AC 2008-826: FORMAL METHODS IN THE UNDERGRADUATE SOFTWARE ENGINEERING CURRICULUM

Mark Sebern, Milwaukee School of Engineering

MARK J. SEBERN is a Professor in the Electrical Engineering and Computer Science Department at the Milwaukee School of Engineering (MSOE), and was the founding program director for MSOE's undergraduate software engineering program. He has served as an ABET program evaluator for software engineering, computer engineering, and computer science.

Henry Welch, Milwaukee School of Engineering

HENRY L. WELCH is a Professor of computer and software engineering in the Electrical Engineering and Computer Science Department at the Milwaukee School of Engineering. He teaches a broad range of course in both the computer and software engineering programs ranging from embedded systems to computer graphics, artificial intelligence, and formal methods.

Formal Methods in the Undergraduate Software Engineering Curriculum

Mark J. Sebern, PhD, PE
Milwaukee School of Engineering
sebern@msoe.edu

Henry L. Welch, PhD
Milwaukee School of Engineering
welch@msoe.edu

www.msoe.edu/se/

Abstract

An informal survey of undergraduate software engineering curricula indicates that many such programs incorporate a course in formal methods. Feedback from industry partners and advisory committees, however, seems to suggest that formal modeling, analysis, and verification techniques are not routinely employed in many software development organizations. Software engineering educators, especially those focused on preparing undergraduate students for practice in the discipline, encounter a number of issues when they take on the task of teaching and applying formal methods, including lack of appreciation by undergraduate students of the potential value and applicability of these techniques, and lack of realistic industry examples or of textbooks and curricular materials that address real-world software engineering practice. In addition, reliable, capable, and well supported tools are difficult to find; some of those that are available do not integrate well into contemporary software development processes.

This paper describes the definition and evolution of one such formal methods course in an undergraduate software engineering curriculum. The discussion includes available notations and languages, tool support, integration of other program outcomes, student feedback, and instructor preparation. Audience participation and discussion are encouraged.

Introduction

The term "formal methods" is generally understood to mean "mathematically based languages, techniques, and tools for specifying and verifying [software] systems" [3]. These approaches often incorporate modeling (functional behavior, timing behavior, performance, or structure), model checking, and theorem proving.

Formal methods topics are included in the SEEK body of knowledge presented in the SE2004 curriculum guidelines [12], along with an outline for a model course (SE313) on the use of formal methods in software engineering. Beyond the mathematical foundations, the curricular guidelines focus on modeling, specification, validation, notation, and tools. A survey of published curricula for the fifteen ABET-accredited software engineering programs (as of January 2008) indicates that at least six programs require a separate course in formal methods. Some other programs offer an elective course or other courses with some formal methods content.

At the Milwaukee School of Engineering (MSOE), the undergraduate software engineering curriculum has included a junior-level course in formal methods (initially SE-381, later SE-3811) that has run every academic year since its initial offering in the fall quarter of 2000. The focus of this course has been on the use of formal modeling and specification, notation and tool usage, and (to the extent possible) the application of formal methods to practical software development. The details of this experience are discussed in the sections that follow.

Z Notation

There are a large number of notations and languages, with each formal methods research group or center offering at least a few original contributions or variations on work done by others [8]. One well known option is the Z (pronounced "zed") notation developed at the Programming Research Group at Oxford University, for which a number of text and reference books are available [5,10,17,18]. Z is based on sets and predicate logic and uses mathematical notation to specify systems and changes of state.

To illustrate how Z can be used in software specification, Figure 1 shows a simple example of a state schema for part of an academic course scheduling system. The upper portion of the schema contains declarations, which in this case correspond roughly to the attributes or associations of a class named **Schedule**. The arrow symbols designate various types of relations; for example, the **prof_dept** variable represents a function that associates each professor with a single department. The lower portion of the schema contains predicates, which represent invariant conditions that should hold for all class instances at all times. For example, all elements in the range of the section-to-course function are required to be members of the set of known courses.

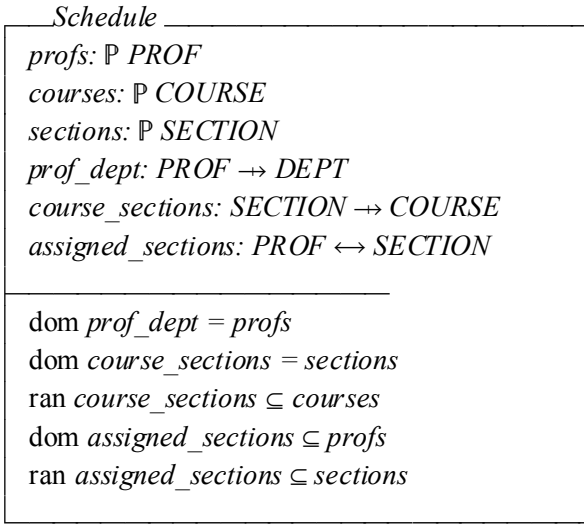

Figure 1. Example Z state schema

Operation schemas are used to specify changes of state, and correspond to method calls. Figure 2 illustrates a simple operation to delete a professor from the system. The **ΔSchedule** declaration at the top serves to include the **Schedule** state schema, but also to create "before" (unprimed) and "after" (primed) states of the state schema elements. Input and output parameters are also declared in this section of the operation schema. By convention, input parameters have names ending in a question mark (e.g., **prof?**) and output parameter names end with an exclamation point.

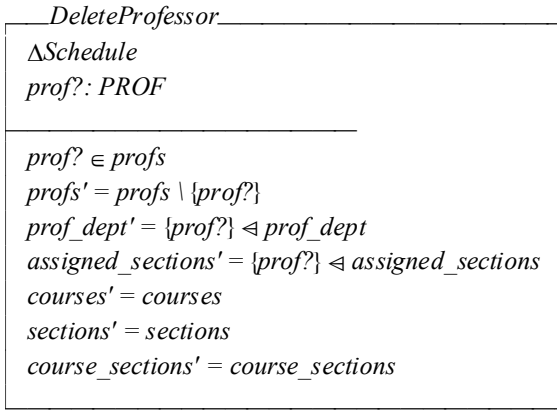

Figure 2. Example Z operation schema

The predicates in the lower portion of the operation schema specify the operation preconditions (e.g., the professor in question must be in the set of known professors), the changes that result from the operation, and explicit specification of the elements that do not change (are the same in the "before" and "after" states).

Of course, the operation specified in Figure 2 can only succeed if the precondition is met. For this reason, it is common to define additional operation schemas to handle exception conditions, as in Figure 3. In this case, the **ΞSchedule** declaration incorporates "before" and "after" versions of the **Schedule** state schema, but also automatically includes predicates to guarantee that the "before" and "after" versions are the same. In other words, this operation is specified not to change the system state. The **stat!** output variable is used to return an error indication.

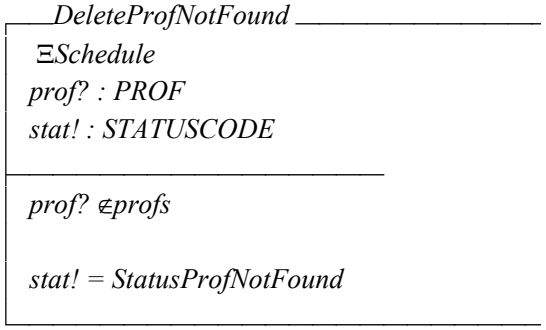

Figure 3. Operation schema for exceptional condition

Since we now have operation schemas to handle both the normal and exceptional cases, we can compose a total operation that will work in either case, using the schema calculus definition of Figure 4. The **Success** schema referenced here simply specifies that the **stat!** output variable has a value that indicates successful completion of the operation.

$$TotalDeleteProfessor \cong (DeleteProfessor \wedge Success) \vee DeleteProfNotFound$$

Figure 4. Complete operation schema defined with schema calculus

A typical Z specification document is prepared in LaTeX, and combines formal Z notation with informal textual descriptions. An example of the "source" format for a Z specification is shown in Figure 5.

```
\begin{schema}{Schedule}
profs : \power PROF \\
courses : \power COURSE \\
sections : \power SECTION \\
prof\_dept : PROF \pfun DEPT \\
course\_sections : SECTION \pfun COURSE \\
assigned\_sections : PROF \rel SECTION
\where
\dom prof\_dept = profs \\
\dom course\_sections = sections \\
\ran course\_sections \subseteq courses \\
\dom assigned\_sections \subseteq profs \\
\ran assigned\_sections \subseteq sections
\end{schema}
```

Figure 5. LaTeX "source code" for schema of Figure 1. Example Z state schema

Beyond document preparation, effective use of a formal notation like Z requires additional tool support. One such tool, Z/EVES, is a syntax checker and theorem prover The Z "source code" is imported into Z/EVES, as shown in Figure 6, ignoring the informal text and keeping only the Z specification elements.

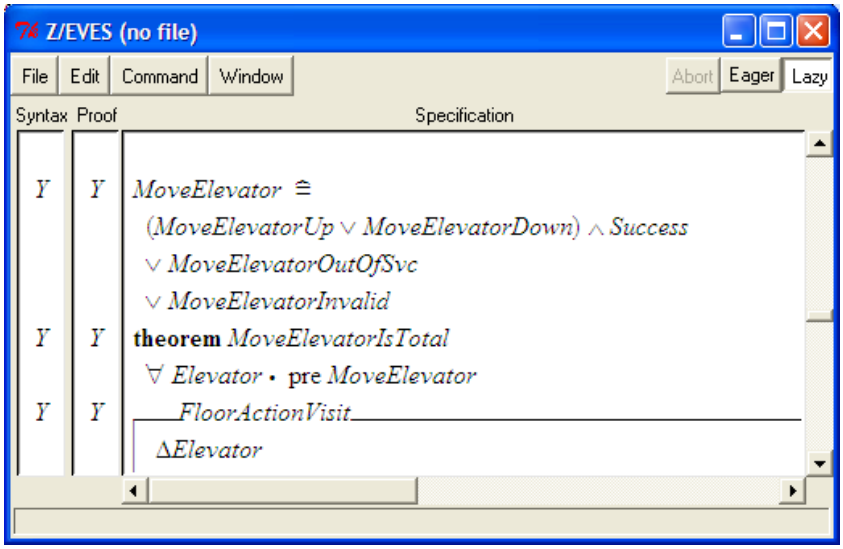

Figure 6. Z specification in Z/EVES checker and theorem prover tool

The theorem prover of Z/EVES is guided by a user-provided proof script and can be used to demonstrate that important assertions about the specification are in fact correct. An example of a Z/EVES proof is shown in Figure 7. Here, an operation totality proof is used to confirm that all relevant cases of an operation have been handled.

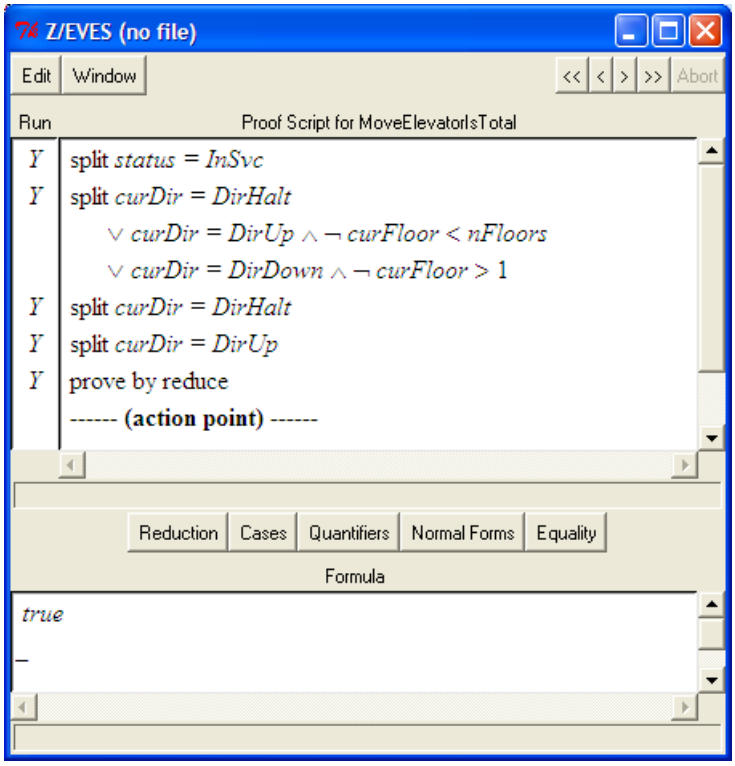

Figure 7. Proving a theorem using Z/EVES

For undergraduate software engineering students, the use of a formal specification notation and a theorem prover is definitely an acquired taste. However, most of these students are able to use these tools successfully to model software systems of moderate complexity, and it is not uncommon for them to discover defects in their specifications or designs when they attempt to prove important assertions.

Java Modeling Language (JML)

One difficulty with a notation like Z, despite its power and benefits, is that it does not link directly to common programming languages or development environments. For this reason, the MSOE formal methods course has in recent years incorporated the Java Modeling Language (JML) [2,4,11,13]. In JML, formal specifications are written directly into Java source code, using special comment syntax. An example of a student's JML method specification is shown in Figure 8. Note that this specification can detail both normal and exceptional behavior, to make sure that all possible cases are covered.

```
public class SimpleList {
      /*@
       @ spec_public
       @ non_null
       @*/
      private Integer[] list;
. . .
      /*@
        @ public normal_behavior
        @  requires size() > 0;
        @  assignable \nothing;
        @  ensures \result != null;
        @ also
        @ public exceptional_behavior
        @  requires size() < 1;
        @  assignable \nothing;
        @  signals_only NoSuchElementException;
        @ pure
        @*/
      public Integer getFirst() throws NoSuchElementException {
            . . .
      }
}
```

Figure 8. JML specification for a linked-list class method

In JML, specification elements may be added at the class, attribute, or method level. Unlike Z, predicates are for the most part expressed using Java language syntax and semantics, augmented with special keywords (including universal and existential quantifiers, for example). Because of their textual format, JML specifications can be added directly to Java source code, rather than being maintained in a separate document.

In order to support reuse and software libraries, JML specifications for new classes can make use of previously created specifications for library and support classes. It is also possible to create abstract specifications that are not directly tied to actual code, such as generic specifications for common data structures, and to use them as a foundation for specifications of concrete realizations.

One use of JML specifications is to support runtime assertion checking (RAC). In this case, a special JML compiler processes the Java source code, inserting additional code at critical points such as method entries and exits. The resulting Java class files incorporate byte code that represents both the original code and the runtime checking; when executed, any violations of the JML specification are reported by means of JML-specific exceptions. Since the code must be executed in order for the specification checking to work, the RAC approach still typically depends on the existence of a suitable test suite.

Another application of JML specifications is to serve as a basis for automated static code analysis. The idea in this case is to analyze the Java source code, looking for conditions that might cause the JML specification to be violated, without actually executing the code. One such analysis tool is ESC/Java2 [4,6,7]. When this tool is applied to the class of Figure 8, the output includes the warning shown in Figure 9.

```
simpleList.SimpleList: getFirst() ...
------------------------------------------------------
...\src\simpleList\SimpleList.java:62: Warning:
Postcondition possibly not established (Post)
        }
        ^
Associated declaration is
"...\src\simpleList\SimpleList.java", line 46, col 6:
          @  ensures \result != null;
             ^
Suggestion [62,1]: none
Execution trace information:
    Executed then branch in
"...\src\simpleList\SimpleList.java", line 56, col 23.
    Executed return in
"...\src\simpleList\SimpleList.java", line 61, col 2.
------------------------------------------------------
```

Figure 9. Excerpt from ECS/Java2 output for method of Figure 8

Although not obvious from the small code sections shown here, the problem the static checker has found is that, in the normal case, the **getFirst** method returns the first element of the **list** array, but the checker is unable to prove that this element is never null. To avoid the warning and to strengthen the specification, one solution is to add an "invariant" predicate, one that is asserted to be true at all method calls and returns. The invariant predicate added to the example code, shown in Figure 10, successfully satisfies the concern expressed by the ESC/Java2 checker.

```
/*@
 @ invariant (\forall int i; 0 <= i && i < list.length;
 @                        list[i] != null);
 */
```

Figure 10. JML invariant predicated added to the class of Figure 8

This assertion introduces another specification requirement, against which all parts of the class code are verified.

Tool Support Issues

One major problem in supporting an undergraduate course in formal methods is access to tool updates and support. Many tools have originated in research groups that by their nature have research, rather than practical tool development, as a primary goal. Whatever the reasons, support problems related to both Z/EVES and JML have been an continuing problem.

The Z/EVES tool was originally developed by ORA in Canada, and support for academic use was available for some time. However, that organization's web site no longer exists and for a long time no successor support group was easily identifiable. The graphical user interface (GUI) of the most recent version of Z/EVES (2.3) requires old versions of Python and related support software, and source code has not been available. A few annoying defects remain, particularly in the GUI, though an emacs-based alternative interface avoids some of these issues. Recently, there is some good news, in that negotiations are underway to make Z/EVES available as open source, or to port a version that is not dependent on the current proof engine [9].

This situation is a little better for the JML tools, for which development has been ongoing. Commonly available versions of the JML compiler have limited ability to handle Java language versions newer than 1.4.2. Some support for Java 5.0 generics is available, and most tools will

execute on a Java 5.0 virtual machine. A stand-alone JML launcher provides a graphical user interface to the JML tools. Integration with the Eclipse development environment is incomplete and often unusable, especially for recent (e.g., 3.3) versions of Eclipse.

The ESC/Java2 static checker can execute on a version 5.0 JVM, but currently will not run in a Java 6.0 environment. The current version does not parse version 5.0/6.0 source, and thus does not support generics; it has limited support for version 5.0 bytecode.

While these tools can perhaps be effectively used by researchers in spite of their limitations, these issues raise barriers for undergraduate software engineering students. Recent and continuing progress in improving the available tools is encouraging, but focused work on providing more polished tools would be a benefit to software engineering educators.

Integration of Related Program Outcomes

Since most undergraduate students do not have enough experience with large-scale software failures to appreciate the potential value of better software specification and validation, the MSOE formal methods course has from the beginning required that each student write a research paper on an example of a software failure that caused significant harm (e.g., personal injury or financial loss), where it appears that inadequate specification or verification played a role in the failure.

In addition to raising student awareness of the risks related to software system failures, this research paper assignment has also been used as an opportunity for students to work within a typical journal format. Since they already have to learn LaTeX in order to work with the Z notation, they are able to employ a LaTeX style definition used by the IEEE Transactions on Software Engineering. The resulting student papers automatically have the appearance of publications in this journal, and many students seem to enjoy seeing their own work "in print."

While the critical thinking and causal analysis required for a paper of this type is a challenge for some students, many have commented positively on the experience. A majority report that this is the first time they appreciated the potential negative effects of poorly done software development.

Student Feedback

While there are variations from one student cohort to the next, there has been a relatively consistent pattern of both positive and negative assessments. The positive comments relate to realizing the potential power and applicability of formal methods, the need for better specification and verification of complex software systems, and a greater awareness of the harm that can be caused by poorly engineered software. The negative comments focus on frustration with unfamiliar and incomplete tools, a lack of conviction that formal methods can actually be applied to contemporary software development, and an impatience with learning knowledge and skills that they may not perceive as being in demand by potential employers.

Instructor Preparation

For a university like MSOE, which has a primary focus on undergraduate education and industry technology transfer and thus does not have a population of full-time graduate students to support research projects, it is difficult to acquire or develop faculty expertise in an area like formal methods. While much information and many other resources are available, there is still a significant gap between the research centers and the faculty members who are trying to make

formal methods a viable part of an undergraduate software engineering program that is for the most part intended to prepare practitioners rather than researchers. We have been fortunate to have colleagues with a strong desire to learn and teach new things and to expand their expertise across a number of curricular areas, including formal methods. Nevertheless, significant opportunities for increased collaboration remain, among undergraduate SE educators and with formal methods researchers and tool developers.

Conclusion

In spite of all the difficulties of incorporating the practical application of formal methods into an undergraduate software engineering curriculum, the authors believe that there is value in doing so. In the short term, study of formal methods provides students with a radically different view of software development than they have encountered in other classes. In the longer term, it seems that one prerequisite for the wider use of formal methods is a population of practicing software engineers who are comfortable with the concepts of formal specification and verification. As an illustration of this point, an exchange from a student-industry panel may be instructive. Industry representatives from the program's advisory committee were meeting with the software engineering student body when one of the students asked the panel whether the represented organizations were actively using formal methods for software development. One by one, the industry participants responded in the negative. Finally, the last representative turned the question around into a challenge, saying "No, but we are waiting for someone like you to join us and show us whey we should be using them."

At the same time, there is never enough room in a software engineering curriculum for all the topics the faculty judge to be important. For this reason, the place (or not) of formal methods in an undergraduate program will continue to be debated. Comments and suggestions from other members of the software engineering education community are solicited and welcome.

References

1. J. P. Bowen and M. G. Hinchey, "Ten commandments of formal methods ... ten years later", *IEEE Computer*, January 2006.
2. L. Burdy et al, "An overview of JML tools and applications", *Eighth International Workshop on Formal Methods for Industrial Critical Systems (FMICS '03)*, Elsevier, June 2003
3. E. Clarke and J. Wing, "Formal methods: state of the art and future directions", *ACM Computing Surveys*, December 1996.
4. D. Cok, "Specifying Java iterators with JML and Esc/Java2", *Proceedings of the 2006 Conference on Specification and Verification of Component-Based Systems (SAVCBS '06)*, ACM, November 2006.
5. A. Diller, *Z: An Introduction to Formal Methods* (2nd edition), John Wiley & Sons, 1994.
6. ESC/Java2 (Kind Software) web site, http://kind.ucd.ie/products/opensource/ESCJava2/.
7. C. Flanagan et al, "Extended static checking for Java", *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, June 2002.
8. Formal Methods Virtual Library, http://vl.fmnet.info/.
9. L. Freitas, personal communication, The University of York, January 2008, http://www-users.cs.york.ac.uk/~leo/.
10. J. Jacky, *The Way of Z*, Cambridge University Press, 1997.
11. Java Modeling Language (JML)web site, http://www.eecs.ucf.edu/~leavens/JML/index.shtml.
12. Joint Task Force on Computing Curricula, IEEE Computer Society and Association for Computing Machinery, *Computing Curriculum — Software Engineering*, July 2004; http://sites.computer.org/ccse/.
13. G.Leavens, A. Baker, and C. Ruby, "Preliminary design of JML: a behavioral interface specification language for Java", *ACM SIGSOFT Software Engineering Notes*, May 2006.

14. Y. Ledru, “Identifying pre-conditions with the Z/EVES theorem prover”, *Proceedings of the 13th IEEE International Conference on Automated Software Engineering (ASE'98)*, October 1998.
15. M. Saaltink. “The Z/EVES system”, *Proceedings of the 10th International Conference on the Z Formal Method (ZUM)*, volume 1212 of Lecture Notes in Computer Science, Springer-Verlag, April 1997.
16. S. Skevoulis and V. Makarov, “Integrating formal methods tools into undergraduate computer science curriculum”, *Proceedings of the 36th ASEE/IEEE Frontiers in Education Conference*, October 2006.
17. J. Spivey. *The Z notation - A Reference Manual* (Second Edition). Prentice Hall, 1992.
18. J. B. Wordsworth, *Software Development with Z*, Addison-Wesley, 1992.

Biographies

MARK J. SEBERN is a Professor in the Electrical Engineering and Computer Science Department at the Milwaukee School of Engineering (MSOE), and was the founding program director for MSOE's undergraduate software engineering program. He has served as an ABET program evaluator for software engineering, computer engineering, and computer science.

HENRY L. WELCH is a Professor of computer and software engineering in the Electrical Engineering and Computer Science Department at the Milwaukee School of Engineering. He teaches a broad range of course in both the computer and software engineering programs ranging from embedded systems to computer graphics, artificial intelligence, and formal methods.