Basic Reasoning Concept Inventory

Joan Krone, Denison University

Just as the Physics community found it useful to identify basic concepts that students need in their curriculum, but often have their own intuitive, but faulty beliefs about, the Computer Science community and the Software Engineering community need to identify a set of basic reasoning principles for our students. It is likely that some students will have developed their own, often erroneous ideas about these concepts, but in many cases, these are principles that the students may not have thought about at all, yet are necessary for students to learn in order to specify and verify good software.

In phase I of our CCLI grant, we identified six basic reasoning principles that our computer science students must understand:

1. The use of Boolean logic to reason about programs is critical for establishing correctness.
2. In order to model software components, one needs familiarity with basic discrete math structures, such as sets, strings, integers and other number systems, relations, and functions.
3. Precise (mathematical) specifications for software components are critical in order to reason about the software and establish its correctness.
4. Modular reasoning must allow for individual components to be certified as correct without a need to re-verify when those components are placed in a larger program.
5. From mathematical specifications for a given component and the components it uses, it is possible to generate mathematical clauses (verification conditions or VC's) that are equivalent to the correctness of the implementation of the given component. Students can use proof techniques from Boolean logic to prove the VC's.

In the following paragraphs we identify particular skills and specific content students need in order to effectively put into practice the principles listed.

In order to use Boolean logic to reason about their programs, students need to be facile with Boolean notation, including standard symbols for "and," "or," "not," "implies," quantification, etc., and must be able to apply fundamental rules of logic, such as modus ponens and the law of the excluded middle. They need the ability to read and understand mathematical notation.

Mastering material in discrete mathematics is important, but not sufficient for specifying and reasoning about software. Students must understand the distinction between mathematical structures and their computer counterparts. For example, they must recognize that the integers provided by computer hardware are not conceptually the same as mathematical integers. We

have ways to describe computer integers (with their associated bounds) so students can clearly distinguish between that formal description and the typical description of mathematical integers found in discrete math textbooks. Of course, it is important for students to see how the two sets are related.

With regard to specifications, the most important lesson for students is that any given ADT (abstract data type) needs to be described mathematically for purposes of reasoning about it. The mathematical specification is a separate entity from any implementation of that abstraction. In fact, students learn that there may be multiple implementations for any given specification, each with its own performance characteristics. It is possible to show the relationship between a given implementation and its specification by an appropriate mathematical relation, which in many cases will be a function.

Each operation associated with the ADT must be precisely specified with appropriate assertions such as pre and post conditions. Reasoning about any procedure can be based on these assertions.

The specification serves as an external view so that a client reading it will know exactly what the component will do without needing to know any implementation details. An implementer will use the specifications as a guide for the internal view not seen by the client-user.

This principle of separation of abstraction from implementation promotes the construction of large programs from component parts. When the specification provides an abstraction for objects of a type, its implementation must include an internal contract: A suitable representation invariant and a correspondence between internal representation of the object and its external abstraction. Other internal implementation assertions include invariants for loops and termination progress metrics for loops and recursion. These internal implementation assertions are necessary and sufficient for reasoning about implementation correctness.

Once a component implementation has been proven correct with respect to its specification, any client can incorporate it into a larger component without any need to re-verify. This principle of modular reasoning is critical for large systems. The programmer of a larger component or system can choose the parts needed for that component just by reading the specifications for those parts. In fact, one can construct such a component even before any implementations of the smaller parts are ready, though, of course, those parts must be implemented in parallel or at some time in order for the entire software system to work.

Once a software component has been mathematically specified, verification conditions (mathematical clauses produced from the assertions in the software) can be generated (automatically with tools or manually). Students can use their skills in applying logic to process

those assertions and ultimately to see if it is possible to construct a proof of these assertions, thereby proving the correctness of the software. Here students discover that although testing a program may catch errors, testing cannot verify correctness. What we see here is the critical connection between the field of mathematics and that of computer science. The table summarizes what students need in order to apply these reasoning principles.

Computer Science Reasoning Concept Inventory

Major Reasoning Topic	Subtopic Summary	Concept Term Highlights
Boolean Logic	Standard logic symbols, standard proof techniques.	Connectives including implication; quantifiers; supposition-deduction.
Discrete Math Structures	Sets, strings, numbers, relations, and other mathematical theories as needed.	Example set notations: element, union and intersection, and subsets. string notations: concatenation and length
Precise Specifications	Mathematical descriptions of software. Interfaces for clients and implementers. Math models for data structures. Specifications of operations.	Mathematical modeling; constraints; specification parameter modes; pre and post conditions; notations for input and output values.
Modular Reasoning	Internal contracts and assertions in implementations. Construction of large programs from certified components. Understanding the role of specifications in reasoning. Each module needs to be proven correct only once.	Representation invariant; abstraction correspondence; loop invariants; progress metrics. Reasoning with multiple specifications; tabular reasoning; goal-directed reasoning.
Correctness Proofs	Construction of verification Conditions (VCs); VCs as mathematical assertions equivalent to program correctness. Application of proof techniques to VC's.	States and abstract values of objects; Assumptions; Obligations; VCs; application of proof techniques on VCs.