

A Graduate Seminar in Tools and Techniques

Patrick J. Graydon, Elisabeth A. Strunk, M. Anthony Aiello, and John C. Knight

Department of Computer Science
University of Virginia
151 Engineer's Way, P.O. Box 400740
Charlottesville, VA 22904-4740, USA
{graydon, strunk, aiello, knight}@cs.virginia.edu

Introduction

In the spring of 2006 we offered a graduate seminar designed to introduce graduate students to the tools and techniques that are being used to construct dependable systems. Our target audience was a group of graduate students who are or will be conducting research in software engineering, particularly in the dependable systems arena. We had several goals with this course. First, we wanted to familiarize these students with the state of the art and practice so that they would be better able to make research contributions. Second, we wanted to help the students learn about scientific critique by asking them to assess the strengths and weaknesses of the tools, both through review of the literature written about the tools and through hands-on experience. Finally, we wanted them to compare some of the tools within this class as an initial effort in helping both researchers and practitioners choose which tools are most suited for the kinds of problems encountered in our profession.

We chose to focus on two distinct topic areas: model checking and model-based development. In either area the students had much to learn, and so either could have been the focus of an entire course. We chose to cover both primarily because these are both important topic areas and secondarily to determine the feasibility of adequately covering both areas in one course. We believe this was a good approach to take, because there is some overlap between the two areas, and because the difference between them underlines the need to assess the suitability of particular analysis techniques when solving problems.

We did not intend to make the students expert, or indeed proficient, in the use of any one model checking or model-based development tool. Rather, we aimed to make students familiar with the technical concepts behind the tools we covered, the kinds of problems to which each tool is applicable, and the power, capability, and limitations of each tool. Each student chose one particular tool to study in more depth than the others, providing the class with insight into the tool and providing general knowledge about the motivation behind that tool. Then the class as a whole learned what the student experts taught, and discussed each student's tool in isolation and with respect to the others. In this way, we wanted to provoke thought and discussion about how and when the tools should be used, the ways in which each tool succeeded or failed, and how the tools compared to each other.

Course outline

At the beginning of the course, each student selected a model checker or model-based development tool. He or she then became solely responsible for presenting it to the other students throughout the course. Students chose the following tools:

- **SLAM** A model checker for device drivers developed by Microsoft [1].
- **BLAST** A model checker for code-level properties developed at the University of California, Berkeley [2].
- **Kronos** A model checker for complex real-time systems developed at Verimag [3].
- **Spin** A model checker for concurrent systems originally developed at Bell Labs [4].
- **SCRtool** A specification tool developed at the Naval Research Laboratory [5].
- **Perfect Developer** A design-by-contract development tool from Escher Technologies [6].
- **SCADE** A model-based development tool produced by Esterel Technologies [7].
- **Simulink** A model-based development tool from The MathWorks [8].

We divided the course into three phases: (1) presentations; (2) laboratory exercises; and (3) comparison discussions. Our course was scheduled for two 75-minute meetings per week over a 15-week semester: the presentation phase occupied the first 9 weeks; the laboratory exercise phase occupied the next 4 weeks, and the comparison presentation phase occupied the final 2 weeks of the course.

In the presentation phase, each student gave a 75-minute presentation in class to familiarize the other students with his or her chosen tool. These initial presentations were intended to convey the overall purpose, organization, and capability of each tool. During each presentation, we encouraged all students to propose questions that would help to evaluate the purpose, capability, and limitation of the tool being presented. In order to prepare students in the class for a detailed presentation and to ask probing questions, the presenting student selected and assigned pre-reading material from the available literature.

In the laboratory phase, each student prepared and directed a 75-minute laboratory session designed to give the other students the opportunity to get hands-on experience with his or her chosen tool. Students were instructed to prepare laboratory exercises that familiarized other students with the tool's user interface, the process of using the tool, and the tool's merits.

The comparison phase was split into two parts: model checking and model-based development. For each part, the students who had chosen that type of tool collectively prepared a presentation comparing and contrasting their tools, which was followed by general discussion lasting the remainder of the 75-minute course period.

The presentations, laboratory exercises, and comparison discussions did not occupy all of the scheduled class meetings. In order to fill the remaining meetings, we scheduled discussions of related papers and guest lectures on related topics. During the presentation phase in particular, we used this technique to give students time to acquire and study their tools.

Students in the course were asked to prepare both a laboratory write-up and a paper describing and analyzing their chosen tool. The laboratory write-ups included a description of the laboratory and all of the materials used in it. The papers were required to be at least 7 pages in length and to include: (1) a description of the tool and its developers; (2) a summary of the problem it was created to address; (3) a discussion of its capabilities; (4) an analysis of the problems to which it is suited and unsuited; and (5) a comparison with the other tools studied in class. Course grades were based upon participation (5%), the presentation (10%), laboratory draft (10%), final laboratory write-up (30%), draft project report (10%), and final project report (35%).

Outside of class, each student investigated his or her chosen tool, prepared a presentation on it, read assigned reading, prepared a laboratory session, and wrote a project report. Of the students responding to a questionnaire distributed at the end of the course, 2 reported spending 1-3 hours per week on these tasks outside of class, 1 reported spending 4-6 hours, and 2 reported spending 7-9 hours.

Some of the tools we examined were not freely available, but in most cases we were able to get the tool's manufacturer to allow us to use the tool in the course without cost. In the remaining cases we located a similar, freely available tool and asked the student present that instead.

An example course unit: SCRtool

As an example of the approach that we used, we describe one student's efforts to present SCRtool, the prototype tool built by the Naval Research Laboratory (NRL) to support the Software Cost Reduction (SCR) tabular notation. Note that the preparation of the material described in this section was part of the class itself. The student who presented the SCRtool (one of us, Graydon) developed the complete presentation and laboratory, and participated in the preparation of the discussion.

A product of research at the NRL by Heitmeyer et. al., SCRtool is a prototype tool intended to allow software engineers to create concise, unambiguous requirements specifications for embedded systems. The tool is designed around the tabular specification notation developed by Heninger and Parnas as part of the SCR efforts.

The presentation

Before the presenting this work, we asked students to read Heitmeyer's "Managing Complexity in Software Development with Formally Based Tools" [1] and Heninger's "Specifying Software Requirements for Complex Systems: New Techniques and Their Applications" [10]. We selected the former paper because it provides a short, accessible description of the SCRtool team's vision for tools. We chose the latter because it describes

both a forebear of the tabular notation used by SCRtool, and because it illustrates the thinking that underlies that tabular notation. There are numerous papers on the SCRtool and the theorem-proving technology of which it makes use, but we felt that the papers we selected provided a good introduction to the kinds of problems the SCRtool is intended for use with and the overall specification approach advocated by its authors.

Because the tool’s authors were in the process of preparing a new version of SCRtool for release at the time of the presentation and suggested that we wait for this version rather than use an older one, we elected to focus the presentation on the tabular SCR notation rather than the tool itself. We were able to obtain the new version after the presentation.

The laboratory exercise

We wanted to expose each student to the main features of the SCRtool. To accomplish this, we designed the laboratory activity in which each student would be required to complete and validate a simple specification using those features. The example specification was for an embedded controller for an automatic Japanese-style bath tub with built-in heating and automatic filling and draining. The laboratory handout given to students described the system’s purpose and intended behavior, enumerated the system’s sensors and actuators, and then guided the students through the process of using SCRtool to complete and validate the specification. In order to give students examples to work from and to limit the amount of work needed to complete the activity, the laboratory materials included a partially-completed specification.

Monitored and controlled variables. SCR specifications model the world in terms of monitored variables, which represent input from sensors, and controlled variables, which represent output to actuators. Software is modeled as a collection of continuous and demand functions that take input from monitored variables and determine the value of controlled variables. In order to expose students to SCRtool’s specification editor we asked them to add variable declarations to the partially-completed specification as needed to complete it. Since the laboratory handout listed all of the system’s sensors and actuators, and since we included all of the needed variable types in the partially-complete specification, students needed only to add variables representing a small number of actuators.

Modes and mode transition tables. In order to simplify function description, SCR specifications declare mode classes, each of which divides system state into a number of modes. As monitored variables change, mode transition tables dictate the resulting effect upon the system’s modes.

The partially-complete specification included a mode class with all of the modes needed to describe the system and the incomplete mode transition table shown in Table 1. In the lab handout, we explained that in the SCR event notation the expression $@T(x)$ specifies the moment when x becomes true, introduced the optional

Table 1: Mode transition table provided to students

Source Mode	Events	Destination Mode
Off	@T(PowerButton = Depressed)	Filling
Filling	@T(WaterLevel = Full) WHEN (Temperature < SetTemperature)	Heating
Filling	@T(WaterLevel = Full) WHEN (Temperature >= SetTemperature)	Ready
Heating	<i>Left for students to complete.</i>	Ready
Heating, Ready	<i>Left for students to complete.</i>	Filling
Ready	<i>Left for students to complete.</i>	ShuttingDown
Ready	<i>Left for students to complete.</i>	Heating
ShuttingDown	<i>Left for students to complete.</i>	Off

WHEN y guard clause that specifies the conditions under which the event causes the mode transition, and asked the students to use their knowledge of the system’s intended functionality to complete the table.

Condition and event tables. SCR specifications model continuous and demand functions using condition and event tables, respectively. Each kind of table specifies the values taken on by one controlled variable under different modes. Rows in both tables represent modes, and columns in both table represent values. In condition tables cells contain logical tests of monitored variable values such that if a cell in the row representing the current mode evaluates to true, the controlled variable will have the value represented by that cell’s column. In event tables cells contain event expressions, so that when the event in the row representing the current mode occurs, the controlled variable is assigned the value corresponding to the cell’s column.

In the partially-complete specification, we included an example condition table and an example event table. Students were directed to examine these and then create tables defining the value of the remaining controlled variables.

Table 2: Sample event table provided for students

Modes	Conditions	
Off, Filling, Heating, Ready, ShuttingDown	@T(UpButton = Depressed) WHEN (SetTemperature < 50)	@T(DownButton = Depressed) WHEN (SetTemperature > 25)
SetTemperature =	SetTemperature + 1	SetTemperature - 1

Checking. SCRtool provides built-in checking for disjointness, coverage, and type correctness. Disjointness and coverage testing ensure that condition and event tables specify exactly one value for the variables they define under all circumstances. Type checking catches syntax errors in table cells, initial conditions that are inconsistent with function definitions, uses of a variable inconsistent with its type, and the like. These checks are fully automatic, take only seconds, and can be started by clicking one tool-bar icon. We wanted to demonstrate the power and simplicity of these checks, so we asked students to check their specifications after they deemed them complete. We did not introduce any deliberate errors into the provided specification, but, since students made mistakes when completing the specification, they were able to see the checker’s power and error-reporting mechanism in action.

Assertions. SCRtool permits specification developers to declare assertions representing properties that must hold at all times and provides an automatic theorem prover for proving these assertions. We asked students to write an assertion representing a safety property for the system and use the tool’s built-in theorem prover to prove it. Theorem proving in SCRtool is a deliberately push-button activity, and we wanted to demonstrate how accessible it is to people unfamiliar with the underlying prover and its strategies.

Unfortunately the standard built-in prover was unable to prove the property we specified quickly enough for all students to be able to complete this part of the laboratory exercise. Because the prover took fifteen minutes or more, depending upon the exact formulation of the property and the speed of the computer, only one out of six groups managed to prove the property during the 75-minute laboratory period. Although our example’s illustration of the runtime this prover requires is arguably instructive, were we to repeat this laboratory activity, we would chose an example safety property that could be proved by the default prover in seconds so that we could use our limited lab time to expose the students to as many aspects of the tool as possible.

Simulation. While students were waiting for the theorem prover, we demonstrated the use of SCRtool’s simulator using an instructor’s machine. Again, we wanted to demonstrate that the simulator is both automatic and simple. Clicking one tool-bar icon causes the system to generate and launch a basic simulator that allows the user to input monitored variable values and changes using standard dialog-box controls and observe the system’s reactions.

Results of comparison discussion

After the laboratory activities, the two groups of students—those who chose model checkers and those who chose model-based development tools—each prepared a short presentation comparing and contrasting the tools. Each presentation was followed by a discussion involving the whole class. The presentation on model-based development tools raised the observations summarized below. We include this list of observations to illustrate the level of comprehension of the technical area, the tools, and their capabilities achieved by the students. The details of the observations are not especially significant.

The kind of software the tool is intended for. SCRtool is intended for the specification of embedded control systems. Perfect Developer and SCADE deliberately target safety-related systems with stringent dependability requirements. Simulink focuses on dynamic systems such as signal processing and communications systems.

The kind of developer the tool is aimed at. SCRtool is intended for use by software engineers with no particular discrete math or formal methods skills. Perfect Developer, in contrast, is aimed at software engineers with strong discrete math skills. Simulink and SCADE are intended for use by control engineers, not software engineers, and so describe systems using signal diagrams.

The tool's limitations. The present-generation SCRtool does not support any collection type such as an array, list, set, or map, and so cannot be used to specify systems whose state includes such a collection. (This capability may be added to a future release.) Perfect Developer requires the developer to write procedural code but offers no way to express concurrency or synchronization. Simulink lacks both a precisely defined semantics and theorem-proving capability. SCADE, like Perfect Developer, offers no way to express concurrency. None of these tools offers the ability to elegantly specify or analyze real-time behavior.

The guarantees made by the tool. SCRtool can guarantee that specifications are complete (in the sense that all tables are fully specified) and unambiguous, and can prove assertions. Perfect Developer and SCADE can guarantee that the implementation matches the specification. Simulink offers only simulation capabilities.

The V&V activities supported by the tool. All of the tools we studied support simulation. SCRtool and SCADE also offer proof of assertions. Simulink checks assertions during simulation.

The tool's code-generation capabilities. SCRtool generates Java code, but this code is intended for simulation rather than production use. Perfect Developer can generate C, Java or Ada code. Simulink can generate C or Ada. SCADE can generate C, Ada, and SPARK Ada.

The tool's usability. SCRtool's GUI needs refinement, but the specification model is simple. Perfect Developer has a much steeper learning curve and its documentation could be enhanced by a context-sensitive lookup feature. Simulink has an intuitive interface, good documentation, and the solid support of its makers. SCADE likewise has an intuitive user, although some students complained that the GUI was so cluttered as to leave little room for diagrams and that the program did not draw and flow wires cleanly.

The tool's scalability. Citing literature, students pointed out that SCR has been used on sizeable projects including thousands of tables. Perfect Developer can handle as much code as developers are willing to write. Simulink and SCADE allow functional blocks to be composed so that the size of a project is limited only by the system's available memory.

Conclusion

We have presented details of a course taught at the University of Virginia designed to introduce students to a variety of elements of formal methods by examination of state-of-the-art tools. The approach used was to have each student in the class study a single tool, and for that student to: (1) present a lecture on the tool; (2) to conduct a laboratory exercise in the use of the tool; and (3) to create a final report analyzing the tool's strengths and weaknesses. The goal was to familiarize the students in the class with a variety of tools and to encourage them in critical thinking in the application of formal methods by asking them to assess the suitability of particular techniques.

While we have no sure way of telling to what extent the course met its goals, we feel that it was a success. The comments and course reviews from the members of the class were positive. The final reports were overall very high quality, and the comparisons of the different tools were particularly impressive. The classification scheme for dimensions of applicability of model-based development tools are described above; the classification scheme for the model checking tools was equally impressive. All of the courseware that was developed (presentations, laboratories, summary discussions, tool reports) are available from the authors.

Finally, we were pleased with the interest the students showed in teaching others. We presented the option to emphasize the laboratory activities to the students as a choice that they could make, telling them that we would reduce the work required for the final report somewhat if they chose to put effort into developing laboratory exercises. The feedback for this option was very strong, with no one objecting to it (other than expressing concerns about workload) and several students enthusiastically supporting it. We feel that there is significant potential for combining results from future similar courses, at the University of Virginia and perhaps at other universities, to start to develop a more comprehensive body of knowledge on how to teach the state of the art to senior undergraduates as an overall technology transfer initiative.

Acknowledgements

We thank Ralph Jeffords of the Naval Research Laboratory for his assistance in obtaining and installing SCRtool at the University of Virginia; and the other tool vendors—Escher Technologies, Esterel Technologies, iLogix, and the MathWorks—for allowing us to use their tools. We also thank all of the students who took this class for their efforts and support for the idea of trying this approach to introducing formal methods. We thank Kendra Schmid, Michael Spiegel, and Benjamin Taitelbaum in particular for their comments on SCADE, Perfect Developer, and Simulink, respectively.

References

- [1] “SLAM Project.” Internet: <http://research.microsoft.com/slam>, [21 July 2006].
- [2] “BLAST.” Internet: <http://embedded.eecs.berkeley.edu/blast>, [21 July 2006].
- [3] “Kronos Home Page.” Internet: <http://www-verimag.imag.fr/TEMPORISE/kronos>, [21 July 2006].
- [4] “Spin - Formal Verification.” Internet: <http://spinroot.com/spin/whatispin.html>, [14 July 2006].
- [5] “Automatic Construction of High Assurance Systems from Requirements Specifications.” Internet: <http://chacs.nrl.navy.mil/personnel/heimmeyer.html>, [21 July 2006].
- [6] “Escher Technologies - Products.” Internet: <http://www.eschertech.com/products/index.php>, [21 July 2006].
- [7] “SCADE Suite :: Products.” Internet: <http://www.esterel-technologies.com/products/scade-suite/overview.html>, [21 July 2006].
- [8] “The MathWorks - Simulink® - Simulation and Model-Based Design.” Internet: <http://www.mathworks.com/products/simulink>, [21 July 2006].
- [9] C. Heitmeyer. “Managing Complexity in Software Development with Formally Based Tools.” *Electronic Notes in Theoretical Computer Science*, vol. 108, 2004.
- [10] K. Heninger. “Specifying Software Requirements for Complex Systems: New Techniques and Their Application.” *IEEE Transactions on Software Engineering*, vol. SE-6, no. 1, January 1980.