# Teaching and Assessment of Mathematical Reasoning Principles Using a Concept Inventory

Svetlana V. Drchova, Jason O. Hallstrom, Joseph E. Hollingsworth, Joan Krone, Rich Pak, and Murali Sitaraman

**Technical Report RSRG-13-06**
School of Computing
100 McAdams
Clemson University
Clemson, SC 29634-0974 USA

August 2013

Copyright © 2013 by the authors. All rights reserved.

# Teaching and Assessment of Mathematical Reasoning Principles Using a Concept Inventory

SVETLANA V. DRACHOVA, Limestone College
JASON O. HALLSTROM, Clemson University
JOSEPH E. HOLLINGSWORTH, Indiana University Southeast
JOAN KRONE, Denison University
RICH PAK, Clemson University
MURALI SITARAMAN, Clemson University

Undergraduate computer science students need to learn analytical reasoning skills to develop high quality software and to understand why the software they develop works as specified. This paper describes a systematic process for introducing reasoning skills into the curriculum and assessing how well the students have learned those skills. It presents a comprehensive Reasoning Concept Inventory (RCI) that captures the finer details of basic skills students need to reason about software correctness. Elements of the inventory can be taught at various levels of depth across the curriculum.

This paper illustrates the use of the inventory over several iterations of a software engineering course and a development foundations course. It summarizes how learning outcomes motivated by the inventory lead to questions that in turn form the basis for a careful analysis of student understanding as well as for fine tuning what is taught.

Categories and Subject Descriptors: **D.2.4 [Software Engineering]:** Software/Program Verification – programming by contract, correctness proofs, formal methods.

General Terms: Design, Reliability, Experimentation, Languages, Theory, Verification

Additional Key Words and Phrases: Evaluation, components, formal methods, learning outcomes, objects, specifications

## 1. INTRODUCTION

Software quality is of paramount importance because computing has become ubiquitous in our society. National infrastructures for energy, health, and transportation rely on the correctness of sophisticated software subsystems that control their operation and ensure safety. The resulting emphasis on modular, component-based development of high quality software demands a new set of skills that future software engineers must possess. To write fully specified, verifiable code, computing students—software engineers of the future—need the ability to reason mathematically about software components and their relationships, be able to model them via mathematical constructs, and understand and write formal specifications and assertions using the precise language of mathematical notation. This paper summarizes our efforts to make this happen with the aid of a Reasoning Concept Inventory (RCI) and assess the impact.

Section 2 of the paper discusses elements of the RCI. Section 3 explains how the inventory forms a suitable basis for developing learning outcomes across the curriculum. Section 4 details four fundamental RCI items covered in our courses. Section 5 presents results from our classroom experimentation and assessment, and explains specifically how learning outcome analysis and assessment, based on the RCI, can guide and improve teaching. Section 6 contains our conclusions.

## 2. A REASONING CONCEPT INVENTORY

### 2.1 Motivation and Related Work

While there are numerous attempts in the literature to inculcate new concepts in computer science, what distinguishes this effort in teaching mathematical reasoning is that we have explicitly identified a concept inventory of principles that need to be taught across the curriculum and based our assessment on learning outcomes derived from that inventory.

The specific purpose of this inventory is to identify the basic set of principles that are central to analytical reasoning about correctness of software and that must be taught in undergraduate computing education. This inventory, while not a test for evaluating if a student has an accurate working knowledge of the set of principles, forms a suitable basis for defining learning outcomes and specific test questions to evaluate the attainment of those outcomes [Drachova 2013]. In the process of learning these principles, students understand and appreciate intricate and important connections between mathematics and computer science. Integrating reasoning as a connecting thread among courses also helps develop a cohesive view of the discipline as students graduate from introductory programming to advanced software development.

The importance of teaching mathematical reasoning in undergraduate computing has had a number of pioneers [Bucci et al. 2001; Gries et al. 2001; Henderson 2003; Krone et al. 2012]; while analytical reasoning is not the focus, the use of specifications and testing to improve problem solving and software quality is the focus of other pioneering educators (e.g., [Edwards 2001; Kumar 2004]).

Concept inventories are not new to the STEM area. In 1992, Hestenes et al. noted that typical physics students were entering their classes with incorrect notions about fundamental physics concepts. Even when presented with material to correct misunderstood ideas, these students tended to revert to their original thinking in subsequent courses. To address this problem, educators developed an inventory of Newtonian physics concepts, known as the Force Concept Inventory (FCI), which they believed to be a necessary part of every physics student's education [Hestenes et al. 1992].

Following the publication of the FCI, a number of concept inventories have been developed and successfully employed in other STEM disciplines, including statistics [Stone et al. 2003], heat transfer [Jacobi et al. 2003], electromagnetism [Notaros 2002], and thermodynamics [Midkiff et al. 2001]. Noting that foundational to mathematical reasoning in computing is the ability to reason using formal logic, Herman et al. have used studies of common student misconceptions to characterize the problems in this area and understand why students develop these misconceptions. They have incorporated the results of these studies and others in the digital logic concept inventory [Herman et al. 2010].

The RCI differs from inventories in other disciplines in that whereas those inventories methodically follow and document what has already been widely accepted in their field of study, it presents a path for educators to set as a goal teaching high quality software development for computing education and achieve that goal through a sequence of micro-steps. Few CS instructors have had the luxury of time to consciously consider the core principles that students need to develop correct software, even if they consider the topic to be important. In this sense, the inventory is the first of its kind. Till now, there has been no accepted or proposed view on what needs to be taught to enable students to reason mathematically about software correctness and develop high quality software. The RCI is also different from the more ambitious effort such as the one in [Goldman et al. 2008], where the authors identified important and difficult concepts in CS using a Delphi process, because it is focused on reasoning. The RCI is simply a natural culmination of information from reasoning experts, both past and present. It has been refined through interactions among over 25 educators and researchers over multiple iterations [Drachova 2013]. While the inventory is motivated by past experiences, its goal is to shape the future of software engineering practices.

There is emerging anecdotal evidence from birds-of-a-feather sessions, and dedicated panels and workshops on mathematical reasoning at SIGCSE over the past several years that educators are starting to realize the central role of analytical reasoning skills in developing high quality software. The hope is that they will use the RCI to shape the future of software engineering practices to produce high quality software. For example, in 2012, the School of Computing faculty at Clemson unanimously agreed to replace the school-wide learning outcome for undergraduate computer science students from "Graduates will be able to apply design and development principles in the construction of software systems," to "Graduates will be able to apply design and development principles in the construction of defect-free software systems that function in accordance with specifications." This updated learning outcome reflects the urgency of producing

high quality software that works correctly. The same faculty had also approved unanimously two years earlier the inclusion of mathematical reasoning principles in two courses required for computer science majors, as a result of the positive gains seen from our assessments. The RCI is a natural starting point towards realizing these goals at Clemson and at other interested institutions.

### 2.2 The Inventory

The RCI is devised to answer this central question: What essential analytical reasoning principles and topics should CS students learn if an educational goal is to teach students to build high quality software? In answering this question, using prior efforts as our motivation, we have identified a set of five essential reasoning principles. We have introduced these principles across the curriculum using a collaborative learning approach aided by "hands-on" reasoning tools.

The RCI consists of five major reasoning areas, each of which is further divided into a hierarchy of subtopics. Only the top two levels are shown in Table I. The subtopics are further refined into concept term highlights (Level 3) and concept details (Level 4). (These details are presented in this paper as and when necessary.) The complete RCI is detailed in [Drachova 2013] and is also available at: http://www.cs.clemson.edu/group/resolve/teaching/inventory.html

Table I. Outline of the Reasoning Concept Inventory

| Reasoning Topic | Subtopic Summary |
|---|---|
| 1. Boolean Logic | 1.1. Motivation<br>1.2. Standard logic symbols<br>1.3. Standard terminology<br>1.4. Standard proof techniques<br>1.5. Methods of proving<br>1.6 Proof strategies<br>1.7 Rules of inference |
| 2. Discrete Math Structures | 2.1. Motivation<br>2.2. Sets<br>2.3. Strings<br>2.4. Numbers<br>2.5. Relations and functions<br>2.6. Graph theory<br>2.7. Permutations and combinations |
| 3. Precise Specifications | 3.1. Motivation<br>3.2. Specification structure<br>3.3. Abstraction<br>3.4. Specifications of operations |
| 4. Modular Reasoning | 4.1. Motivation<br>4.2. Design-by-Contract<br>4.3. Internal contracts and assertions |
| 5. Correctness Proofs | 5.1. Motivation<br>5.2. Construction of verification conditions (VCs)<br>5.3. Proof of VCs |

The first area is Boolean logic (RCI #1). Mastering skills in the area of Boolean Logic enables students to reason about program correctness. It includes understanding of standard logic symbols, proof techniques, and connectives, such as implication, quantifiers, and supposition-deduction.

Skills in the discrete math structures area (RCI #2) provide familiarity with basic structures and enable students to model various software components with mathematical sets, strings, functions, relations, number systems, mathematical theories, etc.

Mastering material in discrete mathematics is necessary, but not sufficient for specifying and reasoning about software. Students must also understand the distinction between mathematical structures and digital structures; e.g., the integers provided by computer hardware are not the same as mathematical integers. We have ways to describe computer integers (with their associated bounds), so students can clearly distinguish between the formal description and the typical description of mathematical integers found in discrete math textbooks. Beyond integers, students need to learn that typical computing objects can be described mathematically, so that it is possible

to reason formally about the operations that manipulate them. For example, students learn that finite sequential structures (e.g., stacks, queues, and lists) can be modeled by mathematical strings (with suitable constraints).

Precise specifications are useful for reasoning about components without knowledge of their internal implementations. This idea of modular reasoning (RCI #4), i.e., the ability to reason about a component in isolation, motivates students to understand internal and external contracts, representation invariants, abstraction correspondence, loop invariants, progress metrics, etc.

In reasoning about components formally, the connections between proofs (RCI #5) and software correctness become apparent. Students learn to construct and understand verification conditions (VCs), which, if proved, establish the correctness of software. They learn the assumptions and obligations for each VC and apply proof techniques to verify them.

These skills are of course related. With respect to prerequisites, we have attempted to organize the RCI in a linear fashion. For example, students need to understand Boolean logic (RCI #1) and discrete structures (RCI #2) to comprehend precise specifications (RCI #3). After having acquired those skills, students proceed to modular reasoning (RCI #4) and correctness proofs (RCI #5).

### 2.3 Relationship to IEEE/ACM Curriculum

The 2008 interim revision of the IEEE/ACM 2001 computing curriculum includes a body of knowledge unit entitled Software Engineering. In this unit, there are core and elective subunits that contain topics related to formal specification, pre- and post-conditions, unit and black-box testing, etc. We have used mathematical, model-based specification techniques as a guide to neatly tie many of these disparate topics together into a coherent package. For example, here is a subset of the specific topics listed in the 2008 curriculum under the Software Engineering (SE) knowledge unit that we are able to correlate with our RCI-based instruction for software design. The relationships to topics in other areas, such as Discrete Structures (DS) and Programming Languages (PL) are also detailed in [Drachova 2013].

- SE/SoftwareDesign (RCI #3 & #4)
  - The role and use of contracts
  - Component-level design
- SE/ComponentBasedComputing (RCI #3 & #4)
  - Components and interfaces
  - Interfaces as contracts
  - Component design and assembly
- SE/SoftwareVerificationValidation (RCI #3)
  - Testing fundamentals, including test case generation and black-box testing
  - Unit testing
- SE/FormalMethods (RCI #3, #4, & #5)
  - Formal specification languages
  - Pre- and post-conditions
  - Formal verification

In the emerging 2013 version of the guidelines, in addition to topics DS, PL, and SE areas, RCI principles also correspond to topics in Software Development Fundamentals (SDF).

## 3. LEARNING OUTCOMES

We have outlined learning outcomes for each principle in each of the five RCI areas in [Drachova 2013]. A learning outcome specifies a performance criterion that a student must meet at the end of instruction to demonstrate that learning has occurred as expected. Learning outcomes are used to monitor student progress, develop instructional materials (e.g., slides, in-class activities, tutorials [Drachova-Strang et al. 2011], tests, etc.) and also serve as a tool for curriculum alignment.

We begin by specifying the learning outcome for a specific RCI area as a general instructional objective. For example, "RCI #3 – Precise Specifications" can be stated as a general instructional

objective as: “Comprehends precise mathematical specifications for software components.” For that general instructional objective, we list specific learning outcomes that indicate the types of student performances we expect, thus giving clarification to what is meant by “comprehends”. To illustrate these specific learning outcomes, we show some of the sublevels of RCI #3 in Table II.

Table II. A partial expansion of RCI #3

| 3.4. Specification of operations |
|---|
| 3.4.1. … |
| 3.4.2. … |
| 3.4.3. Pre- and post-conditions |
| 3.4.3.1. Specification parameter modes |
| 3.4.3.2. Responsibility of the caller |
| 3.4.3.3. Responsibility of the implementer |
| 3.4.3.4. Equivalent specifications |
| 3.4.3.5. Redundant specifications |
| 3.4.3.6. Notation to distinguish an incoming value |

Based on RCI #3.4.3, we list sample learning outcomes (LO) for various types of performances expected from the students.

LO1. *State* the responsibility of the caller of an operation with respect to the precondition.

LO2. *Specify* the result of invoking an operation based on its pre- and post-conditions for particular inputs.

LO3. *Write* a black-box test case based on an operation's pre- and post-conditions.

An additional step in writing the learning outcomes includes capturing the level of difficulty of the expected student performance. This is done by carefully choosing the verb used in the LO. We use a variation of Bloom's taxonomy [Bloom et al. 1956] that normally consists of six levels of cognitive or intellectual outcomes: Knowledge, Comprehension, Application, Analysis, Synthesis, and Evaluation (listed from lower/easier level to higher/harder level). Our simplified version reduces the six levels to three levels by combining adjacent levels: KC (Knowledge-Comprehension), AA (Application-Analysis), and SE (Synthesis-Evaluation). In the list above, specific learning outcome LO1 is at the KC level, LO2 is at the AA level, and LO3 is at the SE level. The Bloom's taxonomy level to which an LO belongs depends on the verb used to describe the complexity of the performance. For example, writing usually involves work at the synthesis level, and is therefore usually a more difficult cognitive task than stating a definition.

As mentioned earlier, we can use these learning outcomes for curriculum alignment. In Table III, the top row lists various undergraduate courses in a CS department's curriculum, starting from 100-level courses on up to 400-level courses. The leftmost column lists the general learning objectives (e.g., RCI #3's "Comprehends precise mathematical specifications for software components.").

Table III. Learning outcomes and curriculum alignment

| | 1XX | 1XX | 2XX | … | 4XX |
|---|---|---|---|---|---|
| 1 | KC | | AA | | |
| 2 | | KC | AA | | SE |
| … | | | | | |
| N | KC | KC | AA | | SE |

The body of the matrix captures where in the curriculum an intended learning outcome is expected. It also indicates the expected difficulty, as we fill in the matrix using the taxonomy codes KC, AA, and SE. For example, in Table III, the row labeled general learning objective "2" indicates that we have at least three specific LOs, with difficulty KC, AA, and SE. The KC learning outcomes appear in a 100-level course, the AA in a 200-level course, and the SE in a 300-level or 400-level course. For the RCI, we expect participating CS courses (i.e., the top row in Table III) to include (among others): discrete structures (math), foundational programming courses (e.g., CS1, CS2, and advanced data structures), analysis of algorithms, software engineering, etc.

Using the RCI and LOs as a framework, we have developed instructional material and various assessments. We utilize the results of these assessments as feedback to make improvements to our materials and adjust how particular concepts are taught. We use these learning goals and outcomes

as a tool for curriculum alignment, i.e., making sure our undergraduate computer science courses collectively cover important reasoning-related topics.

### 4. OUTCOME-DRIVEN TEACHING

We have used collaborative classroom exercises and reasoning tools to teach the principles in a sophomore-level Software Development Foundations (CPSC 215) and junior-level Software Engineering course (CPSC 372); details may be found elsewhere [Leonard et al. 2009; Sitaraman et al. 2009]. While we have assessed finer details of RCI items with quizzes, etc. [Drachova 2013], we discuss and present results for only key principles RCI #3.4, RCI #4.2, RCI #5.2, and RCI #5.3.1 that were tested with final exam questions.

The technical approach used for teaching reasoning in this paper is based on RESOLVE, which is an integrated specification and programming language, devised especially to facilitate automated, mathematical reasoning of object-based programs [Edwards et al. 1994; Sitaraman et al. 2011]. However, the ideas have been adopted and taught in the context of popular programming languages. For example in CP SC 215 at Clemson, the principles are introduced in a Java context [Cook et al. 2012a; Leonard, et al. 2009].

#### 4.1 Teaching RCI #3.4

In Section 3, we expanded RCI #3.4 and provided some specific learning outcomes. Below we show a test question used to assess LO3 at the AA level in a sophomore-level course. In this specification, a Queue of Entry is conceptualized as a mathematical string of entries. In the `ensures` clause, #Q denotes an incoming value; string notation “o” denotes concatenation.

---

**Sample Code Question 1.**

---

Give two test points to show your understanding of the following specifications:

**Operation** Mystery_1(**updates** Q: P_Queue; **updates** E:Entry);
    **requires** |Q| /= 0;
    **ensures** Is_Permutation (#Q, Q o <E>);

**Operation** Mystery_2(**updates** P: P_Queue);
    **requires** |P| > 1;
    **ensures there exists** X, Y: Entry, **there exists** Rst: Str(Entry) **such that**
        #P = <X> o Rst o <Y> **and** P = <Y> o Rst o <X>;

---

Sample Code Question 1 illustrates how students are expected to produce two test points for each operation, consisting of input/output pairs, to show that they understand the pre- and post-conditions. A valid input is determined by the `requires` clause of an operation, and the expected output is determined by the `ensures` clause. We also use non-descriptive operation names (e.g., Mystery_1), so students do not guess what an operation does from its name, but instead examine its formal specifications. By answering this question correctly, students demonstrate they have met the specific learning outcome (LO3). Various questions on different assessment instruments (e.g., quizzes, tests) can be asked to assess learning outcomes such as LO1 and LO2.

To teach specifications, we have used the TCRA (“Test Case Reasoning Assistant”), a tool that takes a student through a series of test-case creation exercises with rapid feedback [Leonard, et al. 2009]. Achieving a sophisticated understanding of pre- and post-conditions directly supports the development of correct software and the understanding of why it is correct.

TCRA is based on the following three precepts: there must be a focus on rigorous mathematical reasoning; the tool must take advantage of the pervasive existence of networked desktop and laptop devices; and the tool must leverage the benefits of the peer instruction approach where traditional classroom learning is augmented through learning from one’s peers.

There are two parts to TCRA, a student application and an instructor application. The student application is designed to permit the student to work through multiple problems similar to those

shown in Sample Code Question 1. While using TCRA, the student is presented with subsequent fully specified *mystery* operations and asked to develop test points that satisfy the operation's specification. This is exactly the same as was discussed above where the student was asked by a test question to produce two test points for each operation, consisting of an input/output pairs, to show that they understand the operation's pre- and post-conditions. During test taking there is no immediate feedback given to the student, but TCRA provides immediate feedback by instantiating the specification with the student supplied test points. If both the `requires` and `ensures` assertions evaluate to `true` then the student is notified that a valid test case was submitted. If either of the instantiated assertions evaluate to `false`, then the student is prompted to correct the test case.

Let us look at an example using Mystery_2 from Sample Code Question 1 (above). Mystery_2 refers to a P_Queue, which in our catalog is a preemptable queue, meaning that it permits adding an item to the front of the queue as well as to the rear. For this example, we will assume we are working with a queue of integers. One possible valid test case for Mystery_2 would be #P = <33,55,77,99> and P = <99,55,77,33>. With this input TCRA will instantiate the Mystery_2 specification as follows:

- requires: |<33,55,77,99>| > 0, which evaluates to true because the length of string #P is four.
  Note the specification refers only to 'P', but since this is the precondition, 'P' and '#P' are the same.
- ensures: <33,55,77,99> = <33> o <55,77> o <99> and
  <99,55,77,33> = <99> o <55,77> o <33>, which also evaluates to true.
  Note: X = 33, Y = 99, and Rst = <55,77> in the instantiation.

Examples of failed test cases include:

- #P = <33>, causes the `requires` clause to evaluate to false.
- #P = <33,55> and P = <33,55>, causes the `ensures` clause to evaluate to false.
- Etc.

TCRA comes preloaded with a number of exercises making it immediately usable. But it is not limited to just these preloaded exercises because the instructor can supply as many additional exercises as desired. For each additional exercise that is supplied, a Java class for that exercise must be provided, where the Java class must conform to a TCRA provided Java interface class.

TCRA monitors student progress using a student selected unique identifier. With this identifier in place, anonymity is preserved, but longitudinal data collection and analysis are supported. The data logged includes: the time of the activity, which exercises were selected (the student has some choice), the test cases provided and the correctness results, and the student's unique identifier.

The TCRA instructor application creates graphs using the logged data and thus provides the analysis. The instructor is able to monitor progress during a closed lab situation where the students are asked to periodically submit their log files, or to assess student performance sometime later. The instructor can tailor the reports by selecting a specific individual or by aggregating the data. Also the instructor can apply filters that permit looking at specific parts of the logged data, e.g., at specific time windows, or specific operations. Armed with these analyses, the instructor can begin to close the instructional loop by modifying how instructional time is used in order to help improve student learning in those areas that seem to be causing the most problems.

To study the effectiveness of the TCRA for improving on learning outcome LO3 at the AA level – "Write a black-box test case based on an operation's pre- and post-conditions", we summarize results from a quiz that we administered consisting of seven operations similar to Mystery_1 and Mystery_2 (above). For each operation, the student was required to provide two valid test cases. The quiz was administered prior to introducing the students to TCRA. The class average on the seven-question quiz was a 75%, which we used as a baseline measurement of student understanding. TCRA was then introduced and a follow-up quiz was given; however, only half of the students were allowed to use the tool during the quiz, the other half was not. On the

follow-up quiz the half that used the tool achieved a 94% average, while the half that was not allowed to use the tool achieved a 78% average, with an overall class average of 84%. Finally, two class periods later, the students who were not allowed to use TCRA on the follow-up quiz were allowed to re-take the quiz using the tool, and their average increased to 94.3%.

### 4.2 Teaching RCI #4.2

Table IV reveals RCI #4.2.3 “Construction of new components using existing components”.

Table IV. Partial expansion of RCI #4.2

| 4.2. Design-by-Contract<br>    4.2.1. …<br>    4.2.2. …<br>    4.2.3. Construction of new components using existing components<br>        4.2.3.1. Implementation of a specification<br>        4.2.3.2. Implementation of enhancement specification |
|---|

The corresponding exam questions test students’ ability to use modular reasoning. The general instructional objective for RCI #4.2 is “Comprehends how components interact using the principle of Design-by-Contract”. To correctly answer questions about this instructional objective, students are expected to use valid inputs to trace example component code built using others, and to come up with an operation’s post-condition based on the trace outcome. Here not only do they have to understand the pre- and post-conditions of the called operations and be able to write/trace an implementation, but also be able to work backwards to come up with the specification based on their observed outcome. A specific learning outcome that we are testing is “Write the operation’s post-condition based on the result of code execution”. This assesses at a higher level (i.e., the SE level) of Bloom’s taxonomy.

For in-depth understanding of modular reasoning principles, students are given projects to implement components according to the formal contracts (provided by an instructor) using other components. If the contract specifications are not followed, it is easy to pinpoint where the errors are. To support contract-based development and reasoning, we have developed and used a web-integrated reasoning environment [Cook et al. 2012b]. Details of project assignments based on the web IDE is the topic of [Cook et al. 2013].

### 4.3 Teaching RCI #5.2

Table V reveals RCI #5.2.2: “Connection between specifications and what is to be proved”.

Table V. Partial expansion of RCI #5.2

| 5.2. Construction of verification conditions (VCs)<br>    5.2.1. …<br>    5.2.2. Connection between specifications and what is<br>to be proved<br>        5.2.2.1. Assumptions<br>        5.2.2.2. Obligations |
|---|

Shown next is an instructional activity where students develop a reasoning table for a given Integer operation called Mystery. In the reasoning table, students write assumptions and obligations for each state of the operation (there are four states in Figure 1). Technical details of developing a reasoning table may be found elsewhere [Sitaraman, et al. 2011]. In state 0, Mystery’s pre-condition may be assumed; in state 3, Mystery’s post-condition must be proven (confirmed). If the code entails calling another operation, the pre-condition for the other operation is entered in the “Confirm” column in the state just prior to the call. Then, in the next state (upon termination), its post–condition may be assumed, i.e., it is entered in the “Assume” column. Students can collaboratively complete this table on the classroom blackboard, either sequentially or in parallel. Each Confirm assertion together with the assumptions (givens) that precede it and that can be used in its proof forms a verification condition (VC). The code is correct only if all VCs can be proved. The course generalizes this reasoning to teach how to write assumptions, obligations, and proofs for code involving objects, as illustrated in the example given next. Answering the question requires knowledge of proofs (RCI #5), Boolean logic, discrete math, specifications, and modular reasoning. The example illustrates that some questions necessarily assess more than one learning outcome, though the emphasis may be on the most advanced topic.

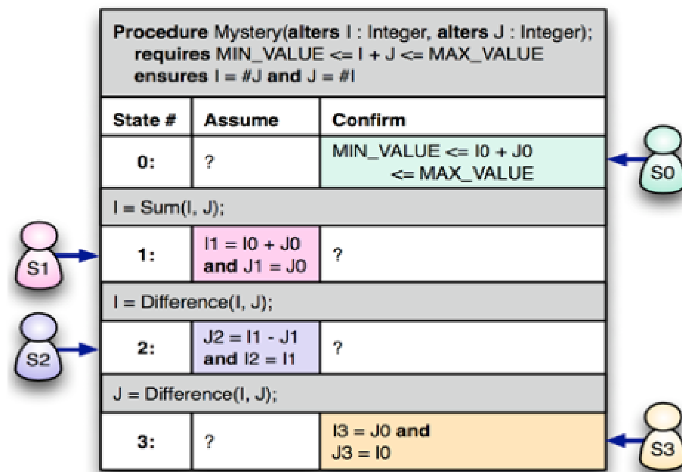


Fig. 1. Reasoning table activity.

Specific learning outcomes related to RCI #5.2 include the following, all in the SE or AA levels of Bloom's taxonomy:

- Analyze code for correctness – see Sample Code Question 1 (below)
- Write assumptions based on supplied code
- Write obligations based on supplied code

**Sample Code Question 2.**

Analyze the following recursive procedure for correctness with respect to the given specification. If incorrect, give one input for Queue Q where the code fails. Show the assumptions, if any, in states 4 and 6. Show the obligations, if any, in states 5 and 7.

```
Operation Remove_Last (updates Q: P_Queue; replaces E: Entry);
    requires |Q| > 0;
    ensures #Q = Q o <E>;

    Procedure
        decreasing |Q|;
        Var Next_Entry: Entry;                  (1)
        Dequeue(E, Q);                          (2)
        If (Length(Q) > 0) then                 (3)
            Remove_Last(Q, Next_Entry);         (4)
            E :=: Next_Entry;                   (5)
            Enqueue(Next_Entry, Q);             (6)
        end;                                    (7)
end Remove_Last;
```

### 4.4 Teaching RCI #5.3.1

Table VI below contains an expansion of RCI 5.3.

Table VI. Partial expansion of RCI #5.3

| 5.3. Proofs of Verification Conditions (VCs) |
|---|
| 5.3.1. VCs as mathematical implications |
| 5.3.1.1. Givens |
| 5.3.1.2. Goals |
| 5.3.2.. Application of proof techniques on VCs |

```
…
```

A simple learning outcome we tested was to ask students to "Identify provable goals when provided with mathematical implications (VCs) by checking relevant givens." Learning outcomes corresponding to RCI #5.3.2 are more challenging. Students were given VCs generated by the reasoning environment [Cook, et al. 2012b] for code examples, such as Remove_Last.

#### 4.5 Putting It Together

To experiment with and reinforce the reasoning ideas, a web-integrated development environment (Web IDE) has been developed at Clemson [Cook 2011] and is available freely at the RESOLVE project website via a browser at http://resolve.cs.clemson.edu/interface. A number of IDE's useful features make it a very useful tool for teaching a variety of reasoning principles at different difficulty levels. For details of actual project assignments used at Clemson and Alabama, please see [Cook, et al. 2013].

Novice students can use the interface to learn specification structure (RCI#3.2), abstraction (RCI#3.3), and operation specifications (RCI#3.4). Junior-level computer science students can experiment with modular thinking, and specifically design by contract (RCI#4.2), internal contracts and assertions (RCI#4.3), construction of verification conditions (RCI#5.2), and their proofs (RCI#5.3).

The IDE features a Component Finder to allow selection from a large number of component specifications, ranging from simple ones, such as integers and stacks, to more complex ones, such as maps and prioritizers. It also offers a variety of realizations (implementations), enhancements (concept extensions), enhancement realizations, and facilities ("main" modules). As mentioned earlier, not only can students use the existing components "as is" by accessing the built-in components, but they can modify them, save them, and create their own modified versions of these modules.

Components open in their own individual tabbed windows that allow the user easy access by switching between open tabs. By clicking a button on the IDE, a user can generate all the VCs (verification conditions that must be proven in order to show correctness) for the module in the current open tab. These VCs appear in a window adjacent to the code, which allows students as well as programmers to see what verification conditions must be proven. The IDE also supports automated proving of VCs, however, since theorem proving is still an open research question, many but not all generated VCs are currently provable.

Because "behind the scenes" RESOLVE is translated into Java code and interpreted by a regular Java interpreter, there is also an option to generate and display Java code, as well as to download an executable Java version of the program.

Screen shots in Figure 2 and Figure 3 illustrate a simple use of the web IDE. The open window in Figure 2 contains an "erroneous" iterative version of the Remove_Last operation, discussed in the last subsection. Iterative realizations, such as this one, are annotated with loop invariants through a *maintaining* clause and a termination progress metric, through a *decreasing* clause. The only defect in this code is the decreasing annotation (which must be |Q|, not |T|). It is erroneous because it indicates that the length of Queue T is decreasing each pass through the loop. The corresponding VC, VC_06, |T' o <E''>| < |T'| is not provable. VCs are generated and proving is attempted when a user clicks the Verify button. The first five VCs were successfully verified and these VCs correspond to establishing the invariant, the goal of Remove_Last operation, and the preconditions of called operations. Technical details of VC generation may be found elsewhere [Sitaraman et al. 2011].

In the figure, the blue oval icons on the left-hand side of the code window indicate line numbers for which VCs are generated. This helps highlight the critical lines of code that need to be verified. If a user hovers his mouse over a blue oval containing the letters "VC", a floating box displays the verification condition and its line number.

Figure 3 contains the results after the errors have been fixed and the Verify option has been selected. The IDE attempts to prove the generated VCs and displays the results on the right where

the check mark indicates that all six VCs were successfully verified. At this point the claim of having a fully functional verifier is not being made. Occasionally VC verification fails due to the fact that not all theorems/assertions have been implemented, and others are still being tested.

This interactive interface opens up many possibilities for teaching RCI principles because of the time it saves through automation of the VC generation and the proving of some of those VCs. For the instructor to generate and prove VCs by hand in the classroom often takes long time, even for simple examples. These features make it usable for teaching simple principles to novice students, as well as complex skills to the more experienced students and researchers.

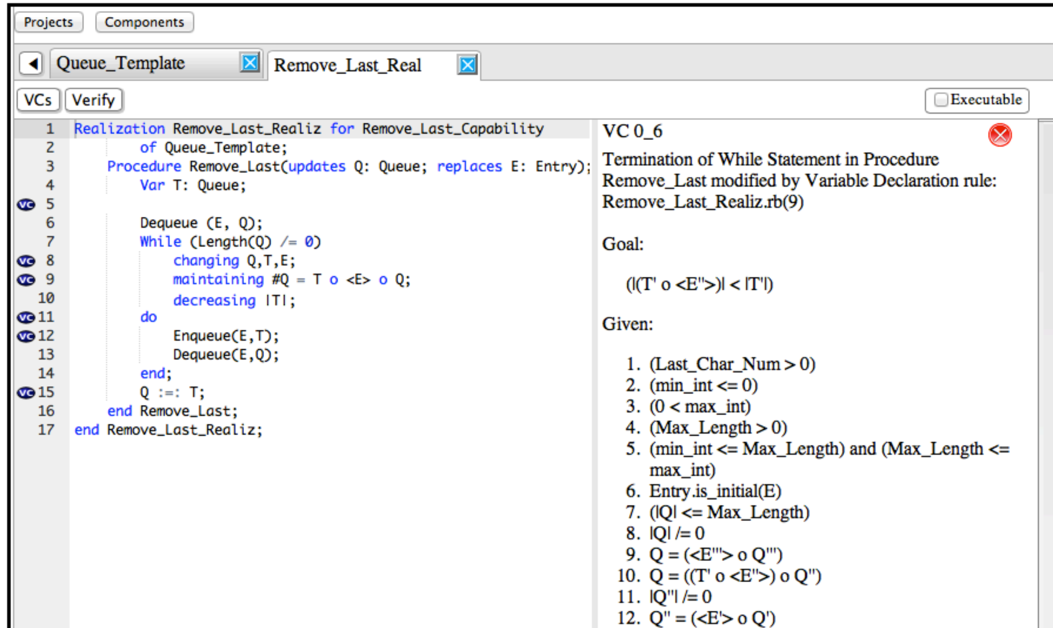


Fig. 2. An unprovable VC showing a bad loop progress metric for Remove_Last Code

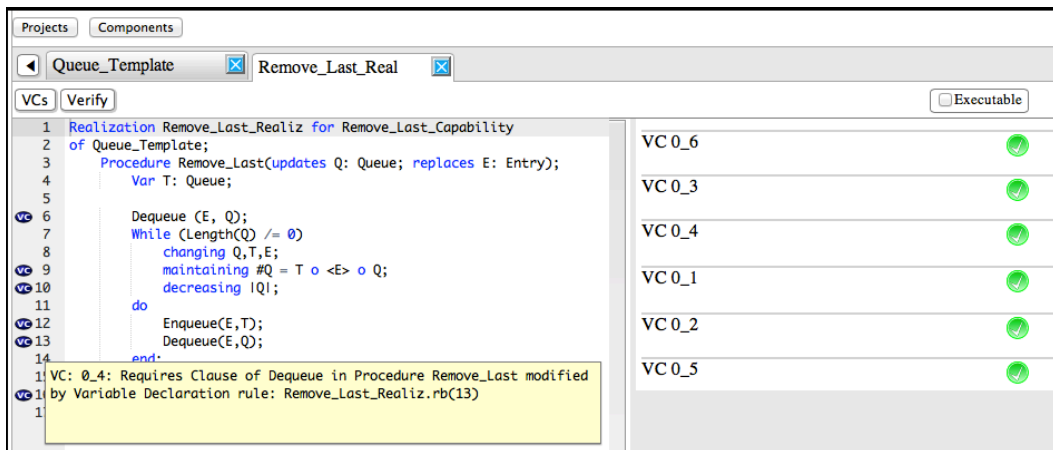


Fig. 3. Correct and proven code for Remove_Last operation

## 5. HOW THE RCI CAN GUIDE AND INFLUENCE TEACHING

### 5.1 Learning Outcome Assessment with RCI

Several institutions have introduced reasoning principles presented in this paper at different levels of difficulty in one or more of their computer science courses: University of Alabama (Software Engineering), Cleveland State University (Software Engineering), Denison University (Software Engineering and Programming Languages), DePauw University (Formal Languages), Indiana University Southeast (Software Engineering), North Carolina State University (Data Structures), Ramapo College of New Jersey (Programming Languages), Southern Wesleyan University (Data Structures), University of San Francisco at Quito, Ecuador (Data Structures), and Western Carolina University (Software Engineering).

This section discusses the assessment data based on specific RCI learning outcomes for multiple offerings of the Development Foundations (CPSC 215) and Software Engineering course (CPSC 372) at Clemson. We focus the discussion on data collected at Clemson, because we have had several offerings and interventions to be able to make meaningful conclusions. A complete set of data, associated confounding factors, observations, interventions, learning outcomes, and specific questions may be found in [Drachova 2013]. We highlight some key data and interventions here to illustrate the benefits of the RCI.

Table VII. CPSC 215 class averages for non-exempt students

| Reasoning Topic | Difficulty Level | Spring 2009 | Spring 2011 | Fall 2012 |
|---|---|---|---|---|
| | | Class Avg | Class Avg | Class Avg |
| RCI#3.4.3(2, 3) | KC | 96% | 96% | 96% |
| RCI#3.4.3(2, 3) | AA | 58% | 60% | 51% |
| RCI#4.1.1.3 | KC | 75% | 80% | 73% |
| RCI#4.2.1.1 | KC | 83% | 80% | 92% |
| RCI#5.2.2.1 | KC | 92% | 88% | 89% |
| RCI#5.3.2 | AA | 10% | 34% | 55% |
| RCI#5.2.2(1, 2) | AA | 84% | 68% | 83% |

The Development Foundations course offers a typical introduction to object-oriented software development, using Java as the implementation language. Approximately two-thirds of the course is devoted to traditional topics, including Java language fundamentals, object-oriented programming principles, design patterns, and other topics. The remainder of the course is devoted to analytical reasoning principles, including an introduction to mathematical modeling, formal contract specifications, contract-based testing, and a brief introduction to software verification. Details of the abstractions discussed in that course may be found elsewhere [Cook, et al. 2012a].

Summative assessment instruments contain questions testing learning outcomes for specific items from each category of the RCI that is taught. Table VII presents the assessment data from three semesters at Clemson for the classes taught by the same instructor: Spring 2009, Spring 2011, and Fall 2012. (The missing semesters in the table, Fall 2011 and Spring 2012, were taught by graduate teaching assistants, and aspects of that data are discussed in the context of an intervention in the next subsection.) These RCI reasoning principles have learning outcomes at the KC and AA levels of difficulty and were assessed using the same set of questions. About one third of the students were usually exempt from the final examination because they already had an ‘A’ going into the final. This data reflects the performance of the non-exempt students.

The percentages in the table indicate the average scores of students on questions pertaining to that RCI item. Overall, the data shows that students are capable of learning RCI reasoning principles in the undergraduate software development foundations course at varying levels of difficulty. Given that the data reflects only the performance of the non-exempt students, we conclude that student performance is acceptable on five out of the seven RCI reasoning principles: RCI# 3.4.3(2, 3) (pre- and post-conditions, level KC), RCI# 4.2.1.1 (specification as external

contracts, level KC), RCI#5.2.2(1, 2) (assumptions and obligations, level AA), RCI# 4.1.1.3 (formal verification, level KC), and RCI# 5.1.1.2 (assumptions, level KC). The data also shows that there is room for improvement for learning outcomes at the AA level for RCI# 3.4.3 and for RCI # 5.3.2 (proofs), though there is an improving trend for the latter.

With regard to the coverage of reasoning topics, the structure of the Software Engineering (CPSC 372) course is similar. The first two-thirds of the course covers traditional topics, with an emphasis on requirements analysis, software design, and project management. The remainder of the course expands upon the reasoning topics introduced in the preceding course, covering more advanced mathematical models, internal and external software contracts, loop invariants, and software verification (both manual and tool-assisted). Additional details may be found in [Sitaraman, et al. 2009]. Table VIII presents student assessment data from four semesters at Clemson: Fall 2010, Spring 2011, Fall 2011, and Spring 2012. Four RCI reasoning principles were assessed, on the AA and SE levels of difficulty. Student performance is acceptable on three out of the four RCI reasoning principles: RCI# 3.4.3 (pre- and post-conditions, level AA), RCI# 4.1.1.2, #3.4.3 (code tracing/inspection, pre- and post-conditions, level SE), and RCI#5.2.2(1, 2) (assumptions and obligation, level SE).

Table VIII. CPSC 372 class averages for all students

| Reasoning Topic | Difficulty Level | Fall 2010 Class Avg | Spring 2011 Class Avg | Fall 2011 Class Avg | Spring 2012 Class Avg |
|---|---|---|---|---|---|
| RCI#5.2.2 (1, 2) | SE | 61% | 73% | 76% | 71% |
| RCI#3.4.3 | AA | 94% | 78% | 89% | 84% |
| RCI#4.1.1.2 | SE | 93% | 79% | 86% | 84% |
| RCI#5.3 (1, 2) | SE | 88% | 88% | 46% | 86% |

### 5.2 RCI and Interventions to Improve Learning

This type of assessment data, collected over sufficiently many terms, serves another key purpose. It provides exactly the type of feedback needed to identify weaknesses in instructional materials, and more broadly, indicates where there is poor curriculum alignment. We consider two examples in this section.

The first example of an intervention corresponds to RCI #5.2.2 when it is introduced in the software development foundations course (CPSC 215). Specifically, we consider a Fall 2011 offering when a graduate teaching assistant (GTA) taught the course for the first time. To assess the learning of RCI #5.2.2 at the AA level, we asked students to write, as part of their final exam, the proof assumptions and obligations (in a reasoning table) for a simple piece of code. For that GTA, for non-exempt students, the average for this question was 64%, with 7 out of 14 students (50%) scoring a 70% or above. Since RCI #5.2.2 is central to understanding the correctness of an operation, we developed additional instructional materials.

We developed three five-minute podcast-type videos that explain in detail how a reasoning table is constructed, and how to write the assumptions and obligations. Each of these videos took approximately two hours for each minute of video to develop. They can be viewed at [Hollingsworth 2012]. These videos were provided to the students during the spring 2012 offering of CPSC 215, taught by the same GTA. The results summarized in Table IX show that in spring 2012, after the introduction of the videos, the results were better for RCI #5.2.2. The non-exempt student average was 79% (up from 64%), and 15 out of 21 (or 71%) scored a 70% or above (up from 50%).

Table IX. Improvements from RCI-based intervention and analysis

| RCI# and Course | Class Average | Percent of Students with a Score ≥ 70% |
|---|---|---|
| RCI #5.2.2, Fall 2011, | 64% | 50% |

| CPSC 215 | | |
|---|---|---|
| **RCI #5.2.2, Spr 2012, CPSC 215** | **79%** | **71%** |
| RCI #5.3.1, Fall 2011, CPSC 372 | 46% | 28% |
| **RCI #5.3.1, Spr 2011, CPSC 372** | **86%** | **90%** |

For another example, consider the performance results for RCI #5.3.1 in Table VIII. In the Fall 2010 and Spring 2011 offerings of CPSC 372, we noticed that the averages, as well as the percentage of students receiving 70% or higher for the learning outcome on proving VCs (RCI #5.3.1) were both high, at about 88%. Yet in the Fall 2011 offering, we noticed that the average had dipped to 46%, and those receiving 70% had dipped to 28%. We were able to relate this drop to the introduction of VCs and proofs that involved the notion of "part of the substring between indices." Subsequently, this led to alternative explanations of the idea and development of a new online tutorial based on the alternative explanation. In Spring 2012, the average for the outcome RCI #5.3.1 climbed back to 86%, and the number of students receiving 70% or higher went up to 90%. This is an example where a simple intervention was sufficient.

### 5.3 Role of the RCI in Instructor Feedback

A key benefit of the RCI is to ensure course alignment with objectives, especially when a variety of instructors teach a course. The sophomore-level software development foundations course was taught by three different instructors in the last academic year. A focus group meeting with three instructors (graduate teaching assistants in the last stages of their doctoral program) led to the finding that a surprisingly large number of important RCI topics had been covered, though the materials were taught over only a three to four-week period. The instructors shared their effective (and ineffective) teaching methods for various RCI topics. In the discussion, it also became clear that (due to inexperience) some topics had been tested inadvertently with questions at a higher-level of Bloom's taxonomy than the course objectives; this in turn helped to improve testing in the course.

The observations made at the focus group meeting regarding the challenges and successes of teaching reasoning topics with the RCI confirm our findings and conclusions made after inspecting the experimental data. A transcribed version of the focus group meeting is available in [Drachova 2013].

## 6. STUDENT ATTITUDINAL ASSESSMENT

Attitude measurement is important because it is well known in social psychology that attitudes not only affect behavior (i.e., they are predictive of future behavior), but that behavior can affect attitudes [Glasman and Albarracín 2006]. Thus, the educational interventions carried out in this research context can be said to have affected attitudes about software quality that will in turn have long-lasting effects on future behavior.

Attitudinal surveys conducted in both CPSC 215 and CPSC 372 indicate that students have positive attitudes about learning the RCI reasoning principles. A questionnaire was administered to students in each section at the beginning and end of the semesters for both courses. The survey and detailed data may be found in [Drachova 2013].

The questionnaire assesses student attitudes on the software engineering topics. The following scale was used in both courses: 1 (strongly disagree), 2 (moderately disagree), 3 (disagree), 4 (agree), 5 (moderately agree), and 6 (strongly agree). T-tests were used to compare students' attitudes before and after taking the class in which the tool was used. Because of multiple comparisons, the alpha level for significance testing was increased to .0002, given that all comparisons are statistically significant at $p < .002$. The rest of this section summarizes key statistically significant attitude shifts.

The analysis shows that at the end of the sophomore-level software development foundations course (CPSC 215), students had developed positive attitudes towards building high quality software. The response to the question, “My conception of how to build high quality software has changed significantly over time,” was significantly more positive after taking the class (pre: M=5.23, SD=0.95, post: M=4.71, SD=1.02, where M is the median value, and SD is the standard deviation). Since high quality software development is the ultimate goal, this shift is important. Another significant change was observed in the question “My conception of the difficulty associated with developing high quality software has changed significantly over time.” It was more positive at the end of the semester (pre: M=4.98, SD=1.17, post: M=4.44, SD=1.19). This is an important finding because it shows that students have learned what is involved in the software development process, exactly what we intended to teach them, and have more positive attitudes toward it. A desirable shift was also found in the question “The difficulty in understanding and modifying a 10,000 line software system has more to do with the style in which the software is written, and less to do with how smart I am,” with the attitude more positive at the end of the semester as well (pre: M=4.78, SD=1.05, post: M=4.30, SD=1.08).

At the end of the junior-level software engineering course (CPSC 372), students had developed positive attitudes towards using mathematical reasoning principles for high quality software. A significant shift occurred for the question “When working in teams, natural language (e.g., English) descriptions of the different components are sufficient for communication among team members” (pre: M=3.48, SD=1.4, post: M=4.57, SD=0.79). This result is noteworthy because it shows that students mastered precise specifications (RCI#3) and the importance of formal contracts. Related significant shifts in the anticipated direction occurred also for the questions “Having precise mathematical descriptions for each software component improves the likelihood that my software will be correct” (pre: M=5.04, SD=0.77, post: M=4.39, SD=0.94 and “I believe that there is a strong correlation between a person’s mathematical background and their ability to design and implement large systems correctly” (pre: M=4.78, SD=0.95, post: M=4.00, SD=1.28). Detailed tables with the survey results for CPSC 215 and CPSC 372 may be found in [Drachova 2013].

The results of the attitudinal surveys indicate that the attitudinal changes occurred exactly in the areas emphasized in each course. CPSC 215 taught basic concepts of software design, and CPSC 372 taught more advanced software engineering skills, including specifications, contracts, etc. In both courses, students’ attitudes have changed in a positive direction as the result of the instruction, which is the desired result.

## 7. CONCLUSIONS

The work presented in this paper builds on a decade of work that has confirmed the need for mathematics in the CS curriculum. The introduction of mathematical reasoning in CS courses has been identified to be important in developing problem solving skills in liberal arts institutions [Baldwin et al. 2010a; Remshagen 2010]. The discussion about teaching reasoning has continued at ACM SIGCSE through “Math Thinking” BoF sessions and panel discussions for the past several years (e.g., [Baldwin et al. 2010b; Gries, et al. 2001; Krone, et al. 2012]).

The work described in this paper has not only defined a concept inventory for mathematical reasoning principles, but also employed it in assessing how well students learn the concepts. We have systematically developed learning outcomes to match the principles in the inventory and devised teaching materials and exam questions to achieve and improve those outcomes. The results of a study over multiple offerings in two courses show the usefulness of this approach for educators in understanding where they are succeeding and where improvements should be made.

**ACKNOWLEDGMENTS**

We thank members of our research groups for their contributions to the contents of this paper. This research is funded in part by the NSF grants CCF-0811748, CCF-1161916, CNS-0745846, DUE-1022191, and DUE-1022941.

**REFERENCES**

BALDWIN, D., BRADY, A., DANYLUK, A., ADAMS, J., and LAWRENCE, A., 2010a. Case Studies of Liberal Arts Computer Science Programs. *Trans. Comput. Educ. 10*, 1, 1-30. DOI= http://dx.doi.org/10.1145/1731041.1731045.

BALDWIN, D., MARION, B., SITARAMAN, M., and HEEREN, C., 2010b. Some developments in mathematical thinking for computer science education since computing curricula 2001. In *Proceedings of the 41st ACM technical symposium on Computer science education* ACM, Milwaukee, Wisconsin, USA, 392-393. DOI= http://dx.doi.org/10.1145/1734263.1734397.

BLOOM, B.S., ENGELHART, M., FURST, E.J., HILL, W.H., and KRATHWOHL, D.R., 1956. Taxonomy of educational objectives: Handbook I: Cognitive domain. *New York: David McKay 19*, 56.

BUCCI, P., LONG, T.J., and WEIDE, B.W., 2001. Do we really teach abstraction? In *Proceedings of the Thirty-second SIGCSE technical symposium on Computer Science Education*, Charlotte, North Carolina, United States 2001, ACM, 364531, 26-30. DOI= http://dx.doi.org/10.1145/364447.364531.

COOK, C.T., 2011. A Web-Integrated Environment for Component-Based Reasoning, M.S. Thesis, Clemson University.

COOK, C.T., DRACHOVA, S., HALLSTROM, J.O., HOLLINGSWORTH, J.E., JACOBS, D.P., KRONE, J., and SITARAMAN, M., 2012a. A systematic approach to teaching abstraction and mathematical modeling. In *Proceedings of the Seventeenth ACM annual conference on Innovation and technology in computer science education*, Haifa, Israel 2012a, ACM, 2325378, 357-362. DOI= http://dx.doi.org/10.1145/2325296.2325378.

COOK, C.T., DRACHOVA-STRANG, S.V., SUN, Y.-S., SITARAMAN, M., CARVER, J.C., and HOLLINGSWORTH, J.E., 2013. Specification and Reasoning in SE Projects using a Web IDE. In *26th Conference on Software Engineering Education and Training (CSEE&T)* IEEE Computer Society, San Francisco, California, United States, 229 - 238.

COOK, C.T., HARTON, H., SMITH, H., and SITARAMAN, M., 2012b. Specification engineering and modular verification using a web-integrated verifying compiler. In *Proceedings of the 2012 International Conference on Software Engineering*, Zurich, Switzerland 2012b, IEEE Press, 2337423, 1379-1382.

DRACHOVA, S.V., 2013. Teaching and Assessment of Mathematical Principles for Software Correctness Using a Reasoning Concept Inventory, Ph.D. Dissertation, Clemson University.

DRACHOVA-STRANG, S., SITARAMAN, M., and HOLLINGSWORTH, J., 2011. Experimentation with Tutors for Teaching Mathematical Reasoning and Specification. In *Proceedings of the Conference on Frontiers in Education*, Las Vegas, Nevada.

EDWARDS, S.H., 2001. A Framework for Practical, Automated Black-Box Testing of Component-Based Software. *Software Testing, Verification and Reliability 11*, 2.

EDWARDS, S.H., HEYM, W.D., LONG, T.J., SITARAMAN, M., and WEIDE, B.W., 1994. Part II: specifying components in RESOLVE. *SIGSOFT Softw. Eng. Notes 19*, 4, 29-39. DOI= http://dx.doi.org/10.1145/190679.190682.

GLASMAN, L.R. and ALBARRACÍN, D., 2006. Forming attitudes that predict future behavior: A meta-analysis of the attitude-behavior relation. *Psychological Bulletin 132*, 5, 778 - 822. DOI= http://dx.doi.org/10.1037/0033-2909.132.5.778.

GOLDMAN, K., GROSS, P., HEEREN, C., HERMAN, G., KACZMARCZYK, L., LOUI, M.C., and ZILLES, C., 2008. Identifying important and difficult concepts in introductory computing courses using a delphi process. In *Proceedings of the 39th SIGCSE technical symposium on Computer science education*, Portland, OR, USA 2008, ACM, 1352226, 256-260. DOI= http://dx.doi.org/10.1145/1352135.1352226.

GRIES, D., MARION, B., HENDERSON, P., and SCHWARTZ, D., 2001. How mathematical thinking enchances computer science problem solving. In *Proceedings of the 32nd SIGCSE technical symposium on Computer Science Education*, Charlotte, North Carolina, United States 2001, ACM, 364754, 390-391. DOI= http://dx.doi.org/10.1145/364447.364754.

HENDERSON, P.B., 2003. Mathematical reasoning in software engineering education. *Commun. ACM 46*, 9, 45-50. DOI= http://dx.doi.org/10.1145/903893.903919.

HERMAN, G.L., LOUI, M.C., and ZILLES, C., 2010. Creating the digital logic concept inventory. In *Proceedings of the 41st ACM technical symposium on Computer science education*, Milwaukee, Wisconsin, USA 2010, ACM, 1734298, 102-106. DOI= http://dx.doi.org/10.1145/1734263.1734298.

HESTENES, D., WELLS, M., and SWACKHAMER, G., 1992. Force Concept Inventory. *The physics teacher 30*, 3, 141-158.

HOLLINGSWORTH, J., 2012. SIGCSE Workshop 2012 Instructional Video Series. http://www.cs.clemson.edu/group/resolve/teaching/ed_ws/sigcse2012/index.htmlw

JACOBI, A., MARTIN, J., MITCHELL, J., and NEWELL, T., 2003. A concept inventory for heat transfer. In *Frontiers in Education, 2003. FIE 2003 33rd Annual* IEEE, T3D-12-T13D-16 Vol. 11.

KRONE, J., BALDWIN, D., CARVER, J.C., HOLLINGSWORTH, J.E., KUMAR, A., and SITARAMAN, M., 2012. Teaching mathematical reasoning across the curriculum. In *Proceedings of the 43rd ACM technical symposium on Computer Science Education*, Raleigh, North Carolina, USA 2012, ACM, 2157208, 241-242. DOI= http://dx.doi.org/10.1145/2157136.2157208.

KUMAR, A., 2004. Web-based tutors for learning programming in C++/Java. *SIGCSE Bulletin 36*, 3, 266. DOI= http://dx.doi.org/10.1145/1026487.1008100.

LEONARD, D.P., HALLSTROM, J.O., and SITARAMAN, M., 2009. Injecting rapid feedback and collaborative reasoning in teaching specifications. In *Proceedings of the 40th ACM technical symposium on Computer science education*, Chattanooga, TN, USA 2009, ACM, 1509046, 524-528. DOI= http://dx.doi.org/10.1145/1508865.1509046.

MIDKIFF, K.C., LITZINGER, T.A., and EVANS, D., 2001. Development of engineering thermodynamics concept inventory instruments. In *Frontiers in Education Conference, 2001. 31st Annual* IEEE, F2A-F23 vol. 22.

NOTAROS, B.M., 2002. Concept inventory assessment instruments for electromagnetics education. In *Antennas and Propagation Society International Symposium, 2002. IEEE* IEEE, 684-687.

REMSHAGEN, A., 2010. Making discrete mathematics relevant. In *Proceedings of the 48th Annual Southeast Regional Conference*, Oxford, Mississippi 2010, ACM, 1900060, 1-6. DOI= http://dx.doi.org/10.1145/1900008.1900060.

SITARAMAN, M., ADCOCK, B., AVIGAD, J., BRONISH, D., BUCCI, P., FRAZIER, D., FRIEDMAN, H.M., HARTON, H., HEYM, W., KIRSCHENBAUM, J., KRONE, J., SMITH, H., and WEIDE, B.W., 2011. Building a push-button RESOLVE verifier: Progress and challenges. *Formal Aspects of Computing 23*, 5, 607-626.

SITARAMAN, M., HALLSTROM, J.O., WHITE, J., DRACHOVA-STRANG, S., HARTON, H.K., LEONARD, D., KRONE, J., and PAK, R., 2009. Engaging students in specification and reasoning: "hands-on" experimentation and evaluation. In *Proceedings of the 14th annual ACM SIGCSE conference on Innovation and technology in computer science education*, Paris, France 2009, ACM, 1562899, 50-54. DOI= http://dx.doi.org/10.1145/1562877.1562899.

STONE, A., ALLEN, K., RHOADS, T.R., MURPHY, T.J., SHEHAB, R.L., and SAHA, C., 2003. The statistics concept inventory: A pilot study. In *Proccedings of 33rd Annual Conference on Frontiers in Education,* IEEE.