Assessment of Learning Outcomes Using a Mathematical Reasoning Concept Inventory

Svetlana V. Drachova
Joseph Hollingsworth
Joan Krone
Jason O. Hallstrom
Murali Sitaraman
Indiana University Southeast
Clemson University
New Albany, IN
Clemson, SC 29634
812.941.2425
864.656.3444

jholly@ius.edu
sdrachosa,jasonoh,murali@clemson.edu

ABSTRACT
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 possess (as a minimum) 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.

Keywords
Evaluation, components, formal methods, learning outcomes, objects, specifications.

1. INTRODUCTION
Software applications are finding increased use in all areas of society; failures can cause loss of missions and lives. A recent shift towards 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, our students 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.

Paper Organization. Section 2 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 at Clemson 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 the 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. 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 [1, 2, 3, 4]. Concept inventories are also 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 [5].

Following the publication of the FCI, a number of concept inventories have been developed and successfully employed in other STEM disciplines, including statistics [6], heat transfer [7], digital logic [8], electromagnetism [9], and thermodynamics [10].

The inventory in this paper is focused on reasoning, unlike the more ambitious effort in [11], where the authors identified important and difficult concepts in CS using a Delphi process. While identifying the topics in a more general CS context benefits from such a process, the RCI is simply a natural culmination of information from reasoning experts, both past and present. The 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 possess (as a minimum) 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.

Keywords
Evaluation, components, formal methods, learning outcomes, objects, specifications.

1. INTRODUCTION
Software applications are finding increased use in all areas of society; failures can cause loss of missions and lives. A recent shift towards 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, our students 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.

Paper Organization. Section 2 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 at Clemson 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 the 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. 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 [1, 2, 3, 4]. Concept inventories are also 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 [5].

Following the publication of the FCI, a number of concept inventories have been developed and successfully employed in other STEM disciplines, including statistics [6], heat transfer [7], digital logic [8], electromagnetism [9], and thermodynamics [10].

The inventory in this paper is focused on reasoning, unlike the more ambitious effort in [11], where the authors identified important and difficult concepts in CS using a Delphi process. While identifying the topics in a more general CS context benefits from such a process, the RCI is simply a natural culmination of information from reasoning experts, both past and present. The 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 possess (as a minimum) 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.

Keywords
Evaluation, components, formal methods, learning outcomes, objects, specifications.

1. INTRODUCTION
Software applications are finding increased use in all areas of society; failures can cause loss of missions and lives. A recent shift towards 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, our students 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.

Paper Organization. Section 2 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 at Clemson 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 the 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. 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 [1, 2, 3, 4]. Concept inventories are also 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 [5].

Following the publication of the FCI, a number of concept inventories have been developed and successfully employed in other STEM disciplines, including statistics [6], heat transfer [7], digital logic [8], electromagnetism [9], and thermodynamics [10].

The inventory in this paper is focused on reasoning, unlike the more ambitious effort in [11], where the authors identified important and difficult concepts in CS using a Delphi process. While identifying the topics in a more general CS context benefits from such a process, the RCI is simply a natural culmination of information from reasoning experts, both past and present. The 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 possess (as a minimum) to reason about software correctness. Elements of the inventory can be taught at various levels of depth across the curriculum.
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 our 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 1. 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 available at the Reasoning Wiki site: http://www.cs.clemson.edu/group/resolve/teaching/inventory.html

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

Table 1. Outline of the Reasoning Concept Inventory

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 knowledge unit that we are able to correlate with our RCI-based instruction for software design.

- 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

3. LEARNING OUTCOMES

In addition to developing the RCI, we have outlined learning outcomes for each of the five RCI areas. Several of our outcomes are related to the outcomes in the corresponding topics of the 2008 IEEE/ACM Curriculum. 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 [14], 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 2.

Table 2. Partial expansion of RCI #3

<table>
<thead>
<tr>
<th>3.4. Specification of operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.4.1. …</td>
</tr>
<tr>
<td>3.4.2. …</td>
</tr>
<tr>
<td>3.4.3. Pre- and post-conditions</td>
</tr>
<tr>
<td>3.4.3.1. Specification parameter modes</td>
</tr>
<tr>
<td>3.4.3.2. Responsibility of the caller</td>
</tr>
<tr>
<td>3.4.3.3. Responsibility of the implementer</td>
</tr>
<tr>
<td>3.4.3.4. Equivalent specifications</td>
</tr>
<tr>
<td>3.4.3.5. Redundant specifications</td>
</tr>
<tr>
<td>3.4.3.6. Notation to distinguish an incoming value</td>
</tr>
</tbody>
</table>

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 L0s 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 [15] 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 3, 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 3. Learning outcomes and curriculum alignment

<table>
<thead>
<tr>
<th>1XX</th>
<th>1XX</th>
<th>2XX</th>
<th>…</th>
<th>4XX</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>KC</td>
<td>AA</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>KC</td>
<td>AA</td>
<td>SE</td>
<td></td>
</tr>
<tr>
<td>…</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>N</td>
<td>KC</td>
<td>AA</td>
<td>SE</td>
<td></td>
</tr>
</tbody>
</table>

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 3, the row labeled “Application, Analysis, Synthesis, and Evaluation” 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 3) 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 (CP SC 215) and junior-level Software Engineering course (CP SC 372); details may be found elsewhere [16, 17]. While we have assessed finer details of RCI items with quizzes, etc., due to space limitations, 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 CP SC 372 final exam questions.

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. In this specification, a Queue of Entry is conceptualized as a mathematical string of entries. In the ensures clause, #D denotes an incoming value; string notation “o” denotes concatenation.

Above we show a test question used to assess LO3. In this specification, a Queue of Entry is conceptualized as a mathematical string of entries. In the ensures clause, #D denotes an incoming value; string notation “o” denotes concatenation.

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 R: Str(Entry) such that  
#P=<X> o Rst o <Y> and P=<Y> o Rst o <X>;

Students are expected to produce two test points for each operation, consisting of an 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 [17].
4.2. Teaching RCI #4.2

Table 4 reveals RCI #4.2.3: “Construction of new components using existing components”.

<table>
<thead>
<tr>
<th>Table 4. Partial expansion of RCI #4.2</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.2. Design-by-Contract</td>
</tr>
<tr>
<td>4.2.1. ...</td>
</tr>
<tr>
<td>4.2.2. ...</td>
</tr>
<tr>
<td>4.2.3. Construction of new components</td>
</tr>
<tr>
<td>4.2.3.1. Implementation of a specification</td>
</tr>
<tr>
<td>4.2.3.2. Implementation of enhancement specification</td>
</tr>
</tbody>
</table>

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 the errors are. To support contract-based development and reasoning, we have developed and used a web-integrated reasoning environment [18].

4.3. Teaching RCI #5.2

Table 5 reveals RCI #5.2.2: “Connection between specifications and what is to be proved”.

<table>
<thead>
<tr>
<th>Table 5. Partial expansion of RCI #5.2</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.2. Construction of verification conditions (VCs)</td>
</tr>
<tr>
<td>5.2.1. ...</td>
</tr>
<tr>
<td>5.2.2. Connection between specifications and what is to be proved</td>
</tr>
<tr>
<td>5.2.2.1. Assumptions</td>
</tr>
<tr>
<td>5.2.2.2. Obligations</td>
</tr>
</tbody>
</table>

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). 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.

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|:

```plaintext
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)
EndQueue(Next_Entry, Q); (6)
end: (7)
end Remove_Last;
```

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
- Write assumptions based on supplied code
- Write obligations based on supplied code

4.4. Teaching RCI #5.3.1

The table below contains an expansion of RCI 5.3.

<table>
<thead>
<tr>
<th>Table 6. Partial expansion of RCI #5.3</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.3. Proofs of Verification Conditions (VCs)</td>
</tr>
<tr>
<td>5.3.1. VCAs as mathematical implications</td>
</tr>
<tr>
<td>5.3.1.1. Givens</td>
</tr>
<tr>
<td>5.3.1.2. Goals</td>
</tr>
<tr>
<td>5.3.2. Application of proof techniques on VCs</td>
</tr>
</tbody>
</table>

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 [18] for code examples, such as Remove_Last. We are not able to provide examples here due to space limitations.
5. HOW THE RCI CAN GUIDE AND INFLUENCE TEACHING

5.1. Learning Outcome Assessment with RCI

We have collected assessment data based on specific RCI learning outcomes for multiple offerings of the Development Foundations (CP SC 215) and Software Engineering course (CP SC 372) at Clemson. We focus our analysis on the outcomes at the end of the SE course. The course teaches typical topics, such as requirements analysis and design (for 2/3 of the course), in addition to reasoning topics (1/3 of the course). Summative assessment instruments contain questions testing learning outcomes for specific items from each category of the RCI that is taught. The end of course assessment instrument assesses RCI #3.4, RCI #4.2.3, RCI #5.2.2, and RCI #5.3.1 (discussed previously). The collected data is shown in Table 7.

Table 7 contains cumulative data from analysis of final exam questions of 4 recent offerings of the software engineering course. Here, the “Class Average” column indicates the class average for the particular RCI item tested, and the column labeled “Percent of Students with a Score ≥ 70%” indicates the percentage that earned at least 70% of the points for the RCI item tested.

<table>
<thead>
<tr>
<th>RCI#</th>
<th>Class Average</th>
<th>Percent of Students with a Score ≥ 70%</th>
</tr>
</thead>
<tbody>
<tr>
<td>RCI #3.4</td>
<td>87%</td>
<td>76%</td>
</tr>
<tr>
<td>RCI #4.2.3</td>
<td>86%</td>
<td>84%</td>
</tr>
<tr>
<td>RCI #5.2.2</td>
<td>72%</td>
<td>61%</td>
</tr>
<tr>
<td>RCI #5.3.1</td>
<td>97%</td>
<td>66%</td>
</tr>
</tbody>
</table>

The inventory-based learning outcome analysis above shows that most students understood concepts RCI #3.4 (specification of operations), RCI #4.2.3 (design-by-contract principles), RCI #5.2.2 (constructing verification conditions to determine code correctness), and RCI #5.3.1 (basics of proofs).

5.2. RCI and Feedback to Improve Teaching

Whereas the cumulative data above shows that the reasoning concepts can be effectively taught and assessed using the RCI-based learning outcomes, this type of 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 give two examples in this section.

First we note that the numbers for RCI item #5.2.2 are comparatively low, though the item is at the SE-level of Bloom’s taxonomy (the highest level of performance). The Fall 2010 average and percentage of students with 70% or higher turned out to be low, at 60% and 40%, respectively. Intervention over the next three semesters helped improve the numbers from 60% to 74%, and 40% to 66%.

Detailed analysis of this learning outcome indicated that we needed to improve the teaching of RCI #5.2.2 when it was introduced in the (pre-requisite) sophomore-level development foundations course (CP SC 215). (Abstractions discussed in that course may be found elsewhere [19].) To assess the learning of RCI #5.2.2 in this course, we asked students to write, as part of their final exam, the assumptions and obligations (in a reasoning table) for a simple piece of code. For one instructor, the average on this question was 64%, with 7 out of 14 (50%) scoring a 70% or above. We noted that for the fall class, only 50% of the students passed this part of the test with a 70% or better. 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 [20]. The topics in these videos were covered in class during Fall 2011, but like any lecture/discussion, the material is often covered only once, and when the students see it, they often say “it looked easy when the instructor did it.” These videos were provided to the students during the Spring 2012 offering of CP CS 215, taught by the same instructor. This time the results on the final were better for RCI #5.2.2, with an average of 79% (up from 64%), and 15 out of 21 (or 71%) scoring a 70% or above (up from 50%). It may seem that 71% is still too low, and of course we agree, however, you must recall that this level of performance is at the highest levels of Bloom’s taxonomy, and this is a 200-level course. More importantly, we expect the numbers to improve further at the end of the later software engineering course.

For another example, consider RCI #5.3.1. In Fall 2010 and Spring 2011 offerings of CP SC 372, we noticed that the averages, as well as the percentage of those receiving 70% or higher for the learning outcome on proving VCs (RCI #5.3.1) were both high, at about 90%. Yet in Fall 2011, 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. 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 simple intervention was sufficient.

These improvements in learning outcomes for RCI #5.2.2 for CP SC 215 and for RCI #5.3.1 for CP SC 372 are summarized below. Boldface rows indicate the improvements after intervention.

Table 8. Improvements from RCI feedback and analysis

<table>
<thead>
<tr>
<th>RCI# and Course</th>
<th>Class Average</th>
<th>Percent of Students with a Score ≥ 70%</th>
</tr>
</thead>
<tbody>
<tr>
<td>RCI #5.2.2, Fall 2011, CP SC 215</td>
<td>64%</td>
<td>50%</td>
</tr>
<tr>
<td>RCI #5.2.2, Spr 2012, CP SC 215</td>
<td>79%</td>
<td>71%</td>
</tr>
<tr>
<td>RCI #5.3.1, Fall 2011, CP SC 372</td>
<td>46%</td>
<td>28%</td>
</tr>
<tr>
<td>RCI #5.3.1, Spr 2011, CP SC 372</td>
<td>86%</td>
<td>90%</td>
</tr>
</tbody>
</table>

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 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 upon request.

6. 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 [21, 22]. This discussion has continued at SIGCSE through “Math Thinking” BoF sessions each of the past 3 years, and panel discussions about mathematical reasoning for CS students at SIGCSE 2010 and 2012. One of the issues that has repeatedly arisen following various mathematical reasoning discussions at SIGCSE in panel, paper, and BoF sessions is the need for a concept inventory.

The work described in this paper has not only defined an inventory, 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 four semesters in two courses show the usefulness of this approach for educators in understanding where they are succeeding and where improvements should be made. In the not so distant future, we hope to collect assessment data from multiple institutions and examine the inter-institutional trends.

6. ACKNOWLEDGMENTS

This research was funded in part by the NSF grants CCF-0811748, CNS-0745846, DUE-1022191, and DUE-1022941.

7. REFERENCES


