. Visualization of error trace in BMA.

verification queries provided by the instructor further ease the evaluation and grading of the submitted assignment. In effect, this observation revealed the potential of the BMA for becoming an automated grading system, through which design correctness, as well as verification skills of students can be tested in an objective manner.

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 MBV perspective to software design and modeling courses is discussed. The premise of the strategy is based on the observation that the 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 integration of MBV 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. 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.

REFERENCES

1. NIST, *The Economic Impacts of Inadequate Infrastructure for Software Testing,* NIST Planning Report 02-3, (2002) accessed May 21, 2005. http://www.nist.gov/director/prog-ofc/report02-3.pdf
2. Standish Group, *The Chaos Report* (1994), accessed April 30, 2005. http://www.standishgroup.com/sample_research/chaos_1994_1.php
3. OASIG, Why do IT Projects so Often Fail, *OR Newsletter*, **309,** 1996, pp. 12–16.
4. J. King, Survey shows common IT woes persist, *Computerworld* (2003), accessed April 30, 2005. http://www.computerworld.com/managementtopics /management/ story/0,10801,82404,00.html
5. C. Mann, Why software is so bad . . . and what's being done to fix it, *MSNBC Technology and Science*, 2002, accessed May 28, 2005. http://www.msnbc.com/news/768401.asp?0dm=C11LT&cp1=1
6. B. Lewis, The 70-percent failure, *Infoworld*, 2003, accessed April 30, 2005. http://www.infoworld.com/articles/op/xml/01/10/29/011029opsurvival.html
7. E. F. Barbosa, R. LeBlanc, M. Guzdial and C. J. Maldonado, The challenge of teaching software testing earlier into design, *Workshop Teaching of Software Testing (WTST),* February 7–9, 2003, Melbourne, Florida, pp. 27–33.
8. S. H. Edwards, Can quality graduate software engineering courses be delivered asynchronously on-line, *Proc. ICSE'2000,* (2001) pp. 676–679.

9. S. H. Edwards, Automatically assessing assignments that use test-driven development, *Workshop Teaching of Software Testing (WTST),* February 7–9, 2003, Melbourne, Florida.
10. D. Gluch and C. Weinstock, Model-Based Verification: A Technology for Dependable System Upgrade (CMU/SEI-98-TR-009, ADA 354756). Pittsburgh, Pa: Software Engineering Institute, Carnegie Mellon University (1998) accessed on May 21, 2005. http://www.sei.cmu.edu/publications/documents/98.reports/98tr009/98tr009abstract.html
11. MDA, The Architecture of Choice for a Changing World (2004). http://www.omg.org/mda/executive_overview.htm
12. J. Davies and A. Simpson, Teaching formal methods in context, *Proc. CoLogNET/FME Symposium, TFM2004*, LNCS 3294, pp. 185–202.
13. K. Robinson, Embedding formal development in software engineering, *Proc. CoLogNET/FME Symposium, TFM 2004*, LNCS 3294, pp. 32–46.
14. S. L. Duggins and B. B. Thomas, An historical investigation of graduate software engineering curriculum, *Proc. 15th Conf. Software Engineering Education and Training (CSEET),* Kentucky, USA, February 25–27, 2002.
15. M. Sebern and M. Lutz, Developing undergraduate software engineering programs, *Conf. Software Engineering Education and Training*, 6–8 March 2000, Austin, Texas, USA, pp. 305–306.
16. D. Gries, *The Science of Programming,* Springer-Verlag, New York (1981).
17. D. L. Parnas, Education for computing professionals, *Teaching and Learning Formal Methods* (eds Dean and Hinchey) Academic Press (1996).
18. L. V. Almstrum, Investigating student difficulties with mathematical logic, *Teaching and Learning Formal Methods* (eds. Dean and Hinchey) Academic Press (1996).
19. D. Garlan, Effective formal methods education for professional software engineers, *Teaching and Learning Formal Methods* (eds Dean and Hinchey) Academic Press (1996).
20. E. Wang, R. Wirtz and M. Greiner, Simulating Corporate Project Engineering for Freshmen, *Proc. Frontiers in Education Conference,* November, pp. 1313–1318 (1998).
21. D. Gluch and J. Brockway, An Introduction to Software Engineering Practices Using Model-Based Verification (CMU/SEI-99-TR-005, ESC-TR-99-005). Pittsburgh, Pa.: Software Engineering Institute, Carnegie Mellon University, 1999. [accessed on may 21, 2005 from http://www.sei.cmu.edu/publications/documents/99.reports/99tr005/99tr005abstract.html]. (1999).
22. G. Booch, J. Rumbaugh, and I. Jacobson, *The Unified Modeling Language User Guide,* Addison-Wesley. (1999).
23. J. Wing, A specifier's introduction to formal methods, *IEEE Computer*, **23**(9), 1990, pp. 8–24.
24. A. Diller, *Z: An Introduction to Formal Methods* (2nd Ed.), John-Wiley & Sons (1994).
25. C. Jones, *Systematic Software Development Using VDM*, Prentice-Hall Intenational Series in Computer Science, Hemel Hempstead (1986).
26. D. Garlan, M. Shaw, C. Okasaki, C. Scott and R. Swonger, Experience with a course on architectures for software systems, *Proc. SEI Conf. Software Engineering Education,* Springer Verlag, LNCS 376, October 1992. Also available as CMU/SEI technical report, CMU/SEI-92-TR-17 (1992).
27. D. Garlan, Integrating formal methods into a professional master of software engineering program, *Proc. Z Users Meeting,* June 1994.
28. J. N. Reed and J. E. Sinclair, motivating study of formal methods in the classroom, *Proc. CoLogNET/FME Symp., TFM 2004,* LNCS 3294, 203-213. (2004).
29. M. Clarke, E. Emerson and A. Sistla, Automatic Verification of finite state concurrent systems using temporal logic specifications, *ACM Transactions on Programming Languages and Systems*, **8**(2), 1986, pp. 244–263.
30. M. Clarke, O. Grumberg and D. E. Long, Model checking and abstraction, *ACM Trans. Programming Languages and Systems,* **16**(5), 1994, pp. 1512–1542.
31. E. M. Clarke, O. Grumberg and O. Peled, *Model Checking,* MIT Press (2000).
32. J. E. Stice, *Developing Critical Thinking and Problem-Solving Abilities,* Jossey-Bass Inc., San Francisco, CA (1987).
33. R. J. Daigle, M. V. Doran, and J. H. Pardue, Integrating Collaborative Problem Solving throughout the Curriculum, *Proc. 27th SIGCSE Technical Symposium on Computer Science Education,* 1996, pp. 237–241.
34. M. J. Oudshoorn and K. J. Maciunas, Experience with a project-based approach to teaching software engineering, *Proc. Southeast Asian Regional Computer Confederation 5th Annual Working Conf. Software Engineering Education,* Dunedin, New Zealand, November 1994, pp. 220–225.
35. L. Yilmaz and S. Wang, Integrating model-based verification into software design education, *J. Science, Technology, Engineering, and Math Education* (in press).
36. J. A. Hall, Seven myths of formal methods, *IEEE Software,* **7**(5), September 1990, pp. 11–19.
37. M. B. Dwyer, G. S. Avrunin and J. C. Corbett, Patterns in property specifications for finite-state verification, *Proc. 21st International Conference on Software Engineering,* May, 1999.
38. T. Shepard, M. Lamb and D. Kelly, More testing should be taught, *Communications of the ACM,* **44**(6), 2001, pp. 103–108.

Shuo Wang is graduate student of Computer Science and Software Engineering in College of Engineering at Auburn University. He earned his BS 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.

Levent Yilmaz is assistant professor of Computer Science and Software Engineering in the College of Engineering at Auburn University. Dr. Yilmaz earned his Ph.D. and M.S.

degrees from Virginia Tech. He received his B.S. degree in Computer Engineering from Bilkent University, in Turkey. He worked as a lead project engineer and principle investigator for advanced simulation methodology, model-based verification, and simulation interoperation technology development efforts. His research interests are on software engineering and simulation modeling education, advancing the theory and methodology of simulation modeling, and agent-directed simulation (to explore and understand socio-technical systems such as software process and project dynamics). He is a member of ACM, IEEE Computer Society, Society for Computer Simulation International, and Upsilon Pi Epsilon.