Weaving Formal Methods into the Undergraduate Computer Science Curriculum (Extended Abstract)

Jeannette M. Wing[1]

Computer Science Department
Carnegie Mellon University
Pittsburgh, PA USA
wing@cs.cmu.edu
WWW home page: http://www.cs.cmu.edu/~wing/

Abstract. We can integrate formal methods into an existing undergraduate curriculum by focusing on teaching their common conceptual elements and by using state of the art formal methods tools. Common elements include state machines, invariants, abstraction mappings, composition, induction, specification, and verification. Tools include model checkers and specification checkers. By introducing and regularly revisiting the concepts throughout the entire curriculum and by using the tools for homework assignments and class projects, we may be able to attain the ideal goal of having computer scientists use formal methods without their even realizing it.

1 Philosophy

Rather than treat formal methods solely as a separate subject to study, we should weave their use into the existing infrastructure of an undergraduate computer science curriculum. In so doing, we would be teaching formal methods alongside other mathematical, scientific, and engineering methods already taught. Formal methods would simply be additional weapons in a computer scientist's arsenal of ways to think when attacking and solving problems.

My ideal is to get to the point where computer scientists use formal methods *without* even thinking about it. Just as we use simple mathematics in our daily life, computer scientists would use formal methods routinely.

By formal methods I mean the specification and verification of hardware and software systems. Some methods will be accessible to undergraduates—these are the ones I hope computer scientists will use without realizing it. Some methods are more advanced, requiring either more mathematical sophistication or domain knowledge—those can be taught in upper-level electives, in graduate courses, or through independent undergraduate research projects. In this paper I focus on the former.

Dating back to the Dijkstra-Gries predicate transformer approach of program development [Gr81], we already have a long history of inculcating undergraduates with notions of program specification and verification. While there are varying degrees of success in teaching programming using this approach, the method is not used by programmers in practice. Moreover, specifying and verifying small, simple programs does not address the problems of scale and complexity faced by software engineers in industry. What should we do differently or why should we be more optimistic now?

First, we should focus on teaching the common elements of all (or most) methods, rather than on the specific notation or stylistic requirements of the method itself. Students writing large programs are not easily going to be able to do a stepwise refinement of design to code following the Dijkstra-Gries approach, but they can certainly learn and apply the notions of program specification, loop invariants, and termination functions. Programming-in-the-small and programming-in-the-large are inherently creative problem solving activities. Thinking in terms of formal methods concepts, e.g., invariants, forces the designer to take a more abstract perspective of a system than that taken with an algorithmic or operational approach. This more abstract thinking invariably provides the designer with new insights and a deeper understanding of the system's desired behavior.

Second, tools are essential. Without sufficient tool support, a method will not scale to practice. Model checking [CGP99] is a successful formal method because it addresses scale in two ways: it is applicable to a narrow problem domain (control aspects of hardware and protocols) and we do not have to specify the whole system before we can do some interesting verification. Furthermore, without appropriate tool support, typical computer science students have no incentive to use them. While mathematics students may be happy to do pencil and paper proofs, computer science students grow up using compilers, interpreters, operating systems, databases, graphical user interfaces, editors, electronic mail systems, spreadsheets, document preparation packages, web browsers, search engines, and so on. Formal methods tools have to be packaged in a way to fit into the way computer scientists work on a daily basis.

What follows is first, a list of the common elements and tools which we can teach to undergraduates, and second, specific suggestions on where to teach them with respect to existing courses found in a typical undergraduate computer science curriculum.

2 What We Can Teach

2.1 Common Elements

Below is a list of the elements that transcend the specific syntax and semantics of most formal methods. A firm understanding of these concepts goes a long way.

- *State machines*. The notion of a *state machine* as a tuple of a set of states, a set of initial states, and a transition relation between states; variations include accepting states, nondeterministic transition relations, finite state machines, labelled states, and labelled transitions. The notion of a *state* as a

mapping from variables to values; enrichments include using typed variables and values, and modeling both the environment and store as needed for imperative programming languages. The notion of *executions* as sequences of interleaved states and transitions; various projections on executions, for example, on just states, transitions, objects, or processes (these projections are often termed *traces* or *histories*). The notion of *behavior* and *observable behavior* of a state machine as a set of executions (or traces or histories).
- *Invariants.* The notion of state invariants; variations include abstract and representation type invariants in the context of abstract data types. The notion of loop invariants for statement-level reasoning.
- *Abstraction mappings.* The notion of abstraction functions for reasoning about abstract data types. The more general notion of simulation relations, for example, in relating states (or transitions or executions) of one machine to those of another.
- *Composition.* The notion of composition as a way to build larger machines (systems). Basic functional composition as in sequential composition of statements and nested and recursive procedure calls. The use of interfaces to compose modules, as already manifest in programming languages like Java and ML. Process composition for concurrent and distributed systems. Problems due to interference when composing concurrent processes.
- *Induction.* The basic notions of mathematical and complete (strong) induction. Structural induction for reasoning abstract data types. Computational induction for reasoning more generally about state machines.
- *Specification.* The notion of writing a formal description of *what* the system is supposed to do, not *how*; i.e., the difference between a specification and code. The notion of a type as a "weak" (in terms of expressibility), but extremely powerful (in terms of practicality) specification. Going further, pre-conditions and post-conditions and other predicates (e.g., Larch's **modifies** clause [GH93]). Going even further, the use of rely and guarantee predicates for reasoning about concurrent programs.
- *Verification.* The notions of correctness and termination of a program, and more generally, notions of safety and liveness properties of concurrent and distributed systems. Proof techniques for showing a system satisfies its specification. Termination functions and well-founded orderings for proofs of termination.

There are clearly mathematical prerequisites or corequisites for understanding these concepts. They are (1) discrete mathematics, minimally, algebraic structures and their properties; and (2) mathematical logic, minimally, first-order predicate logic, and proof techniques.

2.2 Tools

There are two classes of tools that we can use at the undergraduate level: model checkers and specification checkers.

There is no excuse not to be using *model checkers* in our undergraduate courses today. With a verification tool, we can more easily teach that verification complements the testing and simulation activities of practicing hardware and software engineers. Model checkers verify temporal properties of finite state machines. They are fast, completely automatic, and relatively easy to learn. There are industrial-strength, commercial model checkers available on the market. If the trend of using them in the hardware industry continues, then it behooves us as educators to ensure that our students are well-versed in the state of the art verification technology.

Specification checkers are less common and are still making their transition from research environments to industry. One promising kind of specification checker is exemplified by LCLint [EGHT94] and ESC/Java [CSRC00], which both support incremental specifications: as we add more to a specification, the tool can check more of the code. LCLint does much of the traditional lint checks of C programs, including unused declarations, type inconsistencies, and use-before-defintion; additional source code annotations, in the form of pre-/post-conditions, modifies clauses, and representation invariants, enable more powerful checks such as determining violations of information hiding, memory management errors, and dangerous data sharing or unexpected aliasing. ESC/Java (Extended Static Checking [DLNS98] for Java) also relies on annotated source code and can catch many common programming mistakes such as array index bounds errors, null dereference errors, type-cast errors, and deadlocks and race conditions in multi-threaded programs. The checker uses an automatic theorem prover to reason about the semantics of conditional statements, loops, procedure and method calls, exceptions, and mutex locks.

As an aside, I leave for the more advanced student, the upper-level elective courses, and the undergraduate researcher two other classes of formal methods tools: design checkers such as Nitpick [JD96] and Alcoa [Ja00], which are still in the research incubator; and theorem provers, which still require sophisticated users. Design checkers have much promise in their use in upper-level software engineering courses, but need more time to mature. Theorem provers require more expertise than we can expect our students to acquire in one semester, all the while learning other course material.

3 Specific Undergraduate Courses

Introduction to Programming. Here we can teach the concepts of specification and verification but likely only informally and at a high-level. Still, acclimating students to the difference between a specification and code and to the idea of verification in addition to testing is a good first step. Students should get in the habit of writing informal specifications, loop invariants, and termination arguments in their comments.

Data Structures and Algorithms. This course lends itself naturally to introducing and exercising notions of abstraction, representation invariants, inductive proofs, and state machines.

Programming Principles. This is the traditional course that many schools use to teach the concepts of program specification and verification. It may make sense to revisit this course if some of the material is distributed across the others. At Carnegie Mellon we use this course to teach the functional programming language paradigm (we use ML) with a heavy emphasis on types (as weak specications), modules (interfaces versus implementation; composition and abstraction techniques), and the course mantra “code with proof in mind” (recursive programs lend themselves to inductive proofs).

Programming Languages. This course provides the opportunity to revisit more formally the concepts perhaps learned only informally during the students' first year. For example, we can give semantics for imperative and object-oriented programming languages in terms of state machines. We can use logic programming languages to illustrate advantages and disadvantages of using executable specifications, i.e., where specifications are code and vice versa.

Compilers. Translators and interpreters, by definition, provide rich examples of abstraction mappings (defining or simulating one machine in terms of another). Correctness preserving transformations require statements of invariants (formal or not) and soundness arguments (formal or not). Target machines (compiler back-ends) are just state machines. This course comes close to the ideal, where students are using some elements of formal methods without realizing it.

Software Engineering. Students can complement the use of informal CASE tools and semi-formal design methods such as UML with the use of formal ones, e.g., model checkers and specification checkers. Here would be the place to introduce design checkers such as Nitpick and Alcoa.

Computer Architecture. Students can use model checkers such as SMV to verify properties of simple circuits, simple processor designs, bus protocols, and cache coherence protocols.

Operating Systems. Students can use model checkers to check safety properties, e.g., freedom from deadlock, of various mutual exclusion algorithms (e.g., Peterson's tie-breaker algorithm or Lamport's bakery algorithm), and with various synchronization primitives (e.g., semaphores, mutex locks, condition variables).

Networking. Students can use model checkers to check properties of simple network protocols. (A Carnegie Mellon undergraduate did an honors thesis using Nitpick to discover a flaw in the Mobile IPv6 protocol [JNW00].)

Databases. We can use relational databases and other data models to discuss all flavors of invariants. Transactional systems require understanding executions, observable behavior, consistency (correctness) constraints, and interference due to concurrency.

User Interfaces. Modeling the user, environment, and system as a set of interacting concurrent processes can provide the foundation for usage scenarios. Using model checkers such as FDR makes sense here.

Undergraduate upper-level electives such as Artificial Intelligence and Graphics presumably offer other opportunities as well.

4 Future Work

All the real work is future work. The ideas sketched in this paper are just ideas of what might be possible. We are faced with working out the details. The biggest obstacle is getting "buy-in" from our colleagues: convincing co-instructors, curricula committees, and administrators that integrating formal methods unintrusively is a good thing to do.

Also, while philosophically in Section 1 we argued to emphasize concepts, not notation, concrete notation is the conveyor of abstract ideas. To effectively weave in the teaching of elemental concepts with existing courses means adapting notations and methods to the languages already in use. For example, using ESC/Java makes sense to use in a data structures and algorithms course taught in Java; but using Z tools for that same course may require too much additional overhead.

The nitty-gritty hard future work is in thinking of the examples to use in lectures, in designing appropriate homework and exam problems, and in making learning these concepts and tools enjoyable.

We do not have to do everything, and we do not have to do everything all at once. We can begin, for example, by discussing state machines in a programming languages course, and by introducing model checkers in a homework assignment or project of a computer architecture course. The main thing is to start doing something!

References

[CGP99] Clarke, E.M., O. Grumberg, and D.A. Peled: *Model Checking*, MIT Press, 1999.

[CSRC00] Compaq Systems Research Center, `http://www.research.compaq.com/SRC/esc/Esc.html`

[DLNS98] Detlefs, D., K. Rustan M. Leino, G. Nelson, and J.B. Saxe: Extended Static Checking, Compaq SRC Research Report 159, 1998.

[EGHT94] Evans, D., J. Guttag, J.J. Horning, and Y.M. Tan: LCLint: A Tool for Using Specifiations to Check Code, *SIGSOFT Symposium on the Foudations of Software Engineering*, December 1994.

[Gr81] Gries, D.: *The Science of Programming*, Springer-Verlag, 1981.

[GH93] Guttag, J.V. and J.J. Horning, editors: *Larch: Languages and Tools for Formal Specification*, Springer-Verlag, 1993.

[JD96] Jackson, D. and C. Damon: "Nitpick Reference Manual," Carnegie Mellon University Technical Report CMU-CS-96-109, Computer Science Department, Pittsburgh, PA, January 1996.

[Ja00] Jackson, D.: "Alloy: A Lightweight Object Modelling Notation," MIT Technical Report 797, February 2000.

[JNW00] Jackson, D., Y. Ng, and J.M. Wing: "A Nitpick Analysis of Mobile IPv6," to appear in *Formal Aspects of Computing*, accepted January 2000.