TEACHING AND ASSESSMENT OF MATHEMATICAL PRINCIPLES FOR SOFTWARE CORRECTNESS USING A REASONING CONCEPT INVENTORY

A Dissertation
Presented to
the Graduate School of
Clemson University

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy
Computer Science

---

by
Svetlana V. Drachova-Strang
May 2013

Accepted by:
Dr. Murali Sitaraman, Committee Co-Chair
Dr. Joseph E. Hollingsworth (Indiana University Southeast), Co-Chair
Dr. Harold Grossman
Dr. Jason O. Hallstrom
Dr. David P. Jacobs

ABSTRACT

As computing becomes ubiquitous, software correctness has a fundamental role in ensuring the safety and security of the systems we build. To design and develop software correctly according to their formal contracts, CS students, the future software practitioners, need to learn a critical set of skills that are necessary and sufficient for reasoning about software correctness.

This dissertation presents a systematic approach to both introducing these reasoning skills into the curriculum, and assessing how well the students have learned them. Specifically, it introduces a comprehensive Reasoning Concept Inventory (RCI) that captures the fine details of basic reasoning skills that are ideally learnt across the undergraduate curriculum to reason about software correctness, to develop high quality software, and to understand why software works as specified. The RCI forms the basis for developing learning outcomes that help educators to assess the adequacy of current techniques and pinpoint necessary improvements. This dissertation contains results from experimentation and assessment over the past few years in multiple CS courses. The results show that the finer principles of mathematical reasoning of software correctness can be taught effectively and continuously improved with the help of the RCI using suitable teaching practices, and supporting methods and tools.

DEDICATION

To my mother, father, grandmother Faya, and Vishy, who believed in me and supported me throughout this endeavor. *Semper Gratis*.

ACKNOWLEDGMENTS

I would like to thank members of the Clemson and Ohio State RESOLVE Software Research Group (RSRG) for their discussions on the topic. I thank the members of the dissertation committee for taking the time to review and comment on this manuscript.

Special gratitude goes to my professor and advisor Murali Sitaraman and professor Joseph Hollingsworth, for their patience, invaluable suggestions, timely criticisms, long intense discussions, multiple long-distance conferences, and numerous revisions of my dissertation. My thanks are also due to professors Joan Krone, Richard Pak, and Pradip Srimani for their comments on topics of this dissertation. Thanks also to all the educators who have assisted in the development of the RCI and learning outcomes and to the instructors and professors at Clemson and elsewhere for their help with experimentation and assessment.

I also want to expresses my gratitude to the National Science Foundation for their funding of this research project in part through grants CCF-0811748, DUE-1022941, and CCF-1161916.

TABLE OF CONTENTS

Page

Table of Contents (Continued)

Page

LIST OF TABLES

List of Tables (continued)

Table Page

List of Tables (continued)

Table Page

LIST OF FIGURES

Figure Page

List of Figures (continued)

Figure Page

List of Figures (continued)

Figure Page

CHAPTER ONE

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. Malfunction of mission-critical software can cause unacceptable and completely preventable loss of human life. Even defects in the software used in every-day activities can cause loss of client data, financial loss, or just inconvenience.

Developing verified software—software that behaves according to stated formal contracts—has great implications for all areas of software application. Specified and verified software is more reliable, experiences fewer failures, and costs less to maintain time-wise and effort-wise. To develop high quality software according to formal specifications software engineers need to be able to employ mathematical reasoning skills to analyze software components for correctness.

The important idea of teaching mathematical reasoning principles along with traditional SE topics in undergraduate computing has had a number of pioneers [16, 49, 54]. What distinguishes this effort in teaching mathematical reasoning from earlier efforts is that we identify and employ an inventory of analytical reasoning principles that need to be taught across the curriculum in undergraduate computing education to support correct, high quality software development. This work presents a set of necessary and sufficient skills that undergraduate students must acquire in order to prove correctness of typical data structures and algorithms they will develop and reason about in a typical computer

science curriculum. Proving correctness of such items as distributed databases and parallel programs is outside the scope of an undergraduate curriculum, and therefore is not being considered in this dissertation.

These reasoning principles are organized into a reasoning concept inventory (RCI). The RCI is defined and developed at a fine level of detail, so that an assessment program focused on learning outcomes can be derived from this inventory and used for educational evaluation. Then based on the direct evidence of student learning collected using these assessments, new or modified teaching materials and techniques can be developed to enhance student learning.

The idea of a concept inventory itself is not new to the STEM area. In 1992, Hestenes et al. noted that typical physics students were entering their classes with preconceived, 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 the concepts of the Newtonian physics, known as the Force Concept Inventory (FCI), which they believed to be a necessary part of every physics student's education [59].

Following the publication of the Force Concept Inventory, a number of concept inventories have been developed in a number of STEM disciplines, such as ones for chemistry [74], digital logic [58], electromagnetic engineering [118, 119], heat transfer [70], statistics [126], thermodynamics [100], and others. There is also an effort in

Discrete Mathematics that is focused on typical topics taught in that course, but its focus is not on software correctness [3].

The RCI differs from inventories in other disciplines in that whereas those inventories simply follow and document what has already been widely accepted in their field of study, it presents a path for educators to set teaching high quality software development for computing education as a goal 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 need to be learnt 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 inventory has been refined over multiple iterations through interactions among over 25 educators and researchers. It identifies the set of skills that are both necessary and sufficient, proven and self-evident, i.e., if you want a student to mathematically reason about the correctness of a piece of software, then it is essential that they develop the set of underlying skills captured by the RCI.

RCI is consistent with the computer science curriculum requirements addressed in the IEEE/ACM Computing Curriculum 2008, in particular, Software Engineering, Programming Languages, and Discrete Structures knowledge areas. Subsequent section provides details on how teaching the RCI reasoning principles also helps satisfy the Computing Curriculum requirements.

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 [7, 51, 52, 55, 75, 76, 79, 113, 114, 115, 116]. The hope is that they will use the RCI to shape the future of the 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 ~~"G~~raduates will be able to apply design and development principles in the construction of software systems," to ~~"G~~raduates will be able to apply design and development principles in the construction of defect-free software systems that function in accordance with specifications[1]." 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. The RCI is a natural starting point towards realizing these goals.

In the process of learning the reasoning principles detailed in the RCI, the expectation is that students will understand and appreciate intricate and important connections between mathematics and computer science. Integrating reasoning as a connecting thread among courses also helps students develop a cohesive view of the discipline as they first learn to develop software with introductory examples and objects,

[1] Dr. Mike Westall, Professor of Computer Science, Clemson University School of Computing, is credited with raising the motion to make this change.

and then move on into data structures and algorithms, programming languages, software engineering, and other courses.

The inventory in this dissertation is focused only on reasoning about software correctness, unlike the more ambitious effort in [48], which identified important and difficult concepts in the field of Computer Science using a Delphi process. While identifying the topics in a more general computer science context benefits from such a process, the RCI is simply a natural culmination of information from reasoning experts, both past and present. While the inventory is motivated by the past experiences of a number of educators, its goal is different from the goal of the previous inventories.

It is important to note that the RCI is not intended to incorporate all mathematical principles, or all software development principles that need to be taught in undergraduate CS education. Its focus is on those principles that relate to reasoning about software correctness. Ideally, these principles can be and should be taught as complementary ideas to other key CS principles.

Given this background, this dissertation aims to accomplish the following goals:

- To identify the central elements of a reasoning concept inventory (RCI) that are ideally taught across the curriculum in computer science to support high quality software development and refine it to a sufficient level of detail so that it forms a meaningful basis for developing learning outcomes;

- To develop learning outcomes for a subset of the RCI principles that will be

used to drive instruction at various cognitive levels across a sequence of computer science courses and identify improved instructional materials and methods for communicating the ideas to students; and

- To employ the RCI and corresponding learning outcomes for assessment to establish that analytical reasoning principles can be taught effectively in undergraduate CS education and that the RCI is useful for continuous improvement in computer science classes at Clemson and elsewhere.

The rest of the dissertation is organized as follows. Chapter 2 presents our motivation and details the research that has led to the RCI. Chapter 3 gives a research overview of how the RCI can be used across the curriculum to develop meaningful learning outcomes at various cognitive levels of the Bloom's taxonomy. Chapter 4 discusses our approach to assessment using the learning outcomes and contains a number of observations supported by the experimental data. Chapter 5 contains summary of the relevant related work on teaching formal reasoning. Chapter 6 lists ideas to be explored in the future, after the first milestone (this dissertation) has been completed.

Due to their large size, some of the exhibits, critical to the content of this dissertation, are included as appendices. Appendix A illustrates cognitive domain keywords of Bloom's taxonomy used to specify learning outcomes. Appendix B contains the complete multi-level Reasoning Concept Inventory. Appendix C gives an overview of RESOLVE—our selected medium for teaching specification and mathematical reasoning

principles. It provides the context necessary for giving examples to illustrate how specific learning outcomes can be assessed. A full set of the experimental data, separated by semester, course, and instructor can be found in Appendix D. A copy of the student survey questionnaire used in the experimentation, along with the consent form is located in E. Appendix F contains tabulated attitudinal survey data. Appendix G offers a full transcription of a Focus Group Meeting held with instructors who utilized RCI-based learning outcomes, instructional materials and assessment instruments. A listing of a subset of specific RCI items recently taught by different instructors at Clemson is located in Appendix H. Appendix I reproduces the full version of the Basic Reasoning Principles survey used in the professional computing community, while Appendix J includes the workshop survey from SIGSCE 2012 conference. Appendix K presents the results from the surveys taken by the professional computer science community. Appendix L contains a list of exercises that we have utilized to assess RCI reasoning principles at different levels of difficulty. These exercises can be used by instructors as homework, classroom exercises, or as assessment instruments of various RCI topics via tests and quizzes. A list of references concludes this work.

CHAPTER TWO

MOTIVATION FOR THE REASONING CONCEPT INVENTORY

This section introduces and motivates the Reasoning Concept Inventory. It shows how RCI reasoning principles are aligned with specific knowledge areas outlined in the IEEE/ACM Computing Curriculum 2008, specifically the areas of Software Engineering, Programming Languages, and Discrete Structures.

To be able to build high quality software that is reliable, failure-free and works as specified, a key desired outcome is that our students, the future software developers, will be able to reason about correctness of typical components they develop in their undergraduate education. To be able to reason about software correctness, computer science students need a set of skills that enables them to do so. The RCI documents a set of skills both necessary and sufficient for teaching reasoning about software correctness, and using the RCI provides educators with the framework of teaching these critical skills to their students.

2.1. Introducing the Reasoning Concept Inventory

In order to produce correct software students need to acquire a number of prerequisite skills that will be taught across the computer science curriculum. The RCI captures the central principles for reasoning about software correctness. It contains five different skill areas. Elements of each of these five areas and their relevance to software correctness are the topic of this subsection.

The RCI is a culmination of the combined effort of educators and researchers with decades of teaching experience. It is based on prior efforts of our research group and several other groups as well as feedback from educators at multiple institutions. The results of attitudinal surveys in the professional computer science community are presented in Appendix K. After several revisions, numerous meetings, and intense discussions, the reasoning principles to be learnt in order to build high quality software have been identified and categorized into five major areas. Each of these areas is further divided into a hierarchy of subtopics. Only the top two levels (reasoning topic and subtopic summary) are shown below in Table 1. The subtopics are further refined into *concept term highlights* (Level 3) and *concept details* (Level 4), and are shown in the complete multi-level version of the RCI in Appendix B.

The full version of the Reasoning Concept Inventory is also available at the following link: http://www.cs.clemson.edu/group/resolve/teaching/inventory.html.

The RCI skills are necessary, for example, if a software engineer with an undergraduate degree is expected to be able to prove correct any of the following: a piece of simple sequential code without a loop, sequential code containing a loop with accompanying loop invariants, or a recursive procedure. The RCI skills are also sufficient, because they document exactly the skills required to reach the goal of reasoning about a typical piece of software component or a system that is appropriate to the skill level of an undergraduate computer science student.

A suitable goal for most undergraduate students is to be able to write and/or verify a piece of code that will deal with sorting, searching, and processing of such data

structures as arrays, stacks, queues, and lists, and other tasks on the appropriate level of difficulty.

Reasoning Topic	Subtopic Summary
1. 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. An outline of the Reasoning Concept Inventory

Students cannot be typically expected to be able to prove correctness of a

distributed database system, or a software component that is designed for a multi-processor system. These skills are usually beyond the scope of a typical undergraduate computer science curriculum. However, it is necessary to ensure that at the end of their degree students are capable of verifying correctness of some non-trivial piece of code, the correctness of which is not apparent. The next few paragraphs examine the RCI areas and justify why students need skills in each of these areas.

The first area of the RCI is Logic (RCI #1). Mastering skills in the area of Logic is a prerequisite skill that will enable students to reason about program correctness when they master other skills from higher areas. It includes the understanding of standard logic symbols, proof techniques, and connectives, such as implication, quantifiers, and supposition-deduction. Later on, a variety of proof techniques will be used in conjunction with building reasoning tables, and for proving the verification conditions generated for a relevant piece of software.

Skills in the discrete math structures area (RCI #2) provide familiarity with basic mathematical structures, and enable students to model various software components with mathematical sets, strings, functions, relations, number systems, and other mathematical theories. If students do not understand the properties of a mathematical set, or a string, they may not be able to choose the appropriate model for a specific data structure at a later time. Students must understand the distinction between mathematical structures and their computer counterparts. For example, they must recognize that the integers provided by computer hardware are not the same as mathematical integers. There are ways to formally specify computer integers (with their associated bounds), so students can clearly

distinguish between that formal description and the typical description of mathematical integers found in discrete math textbooks. 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 will learn that finite sequential structures (e.g., stacks, queues, and lists) can be modeled by mathematical strings (with suitable constraints), but cannot be modeled by sets that lack order.

Precise specifications (RCI #3) are useful for reasoning about a client component without knowledge of the implementations that the lower-level component it is using. Students should understand the advantages of using the precise language of mathematical notation to specify software components. If they do not understand specifications, they cannot formally reason about the correctness of software components.

The 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. These types of specifications also build on the skills from the discrete math, logic, and precise specifications area.

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

Obviously, the skills in these areas are related. With respect to prerequisites, the RCI is organized in a linear fashion. For example, students need to understand logic (RCI #1) and discrete structures (RCI #2) to comprehend precise specifications (RCI #3). After having acquired those skills, students will move on to modular reasoning (RCI #4) and correctness proofs (RCI #5).

The RCI principles are intended to be taught across the undergraduate computer science curriculum. Different courses will incorporate a tailored set of these principles taught on the appropriate level of difficulty. These topics can be integrated into the existing course syllabi, and can be integrated with the traditional principles taught in software development and software engineering courses. The next chapter elaborates on this issue further.

Subsequent chapters also show how RCI-based learning outcomes are central in monitoring and improving student learning. The learning outcomes help instructors pinpoint exactly when students are learning, and when they need more instruction. A variety of methods that we have used to teach RCI principles is shown, and a collection of sample exercises are put at the disposal of educators. The data section provides convincing evidence that this approach works, and that it can be integrated with the traditional content.

2.2. RCI Relationship to ACM/IEEE Computing Curriculum

This section explains how the topics covered by the RCI are aligned with specific knowledge areas of the ACM/IEEE Computing Curriculum 2008. By teaching skills from

the five areas of the RCI instructors are not only instilling good software engineering practices in their students, but also simultaneously satisfying parts of the ACM/IEEE Computing Curriculum 2008.

The 2008 interim revision of the ACM/IEEE 2001 computing curriculum [69] includes, among others, knowledge areas entitled Software Engineering, Programming Languages, and Discrete Structures. This version of the interim report uses two-letter codes for the knowledge areas, but unlike the previous version, it replaces the sequence numbers with semantically meaningful identifiers, written as single lexical units as seen in the list below.

Under the Software Engineering area, there are core and elective subareas that contain topics related to formal specification, pre- and post-conditions, unit and black-box testing, etc. This work uses mathematical model-based specification techniques to create a focus, or a trajectory through these topics that allows us to neatly tie many of these topics together into a coherent package. For example, a subset of the specific topics listed in the 2008 Curriculum under the Software Engineering Knowledge Area can be correlated with the RCI-based instruction for software design.

- SE/SoftwareDesign
  - The role and use of contracts (RCI#3.2, #4.2)
  - Component-level design (RCI#4.2, #4.3, #3)

- SE/ComponentBasedComputing
  - Components and interfaces (RCI#3)

- Interfaces as contracts (RCI#4.2)
- Component design and assembly (RCI#4.2.2, #4.2.3)

- SE/SoftwareVerificationValidation
  - Testing fundamentals including test case generation and black-box testing (RCI#3.4.3)
  - Unit testing (RCI#4.2)

- SE/FormalMethods
  - Formal specification languages (RCI#3.2, #3.4)
  - Pre- and post-conditions (RCI#3.4.3)
  - Formal verification (RCI#4.1.1.3, #5.2, #5.3)

Similarly, below is the correlation of the RCI principles to the topics from the Programming Languages area:

- PL/AbstractionMechanisms
  - Modules in Programming Languages (RCI#4.1, #4.2)

- PL/ObjectOrientedProgramming
  - Encapsulation and information-hiding (RCI#4)
  - Separation of behavior and implementation (RCI#4)
  - Internal representations of objects (RCI#4.3)

The core and elective subareas in the Discrete Structures Knowledge Area contain topics related to basic logic, proof techniques, functions, relations, and graphs among

others. Below a correlation is shown between subsets from the 2008 Curriculum Discrete Structure Area are correlated with the RCI-based instruction.

- DS/FunctionsRelationsAndSets
  - Functions (RCI#2.5.2, #3.3.1.6)
  - Relations (RCI#1.5.1, #3.3.1.7)
  - Sets (RCI#2.2, #3.3.1.5)

- DS/BasicLogic
  - Propositional Logic (RCI#1.3.1)
  - Logical Connectives (RCI#1.2.1)
  - Truth Tables (RCI#1.2.1.3)
  - Predicate Logic (RCI#1.3.2)
  - Universal and Existential Quantification (RCI#1.2.2)

- DS/ProofTechniques
  - Direct proofs (RCI#1.5.1)
  - Proof by contradiction (RCI#1.5.2)
  - Mathematical induction (RCI#1.5.7)

- DS/BasicsOfCounting
  - Permutations and combinations (RCI#2.7)

- DS/GraphsAndTrees
  - Trees (RCI#2.7.1.5.)
  - Directed and Undirected graphs (RCI#2.7.2)

This subsection has detailed how the Reasoning Concept Inventory aligns with the ACM/IEEE Computing Curriculum 2008. It has shown how some RCI areas correlate with the Software Engineering, Programming Languages, and Discrete Structures areas of the Curriculum. However, because the Curriculum serves as a general guide to developing local computer science curriculums, it cannot adequately guide educators who are not familiar with reasoning principles for software correctness. On the other hand, because RCI has a fine granularity of skills specific to mathematical reasoning, it details the relevant principles in the Curriculum and includes principles that are not explicitly outlined in the Curriculum.

2.3. Chapter Two Summary

This chapter has presented the Reasoning Concept Inventory (RCI), a necessary and sufficient set of reasoning skiltls that enables students to mathematically prove correctness of a software component. The five knowledge areas (Logic, Discrete Mathematical Structures, Precise Specifications, Modular Reasoning, and Correctness Proofs) of the RCI are detailed. It is emphasized that the skills will be taught across the undergraduate computer science curriculum, and that the goal is to integrate them with the traditional content. By teaching RCI skills educators will simultaneously satisfy the requirements of the Software Engineering, Programming Language, and Discrete Structures knowledge areas of the ACM/IEEE Computing Curriculum 2008.

CHAPTER THREE

RESEARCH AND DEVELOPMENT OF LEARNING OUTCOMES USING THE RCI

The detailed Reasoning Concept Inventory (RCI) enumerates the reasoning skills that are necessary for establishing correctness of a typical piece of software and building high quality software in undergraduate computing education. The next milestone is to discover an appropriate paradigm for teaching these skills to students, determine which skills they are successfully mastering and which ones they are not, and delineate a way to assess them. For this, a detailed set of learning outcomes that cover a significant subset of the RCI areas at the appropriate level of difficulty has been developed.

Of the five RCI knowledge areas, current experimentation and assessment are concerned with the areas of Precise Specifications, Modular Reasoning, and Correctness Proofs. Most aspects of the first two areas (Logic and Discrete Math Structures) are routinely covered in the prerequisite courses that students take before starting a fundamentals of software engineering course. This effort concentrates on what students will learn in a sequence of the software engineering courses. At Clemson, such a sequence has been identified as CPSC215 (Software Development Foundations) and CPSC372 (Introduction to Software Engineering).

The undergraduate catalog provides the following description of the courses. Notice the inclusion of specification and reasoning principles in both of these courses, along with module design principles.

CP SC 215 Software Development Foundations: Intensive study of software development foundations. Advanced coverage of programming language primitives, function-level design principles, and standard development and debugging tools. Introductory coverage of module-level design principles, program specification and reasoning principles, and validation and verification techniques.
CP SC 372 Introduction to Software Engineering: Intensive introduction to software engineering. Focuses on each major phase of the software lifecycle. Introductory coverage of requirements analysis, requirements modeling, design modeling, and project management. Intermediate coverage of module-level design principles, program specification and reasoning principles, and program validation and verification techniques.

Developing learning outcomes for the other two areas (Logic and Discrete Math Structures) is left for a future effort, as described in Chapter 6 of this dissertation. Instruction based on learning outcomes for these other two areas will help to ensure that students come to software engineering courses with the necessary prerequisite skills covered. The next subsection shows how the learning outcomes for the RCI reasoning principles have been developed.

There are numerous attempts in the literature to inculcate new concepts in computer science curriculum and to assess student learning. In [28], the idea of outcome-based computer science education is emphasized, noting the importance of measurable outcomes. The importance of theory-based computing education is introduced in [37]. Challenges discussed in [91] include focusing education appropriately, defining curriculum that is forward-looking, and ensuring that software engineering educators have the necessary background. Measuring the effectiveness of specific types of projects

in software engineering education is discussed in [107]. This is a small subset of the vast array of publications that alert the software engineering community to the importance of providing essential background for students and finding ways to assess how well the resulting curriculum satisfies the needs of our community.

The multi-layer structure of the RCI makes it an ideal basis for defining learning outcomes. Several of the RCI topics, and hence the outcomes, are related to the outcomes in the corresponding topics of the 2008 IEEE/ACM Curriculum as discussed in the previous chapter. The learning outcomes specify 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 [36], quizzes, tests, etc.), and also serve as a tool for curriculum alignment. Because different RCI skills will be covered on the appropriate level of difficulty, it is only practical to develop learning outcomes that precisely correspond to that level of difficulty. Section 3.1 explains exactly how it is accomplished.

3.1. Classification of Learning Outcomes Using Bloom's Taxonomy

To address different difficulty levels that reflect a student's progress from an easy to a more complex level of difficulty of each reasoning topic, a sequence of learning outcomes for each topic have been developed. We have used a variation of Bloom's taxonomy [12], which normally consists of six levels of cognitive or intellectual outcomes listed from lower/easier level to higher/harder level for this purpose. Bloom's taxonomy has a longstanding history of being successfully applied in many areas of

education (e.g., [108]). It is well known to educators, and it has influenced all aspects of formal education, from how educational curricula are designed, to how student performance is evaluated in the classroom. Out of the three main domains of educational goals of the taxonomy: cognitive (about knowing), affective (about feelings and attitudes) and psychomotor (about physically doing), the cognitive domain is most applicable to this work. It relates to our efforts in developing the learning outcomes and the exercises that we have used to teach the material. The cognitive domain has a hierarchy of six levels [12]:

- *knowledge*: remembering/recalling previously learned information;
- *comprehension*: understanding the meaning of the previously learned information, plus the ability to describe, discuss, explain, generalize, and summarize;
- *application*: use previously learned information in some new and concrete situations, plus ability to assess, compute, construct, implement, and utilize;
- *analysis*: can break down previously learned information into constituent parts, identify motives and causes, plus make inferences and draw conclusions;
- *synthesis*: apply prior knowledge and skills to produce a new entity, plus categorize, reconstruct, and validate;
- *evaluation*: judge the value of the previously learned information, plus

compare and contrast, criticize and interpret.

Because there are many reasoning topics and subtopics, to avoid overwhelming our assessment instruments, the six levels of the Bloom's taxonomy are reduced to a simplified 3-level version by combining pairs of adjacent levels: KC (Knowledge-Comprehension), AA (Application-Analysis), and SE (Synthesis-Evaluation).

Bloom's taxonomy is widely known for its action verbs, known as domain keywords, associated with each level of difficulty. A number of keyword tables developed by educators can be found online and elsewhere, and are intended for different disciplines. We have used such a table of keywords (shown in Appendix A) while developing our learning outcomes (LOs) for the RCI.

The Bloom's taxonomy level to which a specific learning outcome belongs will depend on the verb used to describe the complexity of the performance expected from the students after learning the associated concepts. For example, *constructing* usually involves cognitive skill at the synthesis level (SE level), and is therefore usually a more difficult cognitive task than *calculating* (AA level), or *stating* a definition (KC level).

3.2. An Example Use of RCI to Develop Learning Outcomes

To illustrate the use of the simplified levels of Bloom's taxonomy to guide the development of learning outcomes from the RCI, consider the following example. To begin, the learning outcome for a specific RCI area is stated as a more general instructional objective. One instructional objective for RCI #3 – "Precise Specifications" may be stated as follows: "Apply precise mathematical specifications for software

components." Note that the verb "apply" used in this instructional objective has a broad feel to it. When designing instructional objectives and specific learning outcomes, a top-down approach is often utilized where instructional objectives specify behavior at a higher level, or at a broader scope, and learning outcomes specify behavior at a more fine-grained level. Specific learning outcomes are listed next and they are associated with the learning objective that indicates the types of student performances we expect, thus giving clarification to what is meant by the verb "apply".

Table *2* shows partial expansion of some sublevels of RCI#3 that illustrate these specific learning outcomes.

3.4. Specification of operations
3.4.1 …
3.4.2 …
3.4.3 Pre- and post-conditions
3.4.3.1 ...
3.4.3.2 Responsibility of the caller
3.4.3.3 Responsibility of the implementer
...

Table 2. Partial expansion of RCI #3

Based on RCI #3.4.3.3, a sampling of specific learning outcomes (LO) for various types of performance expected from the students may be given as shown below. Learning outcomes *LO1, LO2* and *LO3* correspond to the levels KC, AA, and SE.

- *LO1:* *State* the responsibility of the implementer of an operation with respect to the precondition. (Level KC)
- *LO2:* *Examine* an operation's pre- and post-conditions and *specify* a black-box test case based on those conditions. (Level AA)
- *LO3:* *Write* the post-conditions for a given operation. (Level SE)

Not all items in the RCI table have learning outcomes from all three levels. Each item only has as many learning outcome levels as appropriate for that item. For example, consider RCI # 5.1.1.2 (Soundness and Completeness), shown in Table 3.

5. Correctness Proofs
5.1 Motivation
5.1.1 Meaning of correctness
5.1.1.1 Semantics
5.1.1.2 Soundness and relative completeness
...
5.2 Construction of VCs
5.2.1 States and abstract values of objects
5.2.1.1 Naming conventions
...

Table 3. Partial expansion of RCI #5

RCI #5.1.1.2 only has learning outcomes at the KC level. The reason for this is that undergraduate students only need to be generally familiar with the notion of a proof system's soundness and relative completeness. More advanced skills are not required in association with this item:

- *LO*: *Discuss* the meaning of soundness and relative completeness. (Level KC)

These examples illustrate that the additional step in writing the LOs includes capturing the level of difficulty of the expected student performance. Once developed and detailed, the RCI and the corresponding software engineering learning outcomes at various levels are for use across the curriculum, from data structures and discrete structures to programming languages and software engineering. Please refer to the table with detailed learning outcomes and exercise examples in Appendix L

3.3. Learning Outcomes and Curriculum Alignment

The previous subsection provided two examples of how learning outcomes are developed for an area of RCI. This subsection explains how learning outcomes are used for curriculum alignment. In Table 4, the top row lists various undergraduate courses in a typical computer science department's curriculum progressing from 100-level courses on up to 400-level courses. The leftmost column lists the general learning objectives (e.g., RCI #3's "Applies precise mathematical specifications to software components"). The body of the matrix captures where in the curriculum specific learning outcomes are expected. It also indicates the expected difficultly level, as we fill in the matrix using the two-letter Bloom's taxonomy codes KC, AA, and SE. For example, the row with the general learning objective "3.4.3 Precise Specifications" indicates at least three specific LOs: with difficulty levels KC, AA, and SE. The KC learning outcomes appear in a 100-level course, the AA in both the Discrete Structures course and a 200-level course, and the SE level in the Software Engineering course. Some other RCI skill "Skill *N*", for

example, may have the KC learning outcome on both 100-level and Discrete Structures course, AA level on a 200-level course, and SE level in a Software Engineering Course.

For the collection of RCI principles, participating CS courses (i.e., the top row in Table 4) are expected to include (among others): discrete structures (math), foundational software development courses (e.g., CS1, CS2, and advanced data structures), analysis of algorithms, software engineering, etc. This table is a general example, not geared towards Clemson's, or any other specific institution. Every department will have their own version depending on their undergraduate curriculum and course offerings, and type of students attending (e.g., mainly students going into the workforce, or students going on to graduate school, etc.).

The idea of outcomes-based CS education is also emphasized in [28], where the authors note the importance of measurable outcomes, which is one of our goals in developing the inventory. The inventory-based approach also addresses some of the challenges discussed in [91], where the authors focus on defining a curriculum that is forward-looking and ensuring that software engineering educators possess the necessary background.

Skills\Course Level	1XX	Discrete Structures	2XX	…	Software Engineering
Skill 1		KC	AA		
3.4.3 Precise Specifications	KC	KC, AA	AA		AA, SE
…					
Skill N	KC	KC	KC, AA		SE

Table 4. Learning outcomes and curriculum alignment

Section 3.3.1 demonstrates how RCI and LOs are used as a framework for developing instructional techniques and materials. Chapter 4 details the assessments, and how the results of these assessments are utilized as a feedback mechanism for making improvements in the instructional materials and adjustments in the amount of time spent on particular concepts. Ultimately, the goal is to utilize these learning goals and outcomes as a tool for continuous curriculum improvement and for curriculum alignment, i.e., making sure our undergraduate computer science courses collectively and effectively work together to cover important reasoning-related topics. These reasoning-related topics will be taught within the existing course framework, i.e., they can be integrated into the existing syllabus.

3.3.1. Connections between RCI Learning Outcomes and Learning Objectives of the ACM/IEEE Computing Curricula 2008

The basics of developing learning outcomes for our RCI have now been demonstrated. Utilizing RCI and LOs will not only help instructors to pinpoint what students are learning and what they are not learning, but also to make a connection with the more general Learning Objectives outlined in the ACM/IEEE Computer Science Curriculum 2008. The Interim Review Task Force points out that the learning objectives are often considered central to curriculum design, because they indicate what students will be able to do with the knowledge, and encourage students to make effective use of their knowledge as it becomes available. The Curriculum differentiates between the terms ―learning objective" and ―learning outcome", stating that the former phrase has an ―aspirational connotation", and the latter mandatory, implying that students are in fact

expected to have learned something. The Curriculum focuses on learning objectives instead of outcomes, and lists four types of learning objectives:

- *benchmark Learning Objectives* are intended to capture the characteristics that describe what is expected from a student on any computer science program of study;

- *program Learning Objectives* are associated with programs of study and so capture at a high level the anticipated behavioral characteristics of graduates;

- *class Learning Objectives* are associated with individual classes and so indicate the contribution that each class makes to the program Learning Objectives though usually in sufficient detail to guide intended participants (lecturer, students, etc.);

- *instructional Learning Objectives* are associated with knowledge units or topics and so include in great detail the expectations relating to individual parts of the class.

The emphasis of this work is on the learning outcomes instead of the objectives. The purpose of the Curriculum is to provide *guidelines* for the development of a local computer science curriculum by each computer science department. Our effort emphasizes learning outcomes because we expect students to learn specific skills to reason about correctness of software, and by measuring their performance on these skills educators can make necessary improvements and/or adjustments in the content of the course/s that intends to deliver those skills across the entire curriculum.

The Curriculum offers a different treatment of the 6-level version of Bloom's Taxonomy, and associates benchmark and program objectives with the higher levels of this taxonomy, and instructional objectives with the lower levels. Our learning outcomes are based on a modified 3-level version of the taxonomy (KC, AA, SE), and are written in such a way that they include all cognitive levels. As demonstrated earlier, our learning outcomes allow students to progress through a hierarchy of cognitive domains while learning a specific skill.

The learning outcomes in this work are more detailed than the Task Force's learning objectives. For example, the Curriculum lists only four learning objectives that cover the entire SE/Formal Methods area, as shown in Table 5.

The learning outcomes shown in an earlier subsection not only have three levels corresponding to the three levels of Bloom's taxonomy, but they are also gauged to assess one specific skill. The ACM/IEEE Curriculum addresses the entire subarea of formal methods in a small number of general objectives.

Because the Computing Curriculum intends to simply guide local CS departments in the development of their computer science curricula, it does not provide a detailed list of learning objectives. Instead, it provides a small number of them to serve as a starting point, should these departments find it necessary to expand it into a detailed list of learning objectives, customized to their institution's needs. And such expansion of the learning objectives for developing high quality software is what is being accomplished in this dissertation.

- Apply formal verification techniques to software segments with low complexity.
- Discuss the role of formal verification techniques in the context of software validation and testing, and compare the benefits with those of model checking.
- Explain the potential benefits and drawbacks of using formal specification languages.
- Create and evaluate pre- and post-assertions for a variety of situations ranging from simple through complex.

Table 5. Learning objectives for the SE/Formal Methods area (Computing Curriculum 2008)

3.4. Methods of Communicating RCI Principles to Students

Section 3.2 demonstrated how learning outcomes for a subset of RCI areas have been developed by using modified levels of Bloom's taxonomy. This section identifies the methods of communicating them to our students. Methods that have worked well at Clemson are shown next for the purpose of illustration, without an intention to prescribe to other instructors what methods are best suitable for teaching certain skills. Some custom tools have been more effective for teaching certain skills, while other skills can be learned using traditional teaching methods (lectures, quizzes, tests). They are described in section 3.4.1, and are available free online to anyone who would like to use them.

3.4.1. Providing Motivation for Learning RCI skills

Among a variety of things that help students learn, motivation is one of the most important. Even though this is not a tangible teaching tool, it is still a very important component in communicating to students exactly what they will learn and why. Students need to understand why these individual skills will enable them later to perform complicated tasks. To facilitate the process, each of the 5 knowledge areas of RCI contains a Motivation subsection, which gives students adequate explanation of why they need certain skills. Table 6 below illustrates the three areas that are included in our assessment.

Students need to understand the importance of interfaces and precision before they learn to specify various software components, for example. They need to understand why they must be able to analytically reason about software components, and why modular reasoning is an indispensable skill, before they are able to build a reliable software system from components produced by different developers separated from each other chronologically and geographically.

RCI Area	Subarea	Details
3. Precise Specification	3.1 Motivation ...	3.1.1 Motivation for interfaces 3.1.2 Motivation for precision
4. Modular Reasoning	4.1 Motivation ...	4.1.1 Motivation for reasoning 4.1.2 Motivation for modular reasoning
5. Correctness Proofs	5.1 Motivation ...	5.1.1 Meaning of correctness 5.1.2 Motivation for proofs

Table 6. Motivation sections for each subset of RCI

The meaning of correctness and an understanding of how useful proofs can be developed will prepare students to learn to prove correctness of code. When students understand the purpose to their effort, their learning and attitudes improve. The results of the student attitudinal survey are discussed in section 4.4.7.

3.4.2. Teaching Tools and Teaching Medium

The initial method of delivery is classroom lecture in an interactive environment, where students actively engage in problem-solving activity. Short sections of lecture are followed by an instructor-led group discussion, or a single-student activity. Tests and quizzes are traditionally used, and homework is assigned. Collaborative classroom exercises and reasoning tools are also used to teach various combinations of RCI principles. Detailed discussion is available in [25, 36, 121].

Though teaching RCI principles in our classrooms are not based on a specific programming language in CPSC215, a number of Java examples are used because students are already familiar with Java. In the junior-level software engineering course CPSC372 the RESOLVE integrated environment is used as a teaching medium.

Appendix C offers a detailed discussion on RESOLVE. RESOLVE is our choice because it includes a built-in specification language and because it is supported by a web-integrated development environment that already contains a verifier. It provides ready support for software engineering principles, such as abstraction, information hiding, modularity, and others. However, any modern object-oriented language coupled with a suitable specification and reasoning paradigm, such as Dafny [14], can be used to teach

the RCI principles. In fact, though some features of RESOLVE are introduced in CPSC215, that course is taught using Java. These details will be covered in more detail in a later section.

A number of effective teaching methods/tools that we have used to communicate RCI principles to students are demonstrated next. Some of these teaching tools are best illustrated via the RCI principles, as they have been designed specifically to teach these principles.

3.4.2.1. Sample Exercises Corresponding to Example Topic RCI 3.4 (Specification of Operations)

A number of interesting and challenging exercises have been designed to teach various RCI principles. Please refer to Appendix L for a complete list of sample exercises with their corresponding learning outcomes. Using a number of interactive exercises is very helpful in strengthening the principles that are taught. Earlier, a partial expansion of RCI #3.4 (specification of operations) was presented along with some specific learning outcomes. Figure 1 contains a sample exercise, used to teach RCI #3.4 that deals with teaching specification of operations. In the specification a *Queue* of *Entry* is conceptualized as a mathematical string of entries. In the *ensures* clause, Q denotes the outgoing value and #*Q* denotes an incoming value; in string notation "o" denotes concatenation.

Students produce two test points for each operation to show that they understand the operation's pre- and post-conditions. A specification-based test case for an operation

consists of a pair of inputs-outputs for the formal parameters to the operation. A valid input is determined by the *requires* clause of an operation, and the expected output is determined by the *ensures* clause.

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>;
```

Figure 1. An exercise to assess RCI #3.4 at the AA level

To further strengthen the principles we have used *non-descriptive operation names* (e.g., *Mystery_1( )*, *Operation_1( )*, *Guess( )*, etc.), so students do not attempt to guess what an operation does from its name, but instead, examine its formal specifications. By answering this exercise correctly, students demonstrate they have met the specific learning outcome on the AA level. Using learning outcomes as a guide, the exercises are tailored to address exactly the principles that are being taught.

Another sample exercise is presented in Figure 2**.** Notice that the pre- and post-conditions are specified in plain English instead of any specification language. This question is given to students of the sophomore-level Software Development Foundations course (CPSC215) where only basic concepts of design by contract are taught. This

question assesses RCI#3.4, and specifically RCI#3.4.3 (pre- and post-conditions at the SE level).

Examine the following informal specification and answer the two questions below using the notation shown in class.

```
public void mystery(Sequence s1, Sequence sub, Sequence s2, int p);
```

requires:

- sub must be a substring of s1
- p must be less than or equal to the length of s2

ensures:

- sub will be removed from s1; no other changes made to s1
- sub will not be modified; it will be inserted at position p within s2
- no other changes made to s2
- p will not be modified

example:

#s1=<1,2,3,4,5>; #sub=<3,4>; #s2=<1,2,3>; #p=1

s1=<1,2,5>; sub=<3,4>; s2=<1,3,4,2,3>; p = 1

Q1. Provide the formal pre-condition for this method.

Q2. Provide the formal post-condition for this method.

Figure 2. A sample exercise to assess RCI #3.4 at the SE level

3.4.2.2. Test Case Reasoning Assistant for Topic RCI #3.4 (Specifications)

The *Test Case Reasoning Assistant (TCRA)* is used to teach specifications, and particularly test cases. This tool takes a student through a series of test-case creation exercises with rapid feedback [90]. TCRA has been developed at Clemson has been successfully used in CPSC372. It has a simple user-friendly interface with many useful

features. Available free online at (http://www.cs.clemson.edu/resolve/teaching/spec-understanding.html), the tool has two interfaces, student interface (Figure 3) and instructor interface (Figure 4). The student interface allows students to create and test a variety of test cases for an operation, and has a large number of exercises built-in into this tool.

The interface is divided into four windows. Students can select an operation from a template in the window on the top left. The screen shot in Figure 3 shows the operation *Mystery_3* from the Stack_Template which was first selected.

The operation and its specifications are shown in the window on the top right, where the *ensures* and *requires* clauses can be examined. This particular operation can only be called if the size of both Stacks *S* and *T* is greater than zero. The post condition includes an existential clause. It ensures that the value of the outgoing reversed Stack *S* concatenated with the outgoing Stack *T* is the same as the value of the incoming reversed Stack #S concatenated with an incoming Stack T. Integer E is used to hold the Integer that was removed from the incoming Stack #S.

The test case based on these specifications is entered in the table in the bottom left. The column "Argument Name" contains incoming (Stacks #*T* and #*S*) and outgoing (Stacks *T* and *S*) variable names. The "Value" column displays the actual test case values for these variables that the student provides. Syntax checking is done, and errors are reported. In this example, the incoming and outgoing values are: Stack #*S* is <1, 2, 3> and #*T* is <4, 5, 6>, Stack *S* is <2, 3>, and *T* is <1, 4, 5, 6>. After the "Ok" button is clicked, an *instantiated specification* is produced, by substituting the original

specification with the student- provided values. If the ensures and requires assertions evaluate to true the student is notified that the test case is correct. In this example, the test case is correct: the values entered satisfy the *ensures* clause, since the concatenation of a reversed S (<2, 3>) and T (<1, 4, 5, 6>) produces <3, 2, 1, 4, 5, 6>. The result is the same as concatenating the incoming reversed *#S* (<1, 2, 3>) and *#T* (<4, 5, 6>).

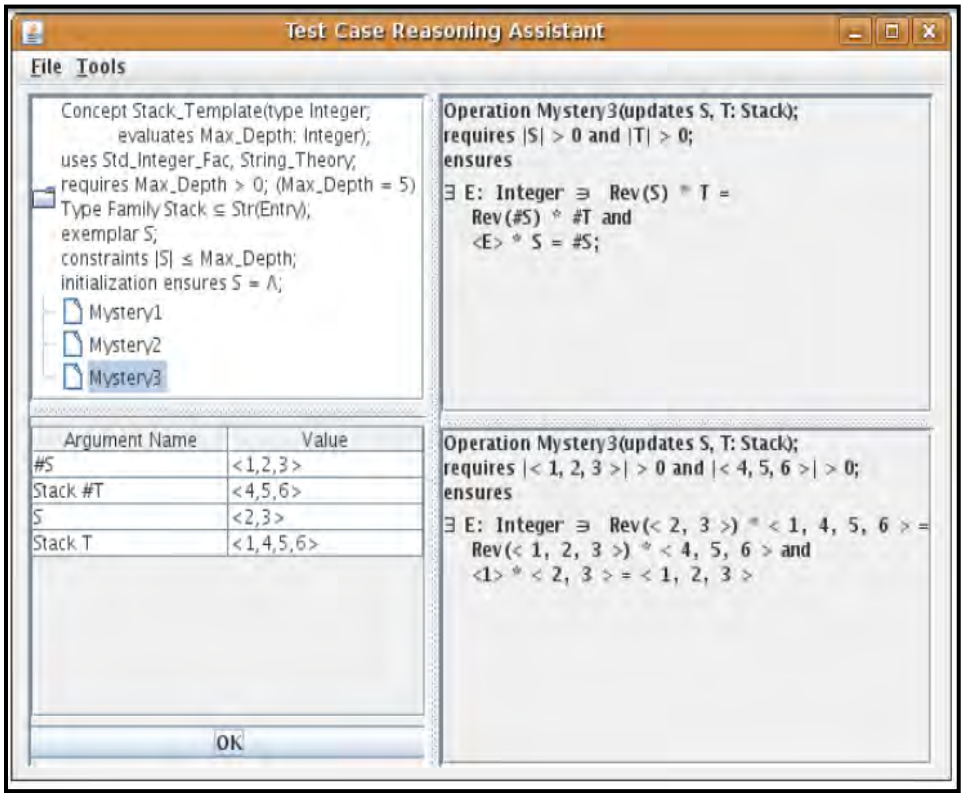


Figure 3. TCRA student interface featuring an exercise

The instructor interface is also very simple, and is shown in Figure 4. It has drop boxes that allow the instructor to select a subset of exercises or create new ones. The interface provides tools for analyzing student activities, such a graph generator.

Instructors monitor student work from the (anonymous) logs that are created during student activity. The logs that have an option of being submitted on and off campus, record what interface/operation is selected, and the correctness of test cases provided. The graphs visually display student progress.

The TCRA tool, detailed in [90] has been custom built to teach operation specifications through the test case generation, specifically the inventory principle RCI#3.4, and serves its purpose well.

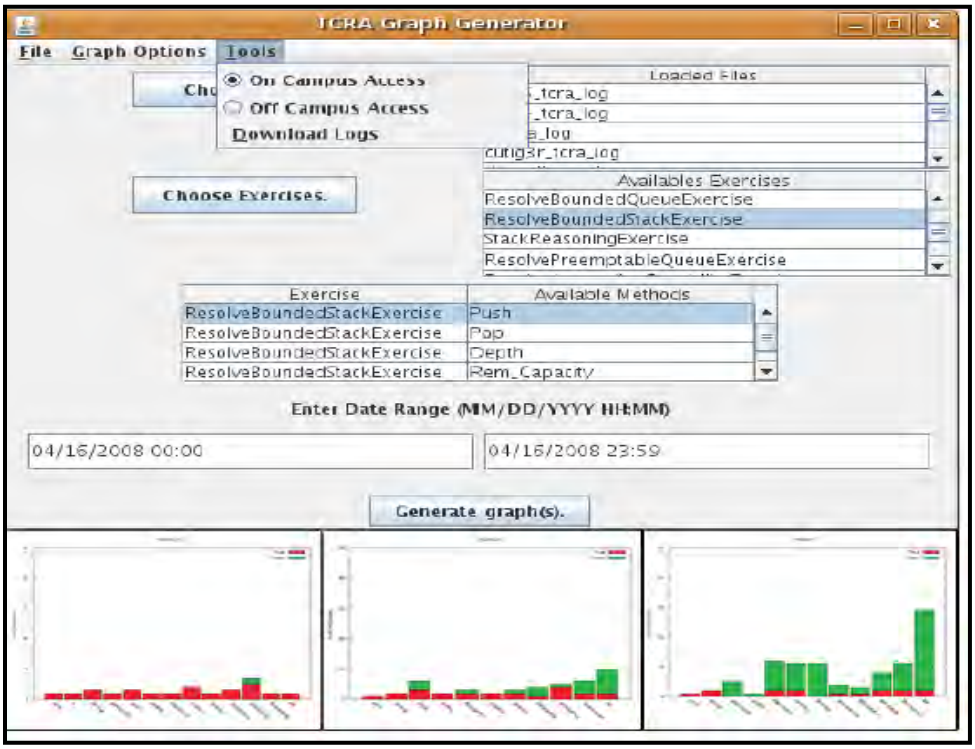


Figure 4. TCRA instructor interface featuring exercise selection

Research [5, 32, 46, 84, 85, 104, 127] in the past few decades has shown that tutorials are effective educational tools that can be used to supplement classroom

instruction or provide independent learning experience. Interactive online tutorials which provide students with instant feedback are of particular importance and have been shown to greatly benefit learning.

3.4.2.3. Online Tutorial

To teach a number of specification-related topics an *online tutorial* discussed in [36] has been used. The tutorial is available at the following link: http://www.cs.clemson.edu/group/resolve/teaching/tutor/index.htm.

The tutorial contains three modules: Mathematical Strings, Understanding Specifications, and Understanding Test Cases. All modules introduce the material as using a small number of clearly presented slides progressing from easy to more difficult concepts. The interface is intentionally simple, so that students do not spend time learning how to navigate the environment.

The mathematical string module introduces the basics of string notations. Another module, Understanding Specifications, explores operation parameter modes and explains the ideas of redundant and equivalent specifications. It introduces RESOLVE's use of mathematical modeling for specifying various data structures and their behavior, with operation specification being one of its specification mechanisms. The short ―info" slides are usually followed by eight to ten multiple choice exercises which provide instant feedback. Figure 5 shows a simple ―info" slide discussing redundant specifications. It explains the concept in simple terms and provides an example, along with the options of

moving to the previous, or next slide, and completely skipping the tutorial to move to the exercise section.

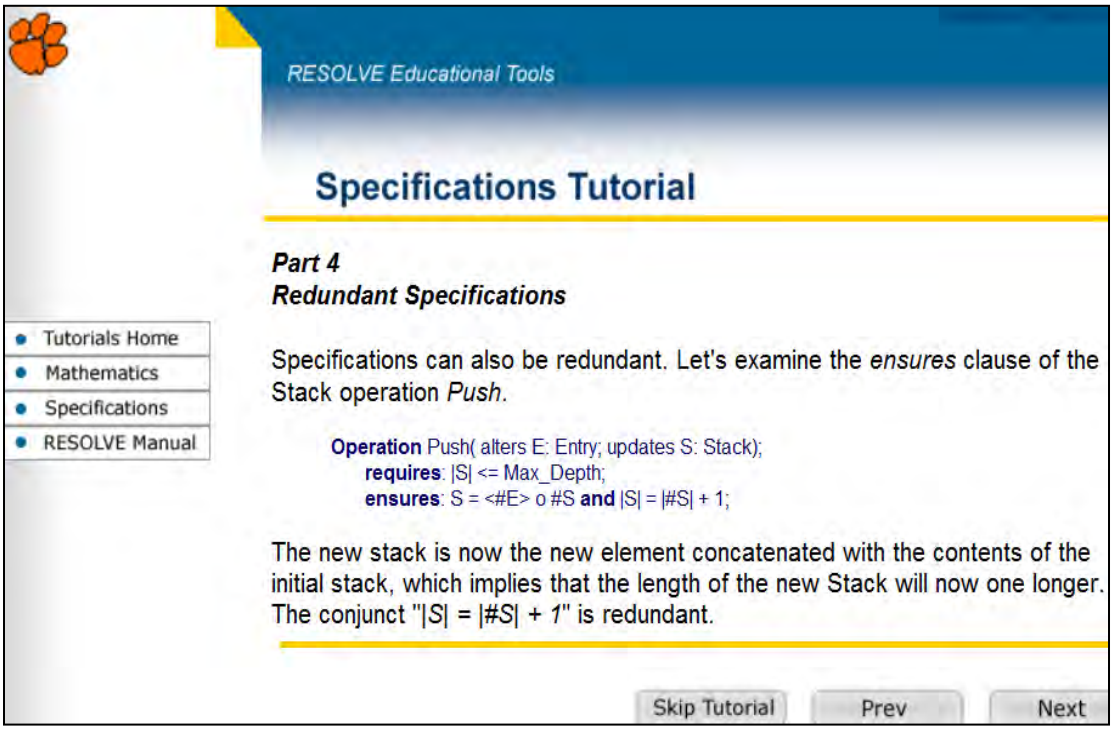


Figure 5. Tutorial screenshot containing a sample "info" slide

After selecting an answer, students can check the correctness of their choice by clicking on the "Check Answer" button. The feedback is displayed on the right-hand side. In this case, a student had to examine the operation *PopCard(),* and determine the outgoing values of the Stack *S* and Card *MyCard*. This exercise tests the knowledge of RCI#3.4.3.1 (Specification parameter modes), and the parameter modes "updates" and "replaces" in particular. On this slide, a feedback is shown for an incorrect student response.

The third module is Understanding Test Cases that provides tutoring for the understanding of specifications by using test cases. Mastering RCI#3.4 (Specification of operations) requires several prerequisite skills, starting with mathematical notation, understanding basic specifications, and finally test cases.

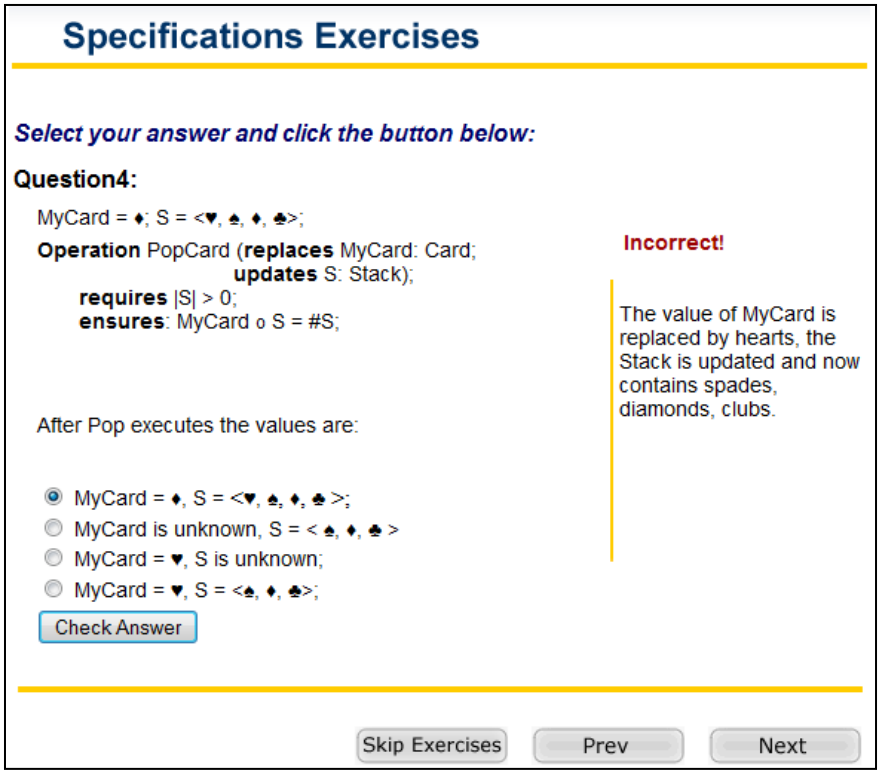


Figure 6. Tutorial screenshot containing a sample exercise slide

3.4.2.4. Multifunctional Web Integrated Development Environment

The Web Integrated Development Environment (Web IDE) has been developed at Clemson [27] 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 makes it a

very useful tool for teaching a variety of reasoning principles at different difficulty levels. To better understand what the IDE has to offer, please refer to the discussion of RESOLVE components in Appendix C.

The IDE features a Component Finder to allow selection from a large number of RESOLVE concept specifications, ranging from simple ones, such as *Integer_Template* and *Stack_Template*, to more complex ones, such as *Map_Template* and *Prioritizer_Template*. It also offers a variety of realizations (implementations), enhancements (concept extensions), enhancement realizations, and facilities (~~–m~~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 ~~–b~~ehind 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 7 and Figure 8 illustrate the web IDE. Screen shots have been cropped to emphasize only certain areas of the interface. Figure 7 shows part of the interface window with the tabs for the four selected components used in a sample student project. The open window contains the actual RESOLVE code, with several modules open in their individual tabs: *Obvious_Flip_Realiz* realization of the *Flipping_Capability* enhancement of the *Stack_Template*'s *Array_Realuz* realization, with the options below the tabs to generate VCs and verify them. When the VCs option is selected, the VCs are generated and displayed on the right side of the window. Students can scroll down and examine each of them.

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. In Figure 7, for example, VC: 0_2, displayed in a pop-up box on the bottom left, was generated on line 10 for the *requires* clause of *Pop(Next_Entry, S)* that removes *Next_Entry* from the Stack *S.* In the right-hand window the generated verification condition for VC: 0_2 is shown.

When the Verify option is selected, the IDE attempts to prove the generated VCs, and displays the results on the right. VCs that successfully verified are marked by a green dot, as shown in Figure 8. Failed VCs will be marked by a red dot. In this example, no VCs have failed. 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.

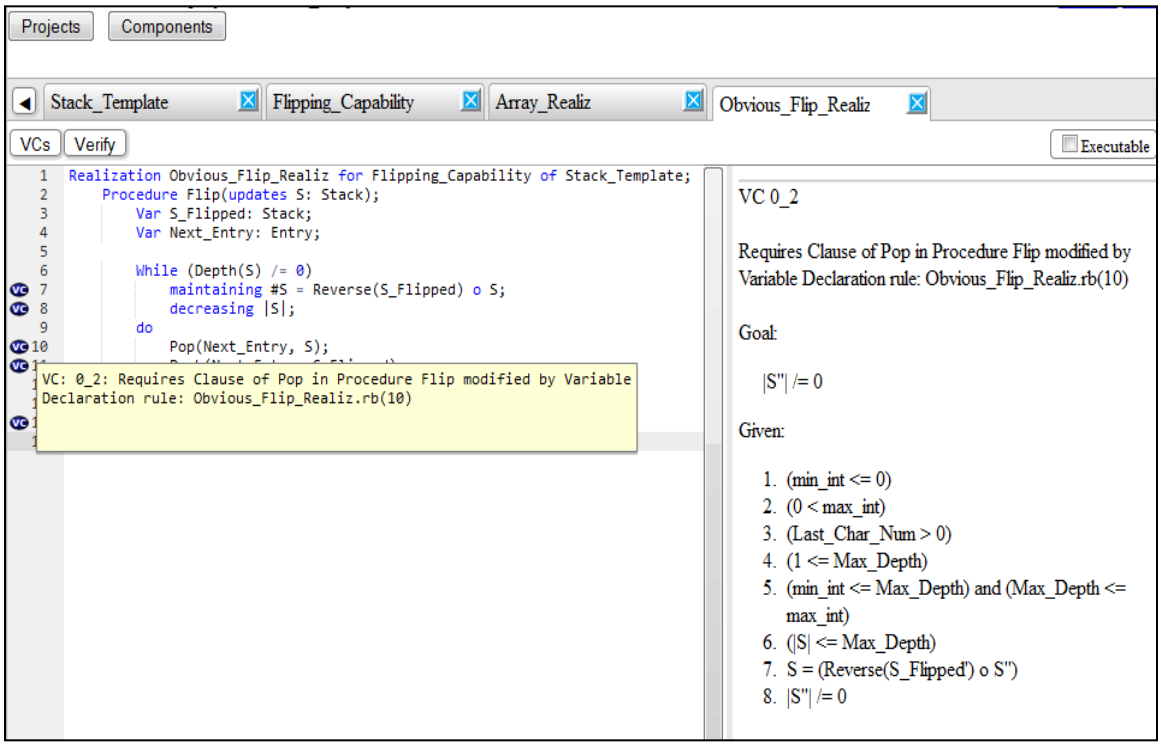


Figure 7. Example Verification Condition (VC) generation with the Web IDE

For example, 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).

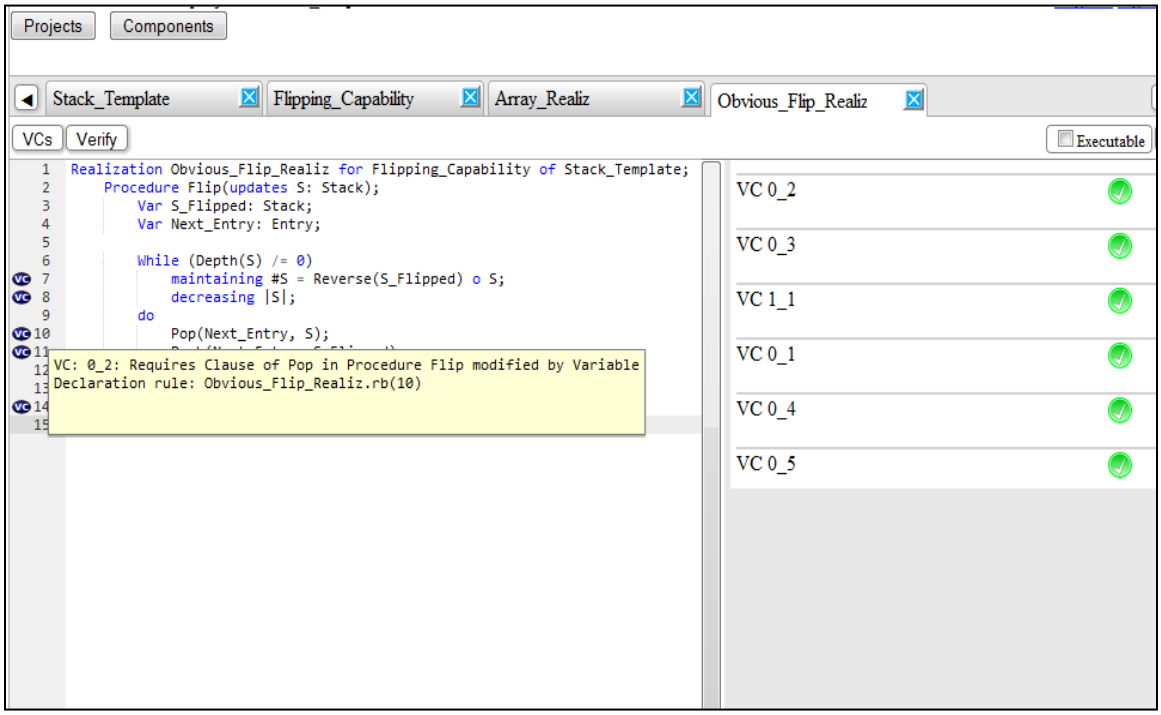


Figure 8. Verifying VCs with Web IDE

3.4.2.5. Sequence of Interactive Online Videos for Topic RCI #5.2

A sequence of instructional videos has been created with the goal of teaching students to prove correctness of the code for a simple operation. This is a new reasoning skill taught at the advanced level of difficulty, and requires that students possess a

number of prerequisite skills. In order to prove the correctness of a code, students must generate verification conditions (VCs), and to generate the VCs they must build a reasoning table with assumptions and obligations.

A sequence of three videos has been created by the professor Joseph Hollingsworth from Indiana University Southeast. These videos are available free online via the www.youtube.com, and on the RESOLVE project website: http://www.cs.clemson.edu/group/resolve/teaching/ed_ws/sigcse2012/index.html.

Each video is a step-by-step demonstration that teaches students a variety of skills from the RCI #5 area (correctness proofs). The three videos cover building a reasoning table with assumptions (RCI#5.2.2.1) and obligations (RCI#5.2.2.2), generating verification conditions (RCI#5.3.1), and proving these VCs (RCI#5.3.2), correspondingly. The videos are short (each about 5-6 minutes long), present only relevant details, and students can watch them from the beginning to the end without losing focus. The recorded demonstration starts with a blank piece of paper, and ends up with the completed correctness proof of the code. The operation used in this demo is simple, and only increments and then decrements an integer. Screenshots of the videos are shown in Figure 9, Figure 10, and Figure 11.

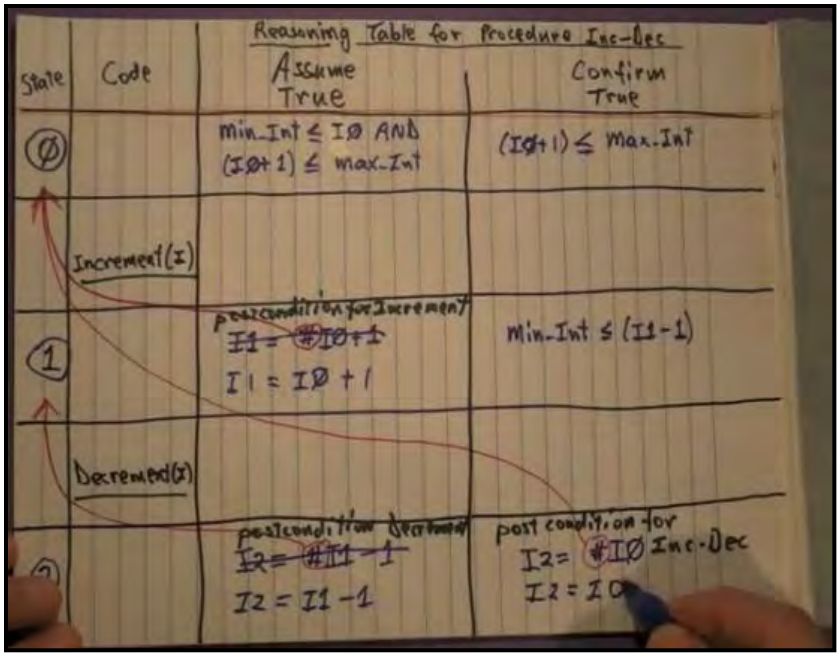


Figure 9. Video on creating a reasoning table

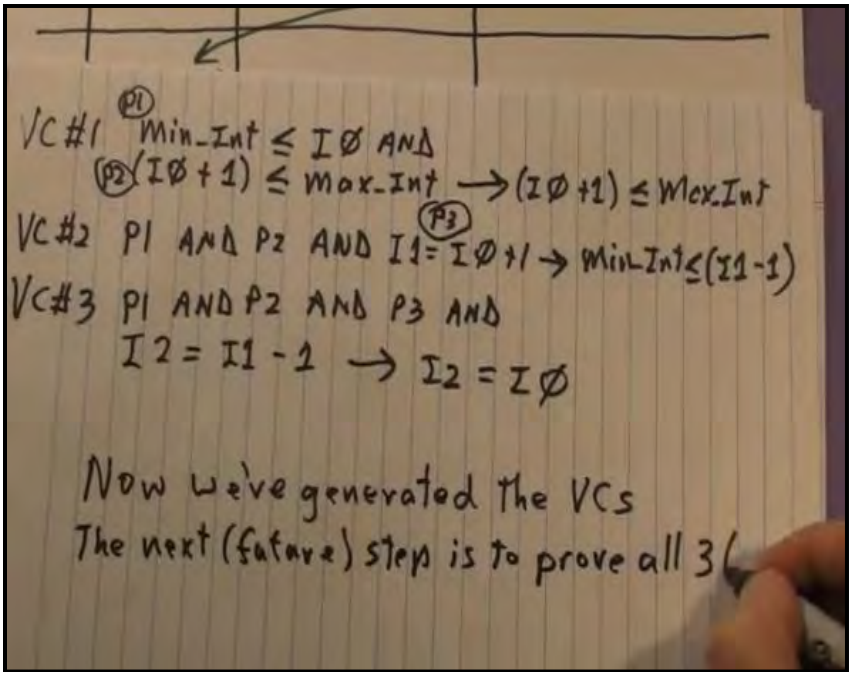


Figure 10. Video on Generating VCs

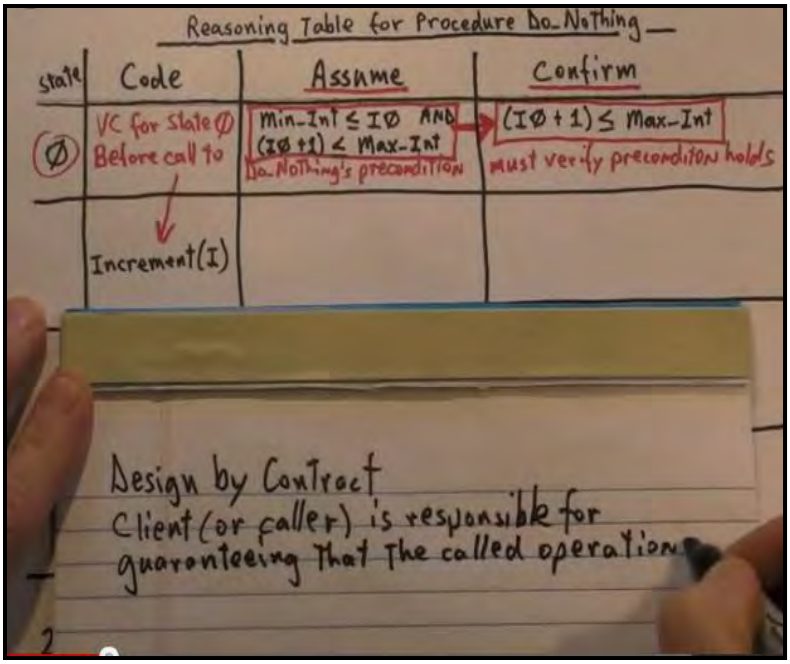


Figure 11. Video on Proving VCs

3.4.2.6. Team Project Assignments

A number of project assignments that encourage students to work in small groups to develop software modules according to given formal specifications have been routinely used. The intent of such a project is to give students a hands-on experience in component-based software development and reasoning, while applying the principles they have learned in the classroom settings. Such principles include, among others, RCI#4.2.2 (construction of new components from built-in components), RCI#4.2.3 (construction of new components using existing components), and RCI#4.3.1 (internal contracts for data representations). A number of prerequisite principles, such as RCI#3.4 (operation specifications) are also learnt in the process.

Modern software industry encourages developers to reuse existing software components that may have been built by engineers separated geographically and chronologically. Project teams routinely consist of software developers living thousand miles from each other and speaking different languages, and individual software components are often outsourced. It is important to provide graduating students with a similar experience in the educational settings before they are assigned such a task in industry. A key educational goal is to demonstrate to students working in teams that modules developed by different developers can be seamlessly assembled in a software system and can operate flawlessly, provided they strictly adhere to the formal specifications, and no violations have occurred. A sample component relationship diagram that illustrates the complexity of such an assignment is shown in Figure 12.

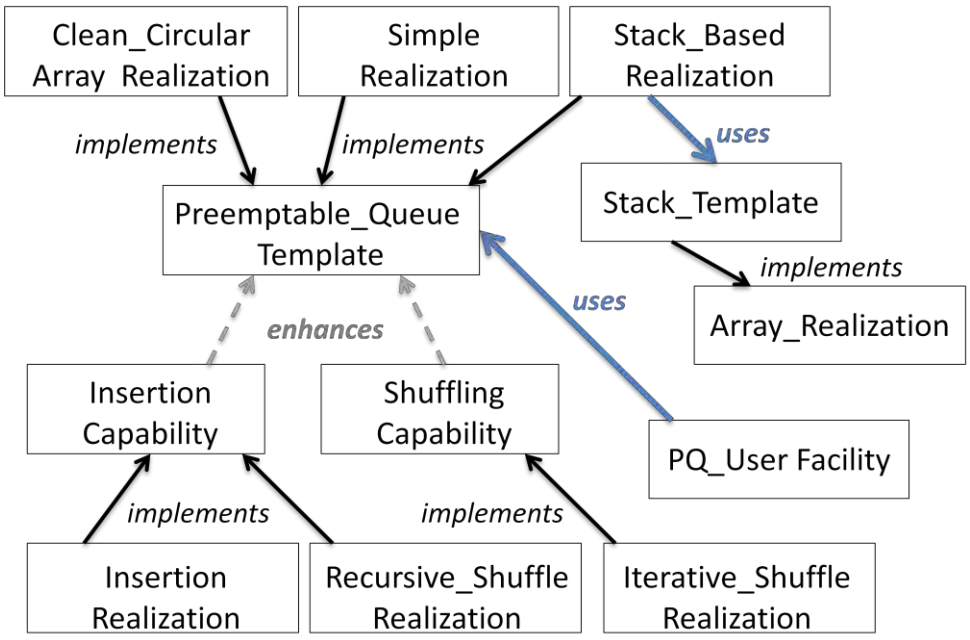


Figure 12. Component Relationship diagram for a sample project

In the example assignment shown, students use the *Preemptable_Queue* template to develop three different realizations - *Clean_Circular_Array_Realization*, *Simple_Realization*, and a *Stack_Based_Realization*, the latter of which uses *Stack_Template*. Two enhancements: *Insertion_Capability* enhancement, and *Shuffling_Capability* enhancement are developed along with their realizations. The last module to be produced is *PQ_User* facility, which is the main program.

Though on the surface this type of an assignment appears identical to any software engineering project assigned in every computer science department, a different methodology is used to ensure that students produced working software components. The component specifications include formal, external and internal contract specifications. To test for external contract violations, complete modules developed by different developers are assembled in a software system, compiled, executed, and the results are examined. For example, a simple scenario integrates the *Clean_Circular_Array*_Realization for the template developed by group #1, *Insertion_Capability* enhancement designed by group #2, *Recursive_Shuffle* enhancement realization coded by group #3, and *PQ_User* facility written by group #4.

To test for internal contract violations, different module operations implemented by different developers are extracted from the original file and combined together in a new file to check for consistency of the internal representation. This sample scenario uses a *Simple_Realization* template realization, and includes operation *Enqueue()* developed by group #1, operation *Dequeue()* written by group #2, operation *Inject()*

written by group #3, and *Swap_Last_Entry()* from group #4. These operations are then combined in a new realization file, compiled, and run.

A goal of many software organizations is that software components can be developed by different software engineers at different times and in different localities, then if these components strictly adhere to the formal specifications, they can seamlessly interoperate no matter who developed them and when. We simulate this in the classroom setting. The instructor randomly selects modules from different groups of –software developers" to be used in the real-time classroom experiment. What makes this assignment especially exciting and attractive to students is that the experiment takes place in the classroom, in front of all students. This experiment is not staged, the results are not known in advance, and only become apparent when the instructor compiles the modules in front of the class, runs, and examines the results.

While correct results (e.g., code compiles correctly and correctly executes on all supplied test cases) offer instant gratification, the incorrect results teach a valuable lesson as well: students have to determine which module has violated the specifications and devise a way to fix it. In either case, students learn the importance of strictly adhering to formal contracts. Though the scale of the project is usually rather small, and the data structures used are already known to students, the real-life software development process is what makes this assignment especially interesting to our students. The anecdotal evidence indicates that students are excited to see the role of contracts in team development.

3.5. Chapter Three Summary

This chapter demonstrates the process of developing learning outcomes for a *subset* of the RCI principles to be taught at various cognitive levels in a sequence of software engineering courses. For assessment, we have focused on the last three areas of the reasoning concept inventory. The first two areas of the RCI are not considered in this work because these skills are taught in prerequisite courses (Discrete Math, Logic, etc.) that students take before they enroll in software engineering courses. The connections between the learning outcomes presented in this effort, and the IEEE/ACM Computing Curriculum 2008 have also been explained.

The methods used to communicate RCI principles to our students have been discussed. Having motivated students to learn the RCI principles, we have used several traditional methods (lecture, quizzes, and tests), along with custom tools tailored for teaching specific skills, such as custom-designed exercises, TCRA, Online Tutorials, Web IDE, a sequence of instructional videos, and exciting hands-on projects. All of these artifacts are based on the concepts outlined in the RCI and have helped to improve student learning of these concepts.

The next chapter presents relevant details of our extensive experimentation and the results from assessment over 4 years involving 17 classes and 346 students.

CHAPTER FOUR

ASSESSMENT USING RCI-BASED LEARNING OUTCOMES

This chapter explains how the Reasoning Concept Inventory, along with the learning outcomes and methods for their instruction, forms the basis for experimentation, data collection, evaluation, and improvement. The experimentation involves two undergraduate software engineering courses CPSC215 and CPSC372 at Clemson and the software engineering course CS315 at University of Alabama.

Other institutions have also introduced reasoning principles at different levels of difficulty in one or more of their computer science courses: 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 of Quito, Ecuador (Data Structures), and Western Carolina University (Software Engineering).

The assessments in this dissertation are based on the data collected at Clemson University and University of Alabama. The required IRB procedures were not in place in time for data from other colleges to be used. However, the reports of the instructors teaching these courses at the aforementioned institutions indicate that students had a positive experience learning the reasoning topics.

At Clemson, CPSC215 is a sophomore-level Software Development Foundations course, and the data analysis includes pilot offerings in Spring 2008 and 2009, and

offerings since institutionalization of the reasoning principles in Spring 2011, Fall 2011, Spring 2012 and Fall 2012 for a total of 6 semesters (8 sections). A portion of the RCI reasoning principles have been integrated with the traditional software engineering topics that are normally covered in this course, and students spend approximately 3-4 weeks learning these principles. The follow-on course at Clemson - CPSC372 is a junior-level undergraduate Introduction to Software Engineering course. Data analysis covers the offerings since institutionalization of the reasoning principles - Fall 2010, Spring 2011 Fall 2011, and Spring 2012, for a total of 4 semesters (4 sections). About a third of this course is devoted to the RCI reasoning principles, and the rest is devoted to traditional SE topics. In Fall 2012, only a limited set of RCI reasoning principles was taught, because three classes in November (when reasoning principles are typically taught) were cancelled due to weather-related and other circumstances. The analysis does not include that offering.

CS315 taught at University of Alabama is a junior-level software engineering course where students spend 3-4 lectures covering a subset of RCI reasoning principles. CS315, including reasoning principles, was offered in Spring 2011, Fall 2011, and Spring 2012.

Table 5 summarizes the collected data by year/semester. The number of sections that were available for data collection is indicated in parenthesis, along with the type of data that was collected. The data consists of student midterm and final examinations, and select assignments and quizzes that incorporated the RCI reasoning principles. Please note that the data of interest was gathered from specific questions based on specific LOs,

which in turn were based on specific RCI reasoning principles, not the entire test or quiz scores.

	Year	Spring	Fall
Pilots	2007		CPSC372 (1) pilot assignment
	2008	CPSC215 (1) pilot exam	
	2009	CPSC215 (1) pilot exam	
After Institutionalization	2010		CPSC372 (1) final exam
	2011	CPSC215 (1) final exam CPSC372 (1) final exam CS315 (1) assignment	CPSC215 (2) final exam CPSC372 (1) final exam, 4 quizzes CS315(1) final exam, assignment
	2012	CPSC215 (2) final exam, 2 quizzes CPSC372 (1) final and midterm CS315 (1) final exam	CPSC215 (2) final exam CPSC372 (1) final exam

Table 7. Student data collection by year/semester, number of sections and data type

Since only the results pertaining to the RCI reasoning principles are of interest in this experiment, but not the overall performance of individual students, the data has been anonymized. Each question is categorized according to the RCI principle it addresses, and rated in terms of the difficulty level using the 3- level version of Bloom's taxonomy (KC, AA, SE). Examples of test questions at different difficulty levels are provided to illustrate the type of questions students were given during the assessment and to serve as the foundation to our conclusions and observation. The observations based on the experimental data are discussed in a further section.

4.1. Assessment of RCI Reasoning Principles in CPSC215

4.1.1. CPSC215 Pilot Experiments in Spring 2008 and 2009

A small set of the reasoning principles was introduced in CPSC215 in Spring 2008 and Spring 2009. The goal of the experiment was to determine if students can successfully master reasoning principles.

In Spring 2008 only one reasoning principle was introduced in this course, specifically, operation pre- and post-conditions. At this point the Reasoning Concept Inventory was still in its infancy stage, but some of the principles were already identified as future important elements. This principle was later categorized as principle RCI #3.4.3 (pre- and post-conditions), with RCI #3.4.3.2 (responsibility of the caller) and #3.4.3.3 (responsibility of the implementer).

Three final exam questions addressed this reasoning principle. The assessment data is presented in Table 8. The first question at the KC level asked students to define the terms pre-condition and a post-condition, with the class average of 100%. Questions two (RCI #3.4.3.2) and three (RCI #3.4.3.2) were in fact parts of the same larger question, asking students to use the notation introduced in class to provide a pre- and post-condition for an operation. These questions were at the AA level of difficulty. The class average on RCI #3.4.3.2 was 72% with 62% of students scoring at or higher than 70%. The class average on RCI #3.4.3.3 was 64% with 54% of students scoring at or higher than 70%. The assessment was only taken by 13 non-exempt students who were not exempt from the final examination. The rest of the students were exempt from the final because they had an ~~–A~~" going into the final exam.

Even though only one reasoning principle was introduced in Spring 2008, it was discovered that students can learn reasoning principles, and do well.

	Difficulty Level	Class Avg	# students >= 70%
RCI #3.4.3 (2, 3)	KC	100%	100%
RCI# 3.4.3.2	AA	72%	62%
RCI# 3.4.3.3	AA	64%	54%

Table 8. CS215 Spring 2008 pilot assessment data for 13 non-exempt students

More reasoning principles were introduced and assessed in Spring 2009, and the data is presented in Table 9. The same three questions assessing RCI #3.4.3 (2, 3) (pre- and post-conditions) were used again, with students scoring at 96%, 59% and 57% correspondingly. The assessment was taken by twelve non-exempt students, with about one third of the class being exempt from the final with an "A".

This time new RCI principles were introduced. RCI #4.1.1.3 deals with formal verification. The assessment question at the KC level asked students about the goals of formal verification. The class average was 75% with 75% of students scoring at or above 70%. RCI#4.2.1.1 (specifications as external contracts) was also assessed at the KC level, with the average of 83%, and 83% of students scoring at or above 70%. Reasoning principle RCI #5.2.2 was assessed using two different questions, both dealing with the construction of a reasoning table, at two different levels of difficulty, AA and SE. Considering the construction of reasoning tables is one of the most difficult topics, students did very well, with class averages of 92% and 84% respectively.

RCI #5.3.2 (application of proof techniques on VCs) was assessed at the SE level, with class average of 10% with only 8% of students scoring in the acceptable range at or higher than 70%. Even though this is a very difficult new topic, we would like to have a higher average.

The formative assessment results from Spring 2009 indicate that students are capable of learning reasoning principles, and do well on most of the topics.

	Difficulty Level	Class Avg	# students >= 70%
RCI# 3.4.3 (2, 3)	KC	96%	92%
RCI# 3.4.3.2	AA	59%	42%
RCI# 3.4.3.3	AA	57%	42%
RCI# 4.1.1.3	KC	75%	75%
RCI# 4.2.1.1	KC	83%	83%
RCI#5.2.2.1	AA	92%	92%
RCI#5.2.2 (1, 2)	SE	84%	83%
RCI#5.3.2	SE	10%	8%

Table 9. CPSC215 Spring 2009 pilot assessment data for12 non-exempt students

4.1.2. Assessment of RCI Principles in CPSC215 after Institutionalization

In Spring 2011, RCI reasoning principles were institutionalized in CPSC215 (Software Development Foundations) at Clemson. This course introduces some of the most important basic concepts, techniques, and tools associated with development and reasoning about software with objects. Because this is a sophomore-level software engineering course, most of the expected learning outcomes are on the KC and AA level of mastery. Note, that the analysis was focused on the lower-performing (non-exempt)

students. At the end of each semester top performers in this course (usually one third of the class) are exempt from taking final examination, because they have an "A" going into the final exam. The rest of the students are given summative assessments that test specific skills required to think abstractly, model data structures with a variety of models, and understand and develop mathematical specifications.

This section shows how a number of RCI principles are assessed using a variety of actual test questions in CPSC215 across several semesters, specifically, Spring 2011, Fall 2011, Spring 2012, and Fall 2012. The assessment questions were constructed either by the course instructor, or by the researchers conducting the experiment. To provide a feel for different kinds of test questions, we give an example each from the precise specifications area (RCI #3), modular reasoning area (RCI #4), and correctness proofs area (RCI #5). For variety, we have picked a KC level question, an AA level question, and an SE level question.

Example #1: Assessing RCI #3.4.3 at the AA Level

Exam questions have been used to test both of the items RCI#3.4.3.1 (responsibility of the caller) and RCI#3.4.3.2 (responsibility of the implementer). They are written on the KC level, where students are expected to explain the meaning of a specific operation's pre- and post-condition. The question in Figure 13 tests RCI#3.4.3 at the AA level.

LO: Provide formal pre-condition and post-condition for an operation. (AA)

Question: Consider the following informal specification:

```
public void Foo (Sequence s1, Sequence sub, Sequence s2, int p);
```

- Pre-condition:
  - *sub* must be a substring of *s1*.
  - *p* must be less than or equal to the length of *s2*.
- Post-condition:
  - *sub* will be removed from *s1*.
  - No other changes will be made to *s1*.
  - *sub* will not be modified.
  - *sub* will be inserted at position *p* within *s2*.
  - No other changes will be made to *s2*.
  - *p* will not be modified.
  - Examples:
    - PRE: #s1 = <1, 2, 3, 4, 5>, #sub = <3, 4>, #s2 = <1, 2, 3>, #p = 1.
    - POST: s1 = <1, 2, 5>, #sub = <3, 4>, s2 = <1, 3, 4, 2, 3>, #p = 1.

1. Provide the formal pre-condition for this method using the notation presented in class.
2. Provide the formal post-condition for this method using the notation presented in class.

Figure 13. Assessing RCI#3.4.3. on AA level in CPSC215 (example #1)

Example #2: Assessing RCI 4.1.1.3 at the KC Level

This RCI item deals with formal verification at the KC level. Though the first time around (Spring 2008) the topic of formal verification was not introduced, it was subsequently included in Spring 2009. The topic has been institutionalized and taught every semester since then. Undergraduate students need to understand that verified code is more reliable, causes less software failures, and can significantly reduce maintenance

later. At this an early point in their computer science experience, they are being exposed to what formal verification is and why it is important.

LO: State the goal of formal verification. (KC)

Question: Circle the phrase that best completes the sentence. The goal of *formal verification* is:

(a) prove that a piece of software works on all valid inputs;

(b) reveal the presence of software bugs;

(c) show that a piece of software is syntactically correct;

(d) improve code efficiency.

Figure 14. Assessing RCI#4.1.1.3 on KC level in CPSC215 (example #2)

Example #3: Assessing RCI#5.2.2 at the SE Level

RCI#5.2.2 (connection between specification and what is to be proved) deals with the construction of reasoning tables. To develop the table correctly, students need to master both RCI#5.2.2.1 (assumptions) and RCI#5.2.2.1 (obligations), along with other prerequisite knowledge, such as operation specifications and some relevant modular reasoning principles from RCI#4. This question, shown in Figure 15 requires combining a number of prerequisite skills. Construction of reasoning tables is one of the more difficult reasoning principles introduced in the sophomore-level course. This learning outcome is on the SE level, the highest level of the modified Bloom's taxonomy, because it requires the student to determine and write down the assumptions and obligations for each state in a piece of software that is to be proved.

LO: Construct a reasoning table with assumptions and obligation when given an operation implementation and specifications. (SE)

Question: Examine specification and implementation of the operation *mystery( )*.
Complete the reasoning table below by providing assumptions and obligations.

```
public static void mystery(Queue q)
  // pre-condition: (|q| != 0)
  // post-condition: there exists x: object, s: string of object
  // such that #q = s o <x> and q = <x> o s
{
   //S0
      Object x;
   //S1
      x = q.dequeue();
   //S2
      q.enqueue(x);
   //S3
      q.length()
   //S4
}
```

State #	Assumptions	Obligations
S0		
S1		
S2		
S3		
S4		

Figure 15. Assessing RCI#5.2.2 on SE level in CPSC215 (example #3)

Being a non-trivial topic new to many instructors, this example requires an elaboration. In this reasoning table, students write verification conditions (VCs), which are the *assumptions* and *obligations* for each state of the operation. In state 0, *mystery*'s pre-condition may be assumed; in state 4, *Mystery*'s post-condition must be proven (confirmed). If the code entails calling another operation, the pre-condition for the other operation gets entered in the "Obligations" 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 –Assumptions" column. So for operation *mystery* in Figure 15 the student must generate five assumptions and five obligations.

4.2. Assessment of RCI Reasoning Principles in CPSC372

4.2.1. CP SC372 Pilot for Demonstrating the Roles of External and Internal Contracts in Software Engineering

The initial pilot experiment in CPSC372 took place in Fall 2007. It was conducted to find out if undergraduate students can successfully learn the RCI reasoning principles. The assessment here involves an analysis of students' project assignments. The assignment was intended to demonstrate to students the importance of mathematical modeling and formal specifications, and specifically the roles of external and internal contracts of software components. The general learning goal in this project was to teach students to develop software components according to external and internal contracts.

Students received a moderately complex assignment in terms of the specification, similar to the sample software engineering assignment discussed in the previous subsection 3.4.2.6, –Team Project Assignments," and worked in small teams. The goal of the project was to demonstrate that modules developed by different developers can be seamlessly assembled in a software system and operate flawlessly, provided that they strictly adhere to the formal specifications and no internal or external contract violations have occurred. Relationships among modules used in such an undergraduate class assignment were illustrated in a previous section in Figure 12. The diagram reflects the

complexity of the relationships between components. The central component, *Preemptable_Queue_Template* is implemented by three realizations and has two enhancements. One of the realizations, *Stack_Based_Realization* uses *Stack_Template*, which has its own *Array_Realization*. Each of the two enhancements is implemented by one or more realizations. Finally, *PQ_User Facility* is the main program that uses the *Preemptable_Queue_Template*.

To check if students understood contract specifications and avoided external contract violations, components developed by different developers were assembled in a software system, compiled, executed, and the results were examined. For example, in the first test run, the *Simple_Realization* for the *Preemptable_Queue_Template* developed by group #1 was used with the *Insertion_Capability* enhancement designed by group #2, *Iterative_Shuffle_Realization* coded by group #3, and the *PQ_User* facility written by group #4. Another test run was based on the *Clean_Circular_Array_Realization* from group #4, *Insertion_Capability* enhancement from group #1, *Recursive_Shuffle_ Realization* from group #2, and the *PQ_User* facility from group #3. A total of possible 196 such component combinations were tested. While testing for external contract violations, it was determined that 169 out of 196 combinations were successful, yielding the rate of success of approximately 86%. In other words, 86% of the random combination of components developed by different students worked correctly, giving a measure of confidence that students indeed developed their individual components according to specified external contracts. External contract violations in students'

projects often resulted from not implementing operations required by a concept (omitting or forgetting them), missing an operation parameter, or using wrong names.

To test for internal contract violations, the code for various primary operations within identical components produced by different developers was extracted from their modules, and then was combined into a new module, compiled, and executed. For example, the concept *Preemptable_Queue_Template* has several operations: *Enqueue(), Dequeue(), Inject(), Swap_Last_Entry(), Length(),Capacity(), and Clear()*. For example, to test if any internal contract violations occurred in the *Simple_Realization*, an implementation of this concept, the following operations were combined in one module, compiled, and ran: operation *Enqueue()* from group #1, *Dequeue()* from group #2, *Inject()* from group#3, and *Swap_Last_Entry()* from group #4, and so on. The number of such random combinations was 164. Internal contract violations gave rise to incorrect implementation of the basic operations, with 156 out of 164 combinations being a success, producing success rate of approximately 95%. Table 13 shows how many of these combinations successfully compiled, ran, and produced correct results.

	# of Combinations	# Success	Success Rate
External Contracts	196	169	86%
Internal Contracts	164	156	95.1%

Table 10. CPSC372 Fall 20007 pilot assessment data

These results need some elaboration. For example, suppose that we do not use internal contracts while using an array to represent a preemptable queue. One of the realizations students were asked to implement *Preemptable_Queue_Template*, named *Simple_Realization* uses an array to represent the preemptable queue. If the queue is represented using an array, there are at least two obvious correspondences and two obvious conventions. The ideas are discussed in detail in Appendix C. Given the 4 combinations, the probability that an arbitrary implementation is correct (by random accident) is 25%. Representing a queue using an array is a relatively simple task, compared to implementing more complex components, such as Lists, Trees, and Maps. These can be represented by other complex components, and without formal internal contract specifications the task is challenging at the least, because there is a larger number of ways of implementing these more complex components. If members of the development team implementing different operations for a component are not provided with the internal contracts, then it is highly unlikely that they could separately build their own parts that would ultimately work together seamlessly. The results of this experiment indicate that in the classroom setting, it is indeed possible to achieve high success rate at integration time, only if students understand the meaning of internal and external contracts.

The results of this pilot experiment indicate that students have met the general learning goal of learning to develop software components according to specifications. They are able to learn formal contracts and they understand the importance of software quality. However, since this learning goal lacks the precise description of what skills

students are expected to master, and at what level, the only conclusion from this assessment that could be made with confidence is that students have learned most of the necessary skills and produced satisfactory results. Without precise learning outcomes, it is difficult to pinpoint exactly which specific reasoning skills students have mastered and which they have not. So, while, this pilot does confirm that students are capable of learning reasoning principles, it also highlights where the RCI and learning outcomes can make contribution. The RCI principles with associated learning outcomes have been introduced with further experimentation, and they have assisted in pinpointing what students are learning and what they are not.

4.2.2. Assessment of the RCI Principles in CPSC372 after Institutionalization

Since the pilot experiment, various combinations of the RCI reasoning principles have been taught in CPSC215, CPSC372, and CS315 using collaborative classroom exercises and other reasoning tools. A detailed discussion is available in [25, 36, 121]. The teaching of the RCI reasoning principles has been driven by learning outcomes. As we have shown in earlier subsections, every area of RCI has an associated learning outcome, along with different levels of difficulty. The following subsections show a sample of the RCI reasoning principles with the level of difficulty appropriate to each topic and course, and how learning outcomes are used to assess student learning. Though CPSC215 is taught using Java, the RESOLVE integrated environment has been used as the teaching medium for CPSC372 course. Appendix C offers a detailed discussion of

RESOLVE and provides the background information necessary to understand the technical details of the following subsections.

A larger number of RCI principles have been introduced in the junior-level software engineering course CPSC372. Unlike CPSC215, in this course learning outcomes were generally at the higher levels of Bloom's taxonomy. This course goes beyond the basic concepts, techniques, and tools associated with the development of predictable software. While these skills are an important focal point, a considerable amount of time is spent examining methods that teach students to reason rigorously about the software they develop and maintain. By the end of the course, students are expected to be able to develop high-quality software and be able to reason about its behavior.

The collected evidence proves that undergraduate students are able to understand mathematical abstraction, as exhibited by their ability to read, write, and use specifications based on these abstractions.

Below is a sample of actual assessment questions from the final examinations. Because of the large number of assessment questions used, only a sample is discussed below.

Example #1: Assessment of RCI #3.4.3 at the SE Level

The question in Figure 16 tests RCI#3.4.3 and includes both items RCI#3.4.3.2 (responsibility of the caller) and 3.4.3.3 (responsibility of the implementer). Because formal specification is a recurring topic, this area is frequently tested throughout the

course with a variety of questions that cover different difficulty levels. The learning outcome of this particular example is on the SE level.

LO: Determine an appropriate post-condition for the given code. (SE)

Question: Write an *ensures* clause to capture the behavior of code precisely.

```
Operation Mystery_3 (updates S: Sequence);
      requires 1 <= |S|
      ensures    ?????
   Procedure
      Var E: Entry;
      Remove_After (0, E, S);
      Insert_After (Length(S), E, S);
end Mystery_3;
```

Figure 16. Assessing RCI#3.4.3 on SE level in CPSC372 (example #1)

Example #2: Assessment of RCI #4.3.1 at the SE level

The assessment question in Figure 17 tests both RCI#4.3.1.1 (abstraction functions/relations and correspondence) and RCI#4.3.1.2. (representation invariants/ conventions). This question corresponds to a learning outcome at the SE level.

Example #3: Assessment of RCI #5.2.2 at the SE level

The test question in from Spring 2012 final examination deals with construction of verification conditions, and includes both RCI#5.2.2.1 (assumptions) and RCI#5.2.2.2 (obligations). Students are expected to provide assumptions and obligations for several states in a reasoning table. This learning outcome is on the SE level.

LO: Generate code for a template without violating conventions and correspondence assertions.

Question: Complete the following implementation of Preemptable_Queue_Template, without violating the conventions or correspondence assertions. Notice that the Stack facility has been enhanced by the flipping capability; your code needs to make appropriate use of this enhancement for full credit.

```
Realization Stack_Based_Realiz for Preemptable_Queue_Template;
     uses Stack_Template;
  Facility Entry_Stack_Fac is Stack_Template (Entry, Max_Length)
     realized by Array_Realiz
     enhanced by Flipping_Capability
        realized by Iterative_Realiz;
    Type P_Queue = Record
       Contents: Entry_Stack_Fac.Stack;
  end;
     convention true;
     correspondence Conc.Q = Q.Contents;

  Procedure Enqueue (alters E: Entry; updates Q: P_Queue);
       .....
  end Enqueue;

  Procedure Inject (alters E: Entry; updates Q: P_Queue);
       .....
  end Inject;

  Procedure Dequeue (replaces R: Entry; updates Q: P_Queue);
       .....
  end Dequeue;

  Procedure Swap_First_Entry (updates E: Entry; updates Q: P_Queue);
       .....
  end Swap_First_Entry;
end Stack_Based_Realiz;
```

Figure 17. Assessing RCI#4.3.1 on the SE level in CPSC372 (example #2)

Overall, the assessment data collected in the course of the experiment led us to a number of important observations, presented in a section 4.4. The data, based on specific learning outcomes is used to create a feedback loop for continuous instructional improvement, continues to be collected. Furthermore, recall from Chapter 2 that the work

on the RCI has involved over 25 educators and researchers. These collaborators are at various institutions, many of whom are already beginning to integrate some of the RCI items more fully into their curriculum.

LO: Provide assumptions and obligations for the given code with its specifications. (SE)

Question: Refer to the following specification and code. Write the assumptions in state 1 and 6 and obligations in states 0 and 5.

```
Enhancement Insert_Front_Capability for Queue_Template;
  Operation Insert_Front(updates Q: Queue; alters E:Entry);
    requires |Q| < Max_Length;
    ensures  Q = <#E> o #Q;
end Insert_First Capability;

Realization Insert_Front_Realiz for Insert_Front_Capability
                                        of Queue_Template;
  Procedure Insert_Front (updates Q: Queue, alters E: Entry);
    Var T: Queue;
  0
    Enqueue (E, Q);
  1
    While (Length (Q) >0)
       changing Q, T, E;
       maintaining T o Q = <#E> o #Q;
       decreasing |Q|;
    do
       Dequeue (E, Q);
       Enqueue (E, T);
    end;
  5
    Q:=: T;
  6
  end Insert_Front;
end Insert_Front_Realiz;
```

Figure 18. Assessing RCI#5.2.2 at the SE level in CPSC372 (example #3)

4.3. Assessment of RCI Reasoning Principles in CS315

4.3.1. Assessment of the RCI Principles in CS315 Regular Course Offerings

The University of Alabama is one of the first universities to follow Clemson in incorporating reasoning skills into the software engineering curriculum. Having collaborated with them for three semesters, the assessment data from Spring 2011, Fall 2011, and Spring 2012 was collected. This course incorporates five RCI reasoning principles. The next subsections show the assessment questions from this course, and discuss the results.

Example #1: Assessing RCI# 3.4.3 (2, 3) at the KC level

This final exam question in Figure 19 assesses RCI#3.4.3.2 (responsibility of the caller) and RCI# 3.4.3.3 (responsibility of the implementer). The same question is used during three consecutive semesters, and has the learning outcome at the KC level.

LO: Explain the responsibility of the caller and responsibility of the implementer. (KC)

Question: Indicate all the correct answers:

____ (a) pre-condition is the responsibility of the caller

____ (b) post-condition is the responsibility of the implementer

____ (c) pre-condition is the responsibility of the caller

____ (d) post-condition is the responsibility of the implementer

Figure 19. Assessing RCI#3.4.3.2 in CS315 on KC level (example #1)

Example #2: Assessing RCI# 5.3 in CS315 at the SE level

The test question in Figure 20 assesses a students' mastery of RCI#5.3 (proving VCs) and includes a prerequisite skill of RCI#5.2.2 (construction of verification conditions). Here students build a reasoning table for a given piece of code. In addition to that, they generate and prove verification conditions. This question was also used every semester with the learning outcome at the SE level.

LO: Generate and prove verification conditions for the given code. (SE)

Question: Fill out reasoning table for the following code. Generate VCs and prove them.

```
Operation Clear_2 (updates S: Stack)
     requires |S| = 2;
     ensures S = empty_string;
   Procedure Clear_2 (updates S: Stack)
      Var Next_Entry: Entry;
      Pop(Next_Entry, S);
      Pop(Next_Entry, S);
   end Clear_2;
```

Figure 20. Assessing RCI#5.3 in CS315 on SE level (example #2)

4.4. Important Observations from the Data Sets

The data analysis provides evidence that students are capable of learning reasoning concepts, of doing well on these topics, and are having positive attitudes to the new material. A number of observations made during the data analyses are discussed next. The observations are grouped into eight sections according to their relevance, and each is followed by a brief discussion. Below is the listing of all the observations. Before proceeding to that section, it needs to be emphasized that the RCI and the learning

outcomes are paramount to making conclusions about experimental data. Because the RCI reasoning principles are divided into five areas, each of which is further subdivided into several levels, and learning outcomes on the appropriate level of difficulty, learning of each RCI reasoning principle can be assessed with a high degree of precision. Being able to exactly pinpoint the deficiencies in particular areas of student learning guides the development of an effective intervention for the area in need. Table 11 provides a quick overview of the observations discussed in detail in subsections 4.4.1 through 4.4.8.

Observations by Relevance
4.4.1. Observations related to students' learning of the RCI reasoning principles
4.4.2. Observations related to student performance on the RCI reasoning principles by course
4.4.3. Understanding variations between multiple offerings of a course
4.4.4. Observations related to conducting interventions
4.4.5. Observations related to instructors teaching reasoning principles
4.4.6. Observations related to incorporating the RCI principles into a course
4.4.7. Observations related to student attitudinal assessments
4.4.8. Observations supported by indirect evidence

Table 11. An overview of the observations

4.4.1. Observations Related to Students' Learning of the RCI reasoning principles

RCI can guide teaching and assessment of principles of specification, modular reasoning, and correctness proofs at increasing levels of sophistication along Bloom's taxonomy across the curriculum. The following observations are discussed in this section:

> *Observation #1.1:* In undergraduate software development foundations course CPSC 215 students are capable of learning RCI reasoning principles.
>
> *Observation #1.2:* In the undergraduate software engineering course CPSC 372 students are capable of learning the RCI reasoning principles taught at advanced levels of difficulty.

Observation #1.1: In undergraduate software development foundations course CPSC 215 students are capable of learning RCI reasoning principles.

Why is it important?

The goal is that the students learn the RCI reasoning principles so that they are able to reason mathematically about software correctness. A number of reasoning principles are taught in the undergraduate sophomore-level software development foundations course. The assessment data indicates that students are capable of learning the reasoning principles taught at different levels of difficulty. Because teaching of these

RCI reasoning principles is driven by the learning outcomes based on the RCI, it is possible to pinpoint exactly where an intervention is needed.

Details:

Table 12 presents the assessment data from the three semesters of CPSC215 (Software Development Foundations) at Clemson: Spring 2009, Spring 2011, and Fall 2012. These RCI reasoning principles have learning outcomes at the KC and AA levels of difficulty. About one third of the students are usually exempt from the final examination because they already have an ―A" going into the final. This data reflects the performance of the non-exempt students.

The chart in Figure 21 illustrates the assessment data across the three semesters. 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) and 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). While such small fluctuations in the students' performance across semesters is expected, a dramatic increase was observed in RCI#5.3.2 (application of proof techniques on VCs, level AA), which increased from 10% to 34%, and, finally, to 55%. This increase is the result of the intervention that consisted of using a series of educational videos described in a previous chapter.

Considering that the data is based on performance of the non-exempt students, achieving a 55% average on this reasoning principle, one of the most difficult principles with the LO on the AA level, is acceptable. Additional interventions are being considered

in order to increase the class average for future sections. RCI# 3.4.3(2, 3) (pre and post-conditions, level AA) has decreased from 60% to 51% and required an intervention as well. Overall, the data shows that students are capable of learning RCI reasoning principles in the undergraduate software development foundations course at different difficulty levels.

Evidence:

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%

Table 12. CPSC215 class averages in Spring 2009, Spring 2011, and Fall 2012 (observation #1.1)

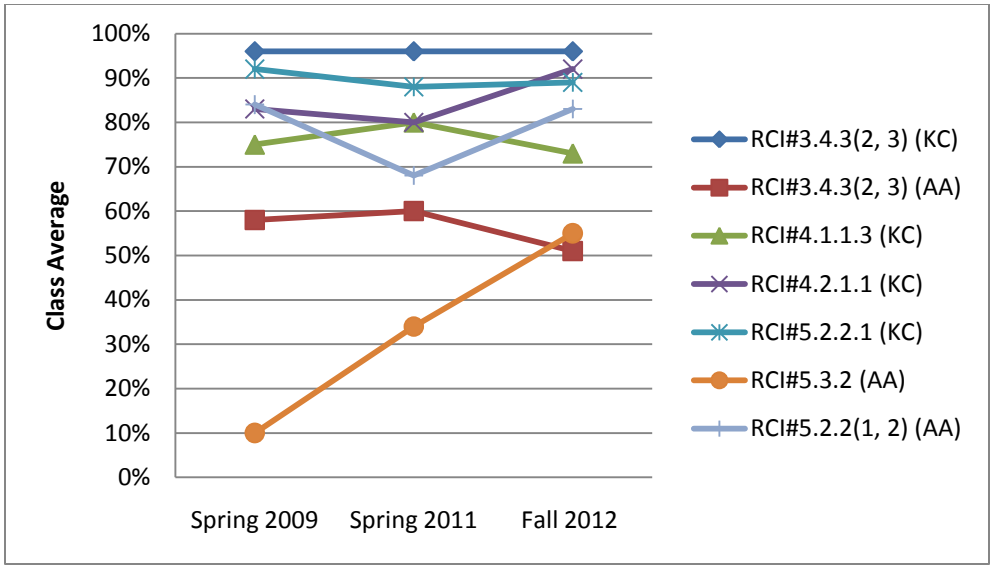


Figure 21. Graphic representation of CPSC215 data from Spring 2009, Spring 2011, and Fall 2012 (observation #1.1)

Observation #1.2: In the undergraduate software engineering course CPSC 372 students are capable of learning the RCI reasoning principles taught at advanced levels of difficulty.

Why is it important?

The RCI reasoning principles are to be taught across the computer science curriculum. Observation #1.1 points out that undergraduate students are capable of learning RCI reasoning principles at lower levels of difficulty. The data collected in the junior-level software engineering course CPSC372 shows that these students are capable of learning RCI reasoning principles on more advanced levels of difficulty.

Details:

Table 13 presents student assessment data from the junior-level software engineering course CPSC372. The data was collected from four semesters: Fall 2010, Spring 2011, Fall 2011, and Spring 2012. Four RCI reasoning principles were assessed on the AA and SE levels of difficulty. The chart in Figure 22 illustrates the assessment data, and indicates that 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). Again, a small fluctuation in the students' performance across semesters is to be expected.

The reasoning principle RCI#5.3(1, 2) (VCs as mathematical implications, and application of proof techniques on VCs, level SE) is one of the most advanced skills taught in this course. Though the format of the assessment questions remains the same each semester, in Fall 2011 the instructor introduced additional complexity into the question. After analysis, it was determined that mathematical results concerning a string operator named Prt_Btwn, that retrieves a substring stored between two indices was used in the VCs, caused particular difficulties for the students in proofs. The class average dropped to 46%. Based on this analysis, the instructor realized that his presentation of the idea was lacking and devised an alternative explanation of the operator; additionally, the online tutorial description of this concept was improved as well. The resulting class average increased to 86%. Overall, the assessment data indicates that students in the

junior-level computer science course are capable of learning advanced reasoning concepts.

Evidence:

Reasoning Topic	Difficulty Level	Fall 2010	Spring 2011	Fall 2011	Spring 2012
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%

Table 13. CPSC372 student performance throughout several semesters (observation #1.2)

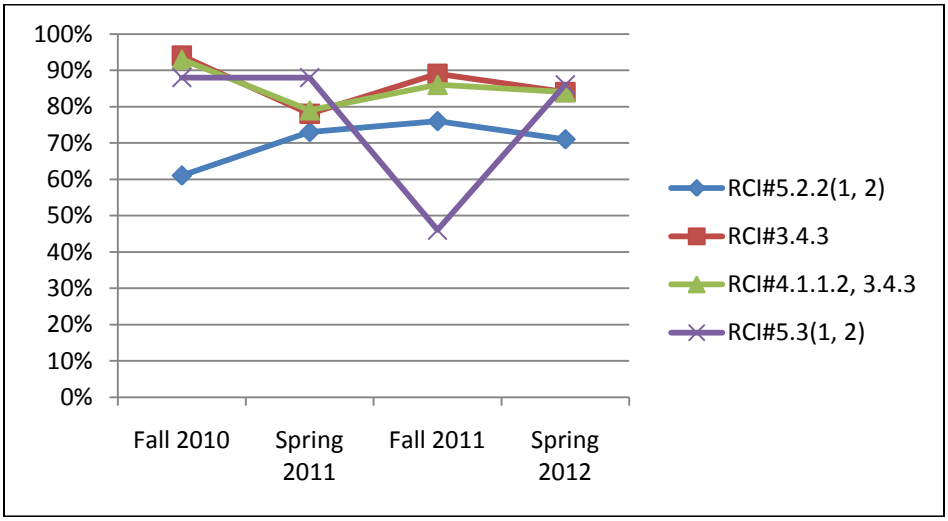


Figure 22. Graphical Representation of CPSC372 data from Fall 2010, Spring 2011, Fall 2011 and Fall 2012 (observation #1.2)

4.4.2. Observations Related to Student Performance on the RCI Reasoning Principles by Course

Within a single course, the RCI provides a comprehensive picture of the reasoning principles that are taught and assessed. The data shown here is collected during the last semester that the course included reasoning principles: CPSC 215 in Fall 2012, CPSC372 in Spring 2012, and CS315 in Spring 2012. The three observations below summarize student success in learning a set of skills from precise specifications, modular thinking, and correctness proofs knowledge areas. The following observations are discussed in this section:

Observation #2.1: CPSC 215 Experimental data from Fall 2012 semester

Observation #2.2: CPSC372 Experimental data from Spring 2012 semester

Observation #2.3: CS315 Experimental data from Spring 2012 semester

Observation #2.1: CPSC 215 Experimental Data from Fall 2012 Semester

Experimental data from Fall 2012 semester from one of the sections of CPSC215 is shown in Table 14. The leftmost column indicates the reasoning principle assessed, followed by its difficulty level. The "Class Avg" column in the table indicates the class average for the particular RCI item tested, and the column labeled "Percent of Students with a Score ≥ 70%" indicates the percentage of the students in the class that earned at least 70% of the points for the RCI item tested.

In Table 14, the student average of all, but the first two RCI principles is in the acceptable range. These two principles RCI#3.4.3 deal with precise specifications, and pre- and post-conditions in particular. Their averages are 44% and 48% respectively and fall below the cut-off of 70%, with only 19% and 38% of students scoring 70% or above. The two principles are taught at the AA level of difficulty. The same principle RCI#3.4.3 was assessed using a question at the KC level, and the student average indicates that 100% of students answered it correctly. Because understanding specifications are prerequisite to learning modular reasoning and correctness proof skills, it was concluded that the particular questions used for assessing the principles might not have been of appropriate difficulty; this topic is discussed further later.

Reasoning Principle	Difficulty Level	Class Avg	% of students >=70%
RCI# 3.4.3.2	AA	44%	19%
RCI# 3.4.3 (2, 3)	AA	48%	38%
RCI# 3.4.3 (2, 3)	KC	100%	100%
RCI# 4.1.1.3	KC	71%	71%
RCI# 4.2.1.1	AA	95%	95%
RCI# 5.2.2.1	AA	81%	81%
RCI# 5.2.2 (1, 2)	SE	81%	81%
RCI# 5.3.2	SE	69%	57%

Table 14. CPSC215 Fall 2012 final exam, 21 students, instructor 1 (observation #2.1)

Observation #2.2: CPSC372 Experimental Data from Spring 2012 Semester

The assessment data in Table 15 and Table 16 are based on the midterm and final

examinations of Spring 2012 semester. Thought we did assess a very small number of the RCI reasoning principles in Fall 2012, the amount of data is not sufficient to make general conclusions about how students are learning the reasoning principles. Reasoning principles could not be covered adequately, due to cancellation of three classes late in the semester when the principles are usually taught.

Reasoning Principle	Difficulty Level	Class Average	% of students >=70%
RCI#3.4 (1, 2)	AA	59%	33%
RCI#3.4.3 (2, 3)	KC	79%	79%
RCI#3.4.3	AA	70%	58%
RCI#3.4.3.3	AA	67%	46%
RCI#4.1.2	KC	75%	75%
RCI#4.2.1	AA	67%	42%
RCI#4.2.3	KC	73%	51%
RCI#4.2.3.2.5	SE	78%	63%
RCI#4.3.1	KC	71%	71%
RCI#4.3.1 (1, 2)	SE	65%	55%

Table 15. Spring 2012 CPSC 372 midterm exam, Clemson University, 24 students (observation #2.2)

Reasoning Principle	Difficulty Level	Class Average	% of students >=70%
RCI#3.4.3	AA	84%	75%
RCI#5.1.1	SE	80%	50%
RCI#5.2.2	SE	71%	55%
RCI#5.3 (1, 2)	SE	86%	90%

Table 16. Spring 2012 CPSC372 final exam, Clemson University, 23 students (observation #2.2)

More RCI principles were assessed on the CPSC372 midterm, than on the final examination. Data in Table 16 indicates that class averages on the final examination were all in the acceptable range, while the four principles in the midterm data table are below the cut-off score of 70%. The principles RCI#3.4.3.3 (responsibility of the implementer, at the AA level of difficulty), RCI#4.2.1 (roles of clients and service providers, at the AA level), and RCI#4.3.1 (1, 2) (correspondence and conventions, at the SE level) were at 67%, 67%, and 65% correspondingly, and were close to the cut-off of 70%. The principle RCI#3.4 (2, 3) (operation signature, and pre- and post conditions) was at 59% with only 33% of students scoring above 70%. *Table 17* shows how student performance improved by the final examination, increasing to 84% with 75% of students scoring at or above 70%. This is likely because they do a significant reasoning project after the midterm, but before the final.

	Spring 2012 Midterm		Spring 2012 Final	
	Class Avg	%students >=70%	Class Avg	%students >=70%
RCI#3.4(2, 3)	59%	25%	84%	75%

Table 17. CPSC372 Improvement in student average from midterm to final examinations (observation #2.2)

Observation #2.3: CS315 Experimental Data from Spring 2012 Semester

The assessment data in this course was collected during the three semesters of Spring 2011, Fall 2011, and Spring 2012. The same test and assignment questions are

used every semester beginning in Spring 2011, and the class was taught by the same instructor. The test and assignment data are shown correspondingly in two tables below.

The assessment data from student assignments is shown in Table 18. The class average on RCI#5.2.2 (connection between specifications and what is to be proved) increased from 71% (with only 39% of students getting the score of 70% or above) in Spring 2011 to 94% in Fall 2011, with everyone in the class getting >=70%. On the other hand, the performance on RCI#5.3 (proof of VCs) decreased from 94% in Spring 2011 to 79% in Fall 2011. A possible recommendation for intervention to the instructor here is the use of reasoning videos to supplement classroom instruction.

	Difficulty Level	Spring 2011		Fall 2011	
		Class Avg	%students >= 70%	Class Avg	%students >= 70%
RCI# 5.2.2	SE	71%	39%	94%	100%
RCI# 5.3	SE	94%	94%	79%	79%

Table 18. Spring 2011 and Fall 2011 CS315 assignment data, University of Alabama (observation #2.3)

The final exam data in Table 19 shows that class average for RCI#3.4.3 (pre- and post-conditions) remained about the same, while decreasing on RCI#5.2.2 (connection between specifications and what is to be proved). The same RCI item was tested earlier in a homework assignment and the class average was 94%. This may be simply a reflection of students doing much between on a homework assignment than on the exam. Alternatively, it may imply that the project didn't reinforce the principles well or that the

exam question differed considerably from the homework assignment. In all cases, the feedback will be useful to the instructors.

	Fall 2011		Spring 2012	
	Class Avg	%students >= 70%	Class Avg	%students >= 70%
RCI# 3.4.3	82%	80%	82%	76%
RCI# 5.2.2	70%	67%	56%	46%

Table 19. Fall 2011 and Spring 2012 CS315 final exam data, University of Alabama (observation #2.3)

Overall, it was observed that all class averages were in the acceptable 70% or above range, except RCI #5.2.2 in Spring 2012 that was at 56%. This principle will require intervention the next time the reasoning principles are taught. We conclude that students at the University of Alabama learnt a subset of the RCI reasoning topics, some well and others not so well.

4.4.3. Understanding Variations between Multiple Offerings of a Course

The RCI can help understand the variations across the multiple offerings of a course, and help understand where the variations come from.

Observation #3.1: RCI helps determine the reasons that contribute to the variation across multiple offerings of the course.

Why is it important?

Some RCI questions have been repeated throughout several semesters, and the data has been used to develop a longitudinal analysis in order to determine the possible presence of an upward or downward trend. The trend helps visualize the changes in student performance across the multiple offerings of the course. Though a small variation from semester to semester is expected, if student performance increased or decreased dramatically, a timely intervention may be needed.

Details:

The data for the three areas of RCI is organized into a Table 20 shown in the evidence subsection below. If there were two instructors teaching different sections of the course, the data has been averaged across the two sections. Figure 23 allows us to study trends. It has already been observed that students have done well on RCI#3.4.3, (at the KC level) (the top line on the chart). It was noted that the change of instructors affected the graph in the downward manner in Fall 2011. During Spring 2008, 2009, and Spring 2011 the same instructor with many years of teaching experience taught the course, and the numbers were at 100%, 96%, and 96% correspondingly. In Fall 2011, two new instructors (last year PhD students) were assigned to teach the course. This was their first time teaching reasoning principles, and the average at that time trended down to 83%. An intervention that consisted in advising instructors on how to teach the topics more effectively was subsequently conducted, and in Spring 2012 the upward trend is apparent, leading us to believe that the intervention made a positive difference. In Fall 2012, the average decreased to 50%, and would require an intervention.

Student performance on RCI#4.1.1.3 (formal verification) at the KC level is stable across the four semesters. Though an almost imperceptible drop took place in Fall 2011, and then again in Fall 2012, the performance is still above 73%. A slight variation in student performance from one semester to another is not unusual.

This is not the case with RCI#5.2.2 (connection between specifications and what is to be proved). Reasoning tables have always been a more difficult subject to both students and instructors; it has learning outcome on a higher level of difficulty and requires that students have mastered a number of prerequisite reasoning principles. There was no known effective methodology for teaching this RCI reasoning principle when it was initially introduced. The average on this topic first decreased in Spring 2011, when an experienced instructor taught the course. To determine the possible reason for the decrease we used the questions found in Table 23 as a guide. Often, there is a combination of reasons why student performance decreases. The possible reason for the decrease in Spring 2011 was identified as the need of new instructional materials, since all the other variable remained the same. As a series of instructional videos (described in section 3.4.2.5) were being developed, a new set of instructors taught the course in Fall 2011, when the average decreased once again. Using Table 23, it was determined that the decrease could possibly be caused by the combination of variables: the new instructors, and the need for the new instructional materials. The instructional videos on building reasoning tables, generating verification conditions (VCs), and proving VCs (discussed in section 3.4.2.5) became available in Spring 2012. The instructors remained the same, but on introducing the same principle using the new teaching methods (videos watched bu

the students outside of class), student performance improved. The average once again increased in Fall 2012. We can conclude with confidence that the new videos have helped to improve performance.

Evidence:

	Difficulty Level	Spring 2011	Fall 2011	Spring 2012	Fall 2012
RCI# 3.4.3 (2, 3)	AA	96%	83%	85%	50%
RCI# 4.1.1.3	KC	80%	77%	81%	73%
RCI#5.2.2 (1, 2)	SE	68%	63%	70%	83%

Table 20. CPSC 215 RCI assessment questions that repeat every semester

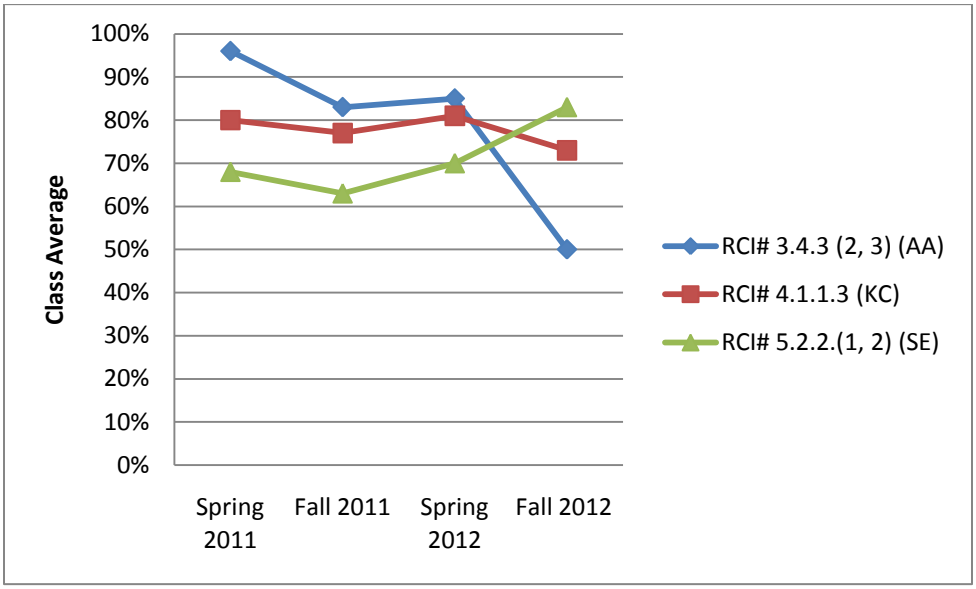


Figure 23. Visualizing data for the repeating RCI questions in CPSC215

4.4.4. Observations Related to Conducting Intervention

The RCI can help pinpoint learning difficulties and suggest places for intervention. A number of observations are discussed in this section:

Observation #4.1: Learning outcomes based on the RCI aid in pinpointing specific areas where students are having difficulties.

Observation #4.2: RCI can pinpoint suitable intervention to improve student learning.

Observation #4.3: Assessment questions should have the appropriate difficulty level.

Observation #4.4: It may take more than one intervention to ask the right questions.

Observation #4.5: Online tutorials serve as a bridge between two courses in the software engineering sequence: CPSC215 and CPSC372.

Observation #4.1: Learning outcomes based on the RCI aid in pinpointing specific areas where students are having difficulties.

Why this is important?

If students are performing well, instructors know they have succeeded. But if the assessment numbers are lower than expected, it would be desirable to know exactly which areas need improvement. RCI can help pinpoint exactly where the problem is. Because the five areas of RCI further expand into three more levels with the concept

details at the last level, and because each concept detail has a corresponding learning outcome at the appropriate difficulty level, it is possible to construct exercises to assess each specific skill. This task would be difficult, if not impossible without this detailed inventory and learning outcomes.

Details:

In Fall 2011, CPSC215 students were given a final examination which contained a question about using mathematical models for conceptualizing objects. The class average on this question (shown in Table 21) was 29% with only 14% of students scoring at or higher than 70%. The question dealt with RCI#3.3.1 (mathematical modeling for conceptualizing objects), and inadvertently involved an idea beyond the knowledge of the students. It was changed to a multiple choice question and student performance immediately improved; essentially this reduced the question from being at SE level to AA level.

Evidence:

RCI	Fall 2011	
	Class Avg	%students with >=70%
RCI #3.3.1.5	29%	13%

Table 21. Evidence from CPSC215 Fall 2011 (observation #4.1)

Observation #4.2: RCI can pinpoint suitable intervention to improve student learning.

Why this is important?

Changing teaching methods or introducing new teaching tools can significantly improve results. Even experienced instructors experiment with a variety of teaching methods. At times, teaching methods that work well with one topic, do not work as well as expected while teaching another. Sometimes no new or special teaching tools are necessary to improve student learning, except spending more classroom time explaining a specific concept. It is not always easy to decide if intervention is necessary. In this situation the benefits of the RCI, through a cascading event, becomes obvious. Without gathering specific data showing direct evidence of student learning it would be difficult to decide if an intervention is necessary, as well as what type of an intervention is necessary.

Details:

An instructor familiar with RESOLVE, but new to teaching reasoning principles have used a new teaching tool as a way to improve student performance. Constructing a reasoning table is a new concept for the students, as well as for many instructors in computer science, and these instructors are still seeking effective methods for teaching it. The short educational videos (discussed in detail in a previous chapter) provide a step by step guide in the construction and proof of a reasoning table for a simple code example. In Fall 2011 and Spring 2012 students were given the same assessment question that calls for the construction of a reasoning table. After viewing the instructional video students' average improved from 64% to 79%, and with only half of the students scoring 70% or

higher in Fall 2011, to 71% of students in Spring 2012, as shown in *Table 22*.

Evidence:

RCI	Fall 2011		Spring 2012	
	Class Avg	% students with >=70%	Class Avg	% students with >=70%
RCI#4.1.1.2, RCI#5.2.2	64%	50%	79%	71%

Table 22. Evidence from CPSC215 Fall 2011 and Spring 2012 (observation #4.2)

Discussion:

Some level of variation in student average across course offerings is expected. Every time a new RCI reasoning principle is introduced, students are expected to do well. Certainly, each educator strives to have students perform at the ―A" or ―B" level, that correspond to ―Excellent" and ―Good" according to the Clemson University Grading system found at: http://www.registrar.clemson.edu/publicat/catalog/sections/aca_regs/. Because a ―C" (―Fair") or a better grade is expected of computer science students at Clemson, this level of performance (i.e., 70% on a 100% scale) has been used as a cut-off for satisfactory performance. If the percentage is noticeably lower, or if the numbers decrease each subsequent semester, intervention is developed and put into place. Without the RCI with learning outcomes it is not possible to assess student performance with precision, and without understanding the reasons of decreased performance it is difficult to devise an effective intervention.

We have categorized the possible reasons for student performance fluctuation into a table that was used as a guide to determine the cause of performance variation. Every

time student data is collected and analyzed, areas requiring intervention can be identified using the RCI. Not only does low performance requires intervention. Occasionally, intervention may be needed if all students score highly on a test question. This usually indicates that the question was too easy. Intervention may consist of changing test questions to match the difficulty level of the material. Analyzing the causes of low performance is more difficult, as many more variables are involved. Intervention may require instructor training, spending more classroom time on the topic, using additional exercises, or using additional instructional tools (TCRA, tutorials, web interface, videos, etc.). At this point, several iterations of interventions have been conducted, and are continuing to be conducted as new data becomes available.

Because of the use of the RCI principles, we have a better process in place and have more confidence in making conclusions about what the instructors are doing right while teaching students. When a teaching method is changed, or time covering a specific topic is increased, and student performance improves as the result, the intervention was effective. However, student performance does not always improve on the subsequent iteration, then it is more difficult to conclude why it is below the acceptable cutoff. A number of variables that might have a negative effect on student performance need to be considered in this case. The variables are summarized below in Table 23. This is not intended as a comprehensive guide, but as a starting point for figuring out where to look for the causes of lower than desired performance.

In general, it is possible for a group of students to be different from one semester to another semester. While analyzing the data, the questions in the table were used to help

determine the cause of low or decreasing student performance, and how performance can be improved on the next iteration.

Variable	Questions to ask
Material Covered	Is this the first time for the concept or material to be included in the class? If ~~"y~~es" were the instructional materials sufficient, or do they need enhancing or clarification? If "no" do the instructional materials require enhancement, or are new instructional materials needed?
Instructor	Is this the same instructor that taught the previous semester? Does the instructor introduce the topic at the same time as before? Was there a change to his/her teaching methods? Did he/she decrease the time spent on teaching a specific topic?
Class period	Were any classes cancelled due to weather, sickness, or other reasons? Are classes scheduled very early in the morning or late at night?
Test	Is this the first time the question is used on the test? Are all students taking the test or are some exempt? Does the test question correspond to the level of difficulty at which the principle was taught? Are there too many questions on the test? Did students have less time to complete the test?
Student	Are there more students in the class? Are students' backgrounds the same as before? Is this an off-semester? Are the general attitudes of the class different this time (students more shy, indifferent, or seem to be lower performing students)

Table 23. Examples of variables affecting student performance and corresponding questions

Observation #4.3: Assessment questions should have the appropriate difficulty level.

Why this is important?

Assessment questions should correspond to the level on which students are

expected to learn the topic. Sometimes it is very difficult to do this when students are expected to attain at just the KC level of proficiency in some area. This is because instructors often find it difficult to ask KC level questions because they think they are too easy. Sometimes, easy questions may work as a ―sanity check" in that they can reveal if students have learned a simple skill when the instructor would just assume that they had learned it. On the other hand, the test question should be challenging enough to make students think about what they have learned, while also being within their capabilities. Everyone in the class scoring 100% on a specific RCI reasoning principle is an obvious indication that the assessment question is too easy. For an example of the opposite kind, see the discussion earlier in this section for Observation #4.1.

Details:

In Spring 2012, 100% of the undergraduate CPSC215 software engineering students scored 100% on the two final exam questions. One of the questions was asking for a definition of a contract programming, another was to choose the correct definition for the term loop invariant, both were multiple choice questions. Though students are expected to be familiar with the concepts on the KC level of Bloom's taxonomy (the lower level), it was discovered that the questions were not challenging enough. The results are shown in Table 24. This means that next semester the difficulty of the question should correspond to the difficulty level at which students are expected to master the material.

Evidence:

RCI #	Class Avg	%students with >= 70%
RCI# 4.2.	100%	100%
RCI# 4.3.2.1	100%	100%

Table 24. Evidence from CPSC215 Spring 2011 (observation #4.3)

Observation #4.4: It may take more than one intervention to ask the right questions.

Why is this important?

There is a concern that if student averages are too low, it may mean that the reasoning principle is too difficult for the students. Sometimes the cause for low numbers is the difficulty of an assessment question. The level of difficulty of an assessment question should correspond to the difficulty level of the learning outcome at which the material was taught. Too difficult, and students get frustrated and fail the question. Too easy, and there is no challenge. This is again an example of where RCI is useful. Learning outcomes are important because they guide instructors on the level of difficulty that a test question should have.

Details:

In Fall 2011, one final exam question in CPSC215 was too difficult for students, and almost none of the students got the answer right. The material was taught on the KC-level of Bloom's taxonomy, but the assessment question was asked on the SE level. Adjusting the difficulty of the question has helped, and next semester the difficulty level

of the question was more appropriate, with 43% of students scored 70% or higher. This is still not an ideal situation and requires more investigation into possible intervention.

Evidence:

RCI #	Fall 2011		Spring 2012	
	Class Avg	% students with >=70%	Class Avg	% students with >=70%
RCI#3.4.3	4%	0%	43%	43%

Table 25. Evidence from CPSC215 Fall 2011and Spring 2012 (observation #4.4)

Observation #4.5: Online tutorials serve as a bridge between two courses in the software engineering sequence: CPSC215 and CPSC372.

Why is it important?

Before taking CPSC372 undergraduate computer science students at Clemson are required to take the prerequisite course CPSC215. In order to review the material taught in CPSC215, students are assigned an out of class homework of taking online tutorials described in section 3.4.2.3. Not only did the tutorials improve student performance on the subsequent quizzes, but they also reduced the time that instructor needed to spend reviewing the material in class.

Details:

To test the effectiveness of the tutorials we have conducted an experiment. Quizzes were given to students after they had taken the online tutorials, one to review the

mathematical strings, another to review operation parameter modes, and the third one to review operation specifications by using test cases. Because mastering RCI#3.4 requires students to have some basic prerequisite skills (starting with mathematical notation, understanding basic specifications, and finally test cases) all three tutorial modules were utilized.

Student performance was assessed after taking each of the three modules. Eighteen undergraduate students in CPSC372 participated in the experiment. The classroom was divided into groups of approximately equal size by the professor. The test group was assigned an appropriate tutorial module where a number of short informational slides are followed by the practice questions with detailed feedback, while the control group did not take the tutorial. After that a quiz was given to both groups to check their understanding of the topic. To keep variables as consistent as possible, all the quizzes were conducted by the same instructor, and students were treated in the similar fashion across all the three experiments.

As the data in Table *26* demonstrates, using the online tutorial to review the reasoning principles learned in the previous course was effective. The first tutorial has the most dramatic difference with control group scoring an average of 78% on the quiz, and test group scoring 97%. The second tutorial on parameter modes indicates the average of the control group of 90%, and test group of 95%.

The third case is the most interesting. Though the results of the conducted experience revealed the difference of the averages of the control and test groups to be only 1%, another interesting fact was observed. A week before the tutorial on test cases

was assigned to the students in the test group, the professor gave all the students in the class a short quiz to check how well they had understood the creation of test cases from specifications. The quizzes were graded as he usually does. A week later the tutorial that covers test cases more in depth was assigned to the test group of students, but not to the control group. After the test group students took the tutorial, all the students in the class took another quiz on test cases. As shown in Table 27, students in both the test and control groups scored lower on the previously given routine classroom quiz. After taking the tutorial the increase in average for the control group was 13%, and rather dramatic for students in the test group at 35%. It was concluded that the online tutorial can be used as an effective tool to review the material learned in the previous course.

Evidence:

Group	Mathematical Strings		Parameter Modes		Test Cases	
	# students	Avg	# students	Avg	# students	Avg
Test	8	7.75/8 (97%)	7	9.5/10 (95%)	7	9.3/10 (93%)
Control	9	6.2/8 (78%)	11	9.0/10 (90%)	8	9.2/10 (92%)

Table 26. Results of the experiment with three tutorial modules (observation #4.5)

Group	Routine Classroom Quiz		Tutorial Quiz		Percent Improvement
	# students	Average	# students	Average	
Test	6	5.8/10 (58%)	7	9.3/10 (93%)	60%
Control	10	7.9/10 (79%)	8	9.2/10 (92%)	17%

Table 27. CPSC 372 student averages after a routine classroom quiz and after tutorial quiz (observation #4.5)

4.4.5. Observations Related to Instructors Teaching Reasoning Principles

Instructors with various backgrounds and levels of experience can successfully teach the RCI reasoning principles. The following observations are discussed in this section:

<u>*Observation #5.1:*</u> An experienced computer science instructor who has never taught RCI reasoning principles can teach these principles as good as an instructor who has already taught them.

<u>*Observation #5.2:*</u> Instructors who have never taught reasoning principles before can teach them well from the outset.

<u>*Observation #5.1:*</u> An experienced computer science instructor who has never taught RCI reasoning principles can teach these principles as well as an instructor who has already taught them.

Why is it important?

A long-term goal is to introduce the RCI reasoning principles into computer science curriculums of other colleges and universities. However, there exists a concern that all computer science educators may not be prepared to teach the RCI reasoning principles. This data shows that students are able to learn from an instructor who is experienced in teaching a variety of computer science courses, and who has never taught the RCI reasoning principles. Because each RCI principle has a learning outcome on one of the three levels of difficulty, these learning outcomes help instructors to determine how to teach these reasoning principles. If instructors know which RCI principle is to be

taught and at which level of difficulty, they can develop the appropriate instructional materials or adopt some of our instructional materials.

Details:

Table 28 presents the assessment data from CPSC215 collected in Fall 2012 semester at Clemson. Instructor 1 has been teaching RCI reasoning principles since 2008. Instructor 2 is an experienced computer science instructor who taught undergraduate computer science courses for a number of years. Instructor 2 taught the RCI reasoning principles for the first time in Fall 2012. The six reasoning principles were assessed at the end of the semester in each section. Students in the section taught by Instructor 2 scored comparable to those in the section taught by Instructor 1, and higher in some instances. Though instructor 2 had access to the instructional and assessment materials (described in Chapter 3), we can say with confidence that this success is due to the fact that experienced educators are already familiar with good teaching methods and when provided with the framework of the Reasoning Concept Inventory, associated learning outcomes, and with access to the instructional materials that are available free from out project website, they can achieve satisfactory results.

Evidence:

Reasoning Topic	Difficulty Level	Fall 2012 Instructor 1	Fall 2012 Instructor 2
		Class Avg	Class Avg
RCI#3.4.3.2	AA	44%	60%
RCI#4.1.1.3	KC	71%	75%
RCI#4.2.1.1	KC	95%	88%
RCI#5.2.2.1	KC	81%	100%
RCI#5.3.2	AA	48%	63%
RCI#5.2.2(1, 2)	AA	81%	85%

Table 28. Fall 2012 CPSC215 student performance in two course sections taught by different instructors (observation #5.1)

Observation #5.2: Instructors who have never taught reasoning principles before can teach them well from the outset.

Why this is important?

Since one of our goals is to eventually have these reasoning principles taught to computer science students of other colleges and universities, this is an important observation. At the beginning of this chapter we listed institutions that are already starting to introduce these principles in their software engineering curriculums. There is a concern among instructors as to how effectively they can teach the topics that have either not been taught in their course before, or were not taught in a systematic manner. These instructors themselves must thoroughly understand the topics to be able to confidently present them to students on the appropriate level of difficulty and at the appropriate time. The key consideration here is that most instructors already have the necessary prerequisite knowledge to teach these skills, and may only need to acquire a small set of

the RCI principles. The Reasoning Concept Inventory has organized all the necessary skills into the five relevant areas, and the learning outcomes at different levels of difficulty along with the sample exercises provided in Appendix L will greatly simplify this task. As discovered, not only good instructors can master the topics themselves, but also, students can learn from an instructor teaching the RCI principles for the first time.

Details:

Table 29 shows data that an instructor that has not previously taught the principles from the Reasoning Concept Inventory, and has not been exposed to RESOLVE can effectively teach the reasoning principles. Specifically, the data shows that 18 students in CS315 undergraduate level SE course have completed an assignment that included a task of constructing verification conditions for a reasoning table. The class average was 71%, where 39% of students have scored greater or equal to 70 percent. It is also observed that student performance dramatically increased the second time this instructor taught these principles: during Fall 2011 the class average was 94% with 100% of students getting a score of 70% or higher.

Evidence:

RCI	Spring 2011		Fall 2011	
	Class Avg	% students with >=70%	Class Avg	% students with >=70%
RCI#5.2.2	71%	39%	94%	100%

Table 29. Evidence from CS315 Spring 2011and Fall 2011 (observation #5.2)

4.4.6. Observations Related to Incorporating the RCI Principles into a Course

It is possible to incorporate a large number of reasoning principles into the course. The following observations are discussed in this section:

> *Observation #6.1:* It is possible to integrate reasoning topics into a software engineering course and teach the traditional content with a ―reasoning" perspective.
>
> *Observation #6.2:* It is possible to cover many reasoning topics in a short period of time.
>
> *Observation #6.3:* Students can learn reasoning principles as well as they learn traditional software engineering principles

Observation #6.1: It is possible to integrate reasoning topics into a software engineering course and teach the traditional content with a ―reasoning" perspective.

Why this is important:

The ultimate goal is to integrate reasoning topics into the undergraduate computer science curriculums in other colleges and universities. However, there is a concern that the new RCI reasoning principles will have to displace the traditional course content. As our experience indicates, reasoning topics can be integrated with the traditional software engineering material. Students still need to master the traditionally taught software

lifecycles, design patterns, and other topics. The reasoning concepts should be integrated into the course wherever they logically fit.

Details:

The discussion of this observation is not based on a specific data table. Instead, the evidence is in part supported by the discussion during the focus group meeting that we have conducted. Three of the graduate teaching assistants (PhD students in their last year) who taught the undergraduate software engineering course at Clemson (CPSC215) have successfully integrated reasoning topics into the existing curriculum. In this course, a number of the RCI reasoning principles were already known to students from a previous course on discrete math, and other reasoning principles are introduced on the KC level of proficiency without requiring significant classroom time. These instructors report that they have integrated reasoning principles into the existing material without difficulties. Please refer to the Appendix G for the transcript of the focus group meeting. By the time these students progress to a junior-level course, such as CPSC372, they have already learned many of the reasoning principles at the KC level in CPSC215. Other principles will be taught in CPSC372, some at the KC level, while others at more advanced levels. Also, a number of the RCI principles are already included in the software engineering curriculum, though they have not been taught in the context of an RCI framework with learning outcomes.

Observation #6.2: It is possible to cover many reasoning topics in a short period of time.

Why this is important?

The Reasoning Concept Inventory is refined to several levels of detail, and at first glance it may appear very difficult to incorporate a reasonable number of the RCI reasoning principles into the existing course content. It is important to remember that the coverage of different principles will be distributed across the curriculum, and each participating course will include principles on the appropriate level of difficulty.

Details:

RCI expands into several levels, with the fourth level containing concept details. However, some concepts have already been mastered in the prerequisite courses. So in subsequent courses they will simply require a review for most students, and will not require additional instructional time. For example, students who have taken the prerequisite Discrete Math course will be familiar with sets, proofs, and etc. Some RCI reasoning principles will be new but taught at the KC level of the Bloom's taxonomy, and students just need to be generally familiar with the new concept. A few of the other topics will be new to students and will be taught at the more advanced AA or SE levels. It is important to remember that the mastery of reasoning principles will occur over many courses and usually in the junior-level courses. The reasoning principles will be distributed across the curriculum, as shown in Table 4 in Chapter 3.

Evidence:

During the focus group meeting it was discovered that the instructors had covered a very large number of the RCI reasoning principles during the 2.5 - 3 week period. The principles were integrated where they logically fit within the traditional content. Some principles, such as sets and numbers were already familiar to the students, and instructors spent only a few minutes reviewing them. Some topics, such as constructing verification conditions, were brand new and taught at the AA level, and took more than one class period to cover. The full transcription of the focus group meeting is located in Appendix G.

Observation #6.3: Students can learn reasoning principles as well as they learn traditional software engineering principles

Why is it important?

The CPSC372 Software Engineering course curriculum contains traditional SE topics, such as requirements analysis and design. Students spend approximately two thirds of the duration of the course learning these topics, and about one third of the course learning the RCI reasoning principles. Since the RCI reasoning principles are new, we need to confirm that students can master the RCI reasoning principles as well as other software engineering topics covered in the course.

Discussion:

Table 30 shows that there is really no difference, i.e., students learned the RCI

material as well as other general SE topics. At Clemson, the spring semester is sometimes an –off semester" for the SE course, that is, the Spring SE course is often taken by the students who were not able to take this course in Fall due to their schedule conflicts, transferring students, and students who failed it the previous semester. During the Spring semester the grades tend to be lower; for example, there were seven students out of 26 who earned an –A" in Fall 2011 and only two out of 32 who earned an –A"s in Spring 2011. The spring class also had five students with –C" or below, whereas there was only one –C" in the fall course. There were 18 students in each offering.

Evidence:

Semester	Class Avg on RCI topics	Class Avg other SE topics
Fall 2010	85%	85%
Spring 2010	79%	78%

Table 30. Fall/Spring 2011 student averages on RCI topics vs. other SE topics (observation #6.3)

4.4.7. Observations Related to 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 [47]. 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 CPSC215 and CPSC372 indicate that students have positive attitudes learning the RCI reasoning principles. A questionnaire was administered to students in each section at the beginning and end of the semesters in two courses across the curriculum: sophomore-level software development foundations (CPSC215) and junior-level software engineering (CPSC372). This summative survey data is presented in Appendix F and the full version of the survey, along with the consent form that students receive prior to their participation, is located in Appendix E.

The questionnaire assesses the 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 student's 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 following observations are discussed in this section:

Observation #7.1: Students in CPSC215 have positive attitudes towards building high quality software

Observation #7.2: Students in CPSC372 have positive attitudes towards mathematical reasoning principles

Observation #7.1: Students in CPSC215 have positive attitudes towards building high quality software

In the sophomore-level software development foundations course (CPSC215), the students' 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). The (2-tailed) significance is 0.02. Since high quality software development is the ultimate goal of the project, the final result in attitude change is statistically significant.

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). Here the 2-tailed significance was 0.03. Likewise, 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. The third question with the same 2-tailed significance of 0.03 was 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).

Observation #7.2: Students in CPSC372 have positive attitudes towards mathematical reasoning principles

In the junior-level software engineering course (CPSC372) significant changes were observed in three of the survey questions. The question that has the most statistically significant attitude change with the 2-tailed significance of 0.00 is "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. A second significant result from CPSC372 comes from the question "Having precise mathematical descriptions for each software component improves the likelihood that my software will be correct" had 2-tailed significance of 0.01 (pre: M=5.04, SD=0.77, post: M=4.39, SD=0.94). The last question "I believe that there is a strong correlation between a person's mathematical background and their ability to design and implement large systems correctly" had a 2-tailed significance of 0.02 (pre: M=4.78, SD=0.95, post: M=4.00, SD=1.28). Detailed tables with the survey results for CPSC215 and CPSC372 are located in Appendix F.

The results of the attitudinal surveys indicate that the attitudinal changes occurred exactly in the areas emphasized in each course. CPSC215 taught basic concepts of software design, and CPSC372 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.

4.4.8. Observations Supported by Indirect Evidence

The indirect evidence gathered during a focus group meeting lead us to the observations that are collectively discussed below.

Observation #8.1: Novice instructors can effectively teach the RCI reasoning principles.

Observation #8.2: There appear to be no insurmountable challenges integrating the RCI reasoning principles with traditional software engineering material.

Observation #8.3: A large number of the RCI reasoning principles can be taught at different levels of difficulty during a short period of time.

A focus group meeting was conducted with the three instructors who taught CPSC215, the undergraduate software development foundations course at Clemson. These instructors are PhD students in their last year of study in the School of Computing at Clemson. All the IRB paperwork necessary to conduct such meeting, including the participants' signed consent forms, was completed, and is available upon request. The meeting was conducted by an IRB-certified moderator, who followed the guidelines for the focus group meeting outlined in the Manual for Conducting Focus Group [101]. According to the manual, "the goal of any focus group meeting is to elicit a rich discussion", and to have "participants' memory primed by the comments from other

members of the group in the environment that is conductive to frank sharing of ideas and opinions" [101].

The goal of this meeting was to discuss the Reasoning Concept Inventory, to determine what RCI items have been introduced in the course, and to take inventory of the successes and challenges. The meeting was led by a moderator whose function was to steer the discussion towards the topics relevant to the goal, and to encourage the instructors to actively participate and share their opinions and experiences. The moderator's role is not to express an attitude towards the opinions of the participants, and neither agree, nor disagree with their opinions, but simply to make notes and lead the discussion. The meeting was also digitally recorded according to focus group guidelines, in order to be transcribed and analyzed later.

The focus group meeting produced valuable information. The instructors provided useful feedback, and the most relevant items are discussed below. The full transcription of the meeting is available in Appendix G. It was discovered that a large number of RCI principles were covered in the course. Each instructor spent about 2.5 - 3 weeks of the course teaching the RCI reasoning principles. Each instructor has an individual teaching style, and introduced reasoning topics where they logically fit within the traditional material. All the instructors covered an almost identical set of reasoning principles. When asked to place a mark in the RCI table next to the items that they have taught/tested and taught/not tested, the marked sets from each instructor are almost identical. Some of the principles were covered at the KC level of the Bloom's taxonomy, and others at more advanced levels.

Though teaching reasoning topics was a new endeavor to these instructors, they only experienced minor difficulties teaching them. They found the guidelines, instructional materials, and assessment questions that were provided to them useful. We can conclude with confidence that even a novice instructor can be successful in teaching reasoning topics. With minimal guidance, instructors can tailor assessment questions to meet their individual teaching styles and the material coverage level.

The difficulties of incorporating RCI topics into the existing course material were also discussed. Though one of the instructors indicated that introduction of reasoning topics initially seemed like a ~~h~~ard left turn", the others did not see a challenge in incorporating the topics into the existing curriculum.

Another important conclusion was that the students were able to learn reasoning topics, and performed comparably to the traditional topics. This qualitative evidence collected at the group meeting correlates with the research data from CPSC372 course located in Table 27. Both indicate that students are capable of learning reasoning topics just as well as traditional ones.

Some challenges were noted as well. Some of the students had insufficient programming background, and some students had misconceptions that mathematics is always hard. As one of the instructors pointed out, proofs appeared to be challenging to some students, because they lacked prerequisite knowledge of mathematics.

4.5. Chapter Four Summary

This Chapter lists a number of assessments questions used to test reasoning principles at the appropriate levels of difficulty in three courses: sophomore-level Software Development Foundations course (CPSC215) at Clemson, junior-level Introduction to Software Engineering course (CPSC372) at Clemson, and junior- level Software Engineering course (CS315) at the University of Alabama. The results are followed by several categories of observations that are based on experimental data reflecting the evidence of student learning. Some of the observations are conclusive, while others are not. More observations will be made as data collection continues. But the important conclusion is that Reasoning Concept Inventory is foundational for both teaching and assessment. Without the RCI and learning outcomes with different levels of difficulty it would be difficult to devise a systematic method of teaching reasoning principles; difficult to build good assessments, and difficult to pinpoint where intervention is needed to improve learning.

CHAPTER FIVE

SUMMARY OF RELATED WORK IN FORMAL METHODS

The idea of teaching mathematical reasoning and formal methods has several pioneers [9, 18, 40, 41, 60, 92, 105, 112, 125, 130]. In [56], Henderson gives several reasons for why mathematics is important for software engineers. Not just mathematics itself, but also the ability to think mathematically is discussed in [57]. Abstract software artifacts require abstract reasoning. A software system is simply a mathematical model of some process of desired computation, and mathematics is the only tool for rigorous reasoning and analysis. Henderson gives a general definition of the mathematical reasoning: ―Applying mathematical techniques, concepts, and processes, either explicitly or implicitly, in the solution of problems", and summarizes his view on the importance of mathematics stating that software engineers ―must learn to use mathematics to construct, analyze and check models of software systems, to compose systems from components, to develop correct, efficient system components, to precisely specify the behavior of systems and components, and to be able to analyze, test and evaluate systems and components" [56].

In an earlier paper [33], Devlin mentions the benefits of mathematics for software engineers. He states that mathematics is an essential prerequisite for a software engineer, because what students get from academic experience with mathematics is the experience of rigorous reasoning with purely abstract objects and structures, and in software engineering every concept, construct, and method is abstract. Even when students say

they do not (explicitly) use mathematics, they still use it every day, via mathematical thinking.

This work is supported by the earlier work of Howe, Long, Sitaraman, Weide, and others discussed in [67] and [123], where the authors discuss introduction of component-based software engineering (CBSE) principles early in the undergraduate computer science curriculum. They focus on CS1/CS2 content and investigate approaches to teaching, specifically the "components-first" programming-first paradigm. Their objective is to teach CBSE concepts early in the computer science curriculum using the RESOLVE framework as a delivery vehicle, which they state has advantages over the traditional presentation. Among the benefits are knowledge that it is possible to prove that a software component works correctly without executing the code, the ability to understand formal specifications up to several lines long without having taken prerequisite courses in discrete math, and the ability to internalize the language-neutral component taught. Principles taught in [123] include a modular style of software development, an emphasis on human understanding of component behavior using formal specifications, in addition to classical topics. Their statistically significant results confirm that CBSE principles can be taught without displacing the "classical" principles taught in introductory courses, and that students can understand and reuse formally specified components without knowing their implementations.

Following up on this work, Bucci, et al., document the importance of teaching abstraction and mathematical thinking early in the computer science curriculum [16]. They note that "abstraction is one of the cornerstones of software development and is

recognized as a fundamental and essential principle to be taught as early as CS1/CS2." They have experimented with presenting list and other components in RESOLVE using mathematical types to describe program types and their behavior, and specifically, using mathematical models that are already familiar to all the CS1/CS2 students (sets, functions, integers). They have concluded that abstraction can be successfully taught as early as in the CS1/CS2. In another paper [83], the authors present a specification-based approach to reasoning about component-based programs used to reason about pointers, and show how their modeling technique alleviates the problems caused by pointer misuse.

Bucci, et. al. [18] address pedagogical challenges of presenting and using formally-specified software components in CS1/CS2 courses. Their "low-tech" approach of employing physical manipulatives (toys such as plastic cups or Lego© blocks) is not only fun, but also "amazingly effective" in helping students to develop mental models for mathematical concepts and to understand the behavior of software components through their cover stories (specifications) without knowing implementation details.

Teaching formal methods is closely related to mathematics and logic, so Alstrum et. al. describe the growth path of Formal Methods in Education, and point out that as far back as ACM/IEEE Computing Curriculum 1991 and 2001, and SWEBOK (http://www.computer.org/portal/web/swebok) have argued for the use of discrete math and formal methods in CS education [4].

Maibaum has investigated differences between conventional engineers and software engineers, stating that software engineering is in fact engineering, and it needs

mathematical foundations [94]. He notes that to develop mathematical foundations students need formal methods, and that typical formal methods curricula fails to prepare software engineering students to be engineers. The deficiency was also mentioned back as early as 1990 by D. L. Parnas, who has criticized computer science programs for adopting the latest ―fads", insisting that it should take a classical engineering approach of teaching fundamentals. He notes that ―those who work in theoretical computing science seem to lack an appreciation for the simplicity and elegance of mature mathematics" [110].

Other related work in the field of formal methods education may be found in [1, 23, 31, 35, 62, 82, 93, 87]. Liu suggests introducing specification patterns, using formal notation, diagrams, and a specification language. Chiang chose to use TUG method, a specification language based on definite clause grammars and regular expressions in combination with the waterfall model of software development [23]. Zingaro uses a lightweight formal method approach with Java using invariants and avoids formal proofs [136].

In [109], Eiffel was used with a light-weight approach. A tool designed to check correctness of simple programs constructed according to the structured programming methods and capable of detecting too weak invariants, finding errors in statements, and alerting out-of-bounds in assertion and run-time errors is presented in [34]. Jahob analysis system that uses a subset of Java as a programming language, a subset of Isabelle as the specification language, and incorporates reasoning procedures for sets with cardinality constraints in discussed in [86].

Many attempts have been made to minimize mental resistance [49] typical to many students when it comes to learning formal methods. McLaughlin also introduces formal methods in the first year, teaching a technique of backward derivation of programs that uses logic notations, and proofs designed by Dijkstra, Feijen, and others [96].

The importance of software specifications and reasoning is also receiving industrial attention. Davies states that our reliance on software systems outweigh our trust in its correctness, and notes that one of the ways of building trust into our systems is using formal methods during their development [30]. Mike Holloway, a NASA Langley Research Center research engineer and a member of the NASA formal methods team since 1992, addresses the need for formal methods and mathematics education for software engineers in [65]. He has noted that though software engineers are somewhat reluctant to use formal methods, the use of formal methods is the most rigorous fault avoidance technique. He has compared formal methods at that time (1997) to the Wright Flier rather than a Boeing 777, but remarks that a combined effort of mathematicians, logicians, researchers and practitioners will bring improvement to the field.

Formal methods are applied at various points of the software lifecycle [134], used for specifying behavior or property of the system being developed, or for verification - proving correctness with respect to the formal specification or property. Such formal specification tools as Alloy [6, 44], Communicating Sequential Process [15, 103, 135], Larch [22, 133], LCLint [38], Vienna Development Method [42], and Z [11, 103, 111] have been used to build models of how software should operate, which are then validated via math proofs to show if the system is correct.

A number of papers have been written on formal methods for industrial applications. Formal methods in high performance computing, intelligent swarm technologies, and security engineering, and others are discussed in [2, 13, 20, 45, 61, 95] respectively. Woodcock in [134] offers a comprehensive overview of formal methods in industry and lists a number of successful projects that have employed formal methods (Transputer project, Airbus, Tokeneer Secure Entry, etc.). The importance of formal methods for ensuring component interoperability is highlighted in [24, 30]. In [24], researchers used the ICM (Interoperable Component Module) enhanced by formalism, and provide a testing framework based on a formal specification model. In [30] the author states that it is ―time for formal methods to save the world." He shows that formal methods are of importance in real-life applications, and we need education in formal methods, because ultimately that is the most effective way to transfer formal methods into industry [93].

A significant body of research exists in the area of software verification [19, 63, 68, 102, 117, 120, 129]. Efforts include development of verification systems for existing and new languages. While some systems are for functional languages, such as Isabelle [72], and PVC[29], others support imperative languages, such as and Ada-based SPARK [71], Dafny[14], JML and Jahob for Java component specification and verification [86, 89], and VCC for C[131]. In [73], the authors provide comparison of a number of verification systems. While, in principle, any of these systems could be used to reinforce the RCI reasoning principles, RESOLVE language and systems are especially tuned for an educational setting.

Unlike these prior efforts which focus on reasoning in general, in this dissertation we have identified a reasoning concept inventory that lists specific reasoning skills that students need to learn to reason analytically about software components. Because this inventory has a fine level of granularity, and has corresponding learning outcomes on three different levels of difficulty, it can be used for pinpointing learning obstacles, conducting an intervention, and evaluating student performance. Such an assessment, not possible without the RCI, was not attempted in the prior efforts.

CHAPTER SIX

SUMMARY AND THE FUTURE DIRECTIONS

In this dissertation, the central elements of a Reasoning Concept Inventory to be taught across the undergraduate software engineering curriculum have been identified. Several iterations of revisions have been used to include the most pertinent details and to eliminate those that are less critical. This has been done in the collaboration with professors, instructors and researchers of other Universities. The reasoning principles included in our Reasoning Concept Inventory are both necessary and sufficient to teach students to reason mathematically about software correctness. These principles are organized into five current knowledge areas including Logic, Discrete Mathematical Structures, Precise Specifications, Modular Reasoning, and Correctness Proofs.

Using Bloom's taxonomy as a blueprint, learning outcomes with three levels of difficulty have been developed for a subset of RCI knowledge areas including Precise Specifications, Modular Reasoning and Correctness Proofs. Boolean Logic and Discrete Mathematics areas have been excluded from the subset because these topics are covered in a variety of courses at the earlier stages of the computer science education. However, the possibility of an earlier topic to be revisited later with a greater level of complexity is not excluded. A number of teaching methods used to communicate the RCI principles to our students (TSRA, Web Interface, tutorials, assigned projects, a variety of customized exercises and classroom lectures) have been detailed.

The analysis of the data collected during the experiment spanning a number of academic semesters has been presented, and relevant observations are made. It is

confirmed experimentally that students can learn to reason about software components mathematically, and have positive attitudes doing so. It has also been shown that using learning outcomes on the appropriate levels of difficulty to drive teaching of the reasoning skills allows educators to pinpoint exactly which skills students have mastered and which they have not. The findings continue to be used for curriculum alignment and for adjusting the amount and quality of time that instructors spend teaching various skills, as well as the methods they employ.

Development of the Reasoning Concept Inventory for the Software Engineering area of STEM, along with the Learning Outcomes that precisely state the expected level of mastery for each item, and using that as a basis for systematic teaching, accurate assessment, and intervention are the key contributions of this work.

This dissertation demonstrates that using our Reasoning Concept Inventory (RCI) in conjunction with learning outcome-driven instruction is an indispensible tool for teaching future software practitioners. Teaching the right principles the right way, with software correctness as an ultimate goal, the students will not be only able to satisfy the computer science program requirements and the IEEE Computing Curriculum 2008 requirements, but also will be able to meet the high quality expectations that today's pervasive computing demands.

Several future directions will extend the benefits of this research are outlined below.

- Continue meticulous data collection and analysis, and continue evaluating how students are learning the RCI principles in order to discover if/what interventions are needed;

- Use assessment data based on the RCI to evaluate if new/improved tools are needed to teach reasoning principles to the undergraduate software engineering students;

- Provide a large bank of useful exercises with the answers and explanations to serve as an instructional resource for educators using RCI at our and other institutions;

- Introduce reasoning principles to other schools and provide educational support to the instructors on their initial stages of teaching reasoning principles;

- Provide educational workshops at conferences and publish research papers to educate the computing community about the further findings of this project, and to encourage them to use the RCI as the basis for teaching and assessing reasoning principles

APPENDICES

Appendix A

Bloom's Taxonomy: Cognitive Domains vs. Cognitive Domain Keywords

Cognitive Domains		Cognitive Domain Keywords
"LC" Level Verbs	Knowledge	cites, counts, defines, describes, draws, identifies, knows, labels, lists, matches, names, outlines, points, recalls, recites, records, recognizes, repeats, reproduces, selects, states, tabulates, underlines
	Comprehension	associates, classifies, compares, converts, contrasts, differentiates, discusses, distinguishes, estimates, explains, expresses, extends, extrapolates, generalizes, gives examples, locates, paraphrases, reports, restates, reviews, rewrites, summarizes
"A" Level Verbs	Application	applies, calculates, changes, computes, constructs, demonstrates, determines, discovers, examines, illustrates, interprets, locates, manipulates, modifies, prepares, produces, relates, reports, restructures, solves, translates, uses
	Analysis	analyzes, appraises, breaks down, calculates, classifies, compares, contrasts, debates, diagrams, deconstructs, differentiates, distinguishes, examines, experiments, identifies, illustrates, infers, inspects, outlines, questions, relates, selects, summarizes, tests
"SS" Level Verbs	Synthesis	categorizes, combines, compiles, composes, constructs, creates, designs, explains, formulates, integrates, generates, manages, modifies, organizes, plans, prepares, produces, proposes, rearranges, reconstructs, revises, specifies, summarizes, writes
	Evaluation	assesses, chooses, compares, concludes, contrasts, criticizes, critiques, determines, estimates, evaluates, explains, grades, interprets, justifies, measures, ranks, rates, recommends, revises, scores, selects, standardizes, summarizes, supports, tests, validates, verifies.

Appendix B

Complete Reasoning Concept Inventory

Reasoning Topic	Subtopic Summary	Concept Term Highlights	Concept Details
1. Boolean Logic	1.1. Motivation	1.1.1. Motivation for Boolean Logic	
	1.2. Standard logic symbols	1.2.1. Connectives including implication	1.2.1.1. Simple statements 1.2.1.2. Connectives (NOT, AND, OR, IF... THEN, IFF) and compound statements 1.2.1.3. Truth tables 1.2.1.4. Logically equivalent statements
		1.2.2. Quantifiers	1.2.2.1. Universal quantifier 1.2.2.2. Existential quantifier
	1.3. Standard terminology	1.3.1. Proposition	1.3.1.1. Propositional variables 1.3.1.2. Compound propositions 1.3.1.3. Proof arguments
		1.3.2. Predicate logic	1.3.2.1. Predicate calculus
		1.3.3. Proof	1.3.3.1. Informal proof 1.3.3.2. Axiom 1.3.3.3. Premise 1.3.3.4. Conclusion
	1.4. Standard proof techniques	1.4.1. Supposition deduction	
	1.5. Methods for proving	1.5.1. Direct proof 1.5.2. Proof by contradiction	

		1.5.3. Vacuous proof 1.5.4. Trivial proof 1.5.5. Proof by cases 1.5.6. Exhaustive proof 1.5.7. Proof by induction	
	1.6. Proof strategies	1.6.1. Forward reasoning 1.6.2. Backward reasoning	
	1.7. Rules of inference	1.7.1. Modus ponens 1.7.2. Simplification Conjunction 1.7.3. Universal instantiation 1.7.4. Universal generalization 1.7.5. Existential instantiation 1.7.6. Existential generalization	
2. Discrete Math Structures	2.1. Motivation	2.1.1. Motivation for Discrete Mathematics	
	2.2. Sets	2.2.1. Set basics	2.2.1.1. Set naming conventions 2.2.1.2. Universal set U 2.2.1.3. Empty set ϕ 2.2.1.4. Element naming conventions 2.2.1.5. Subset 2.2.1.6. Subset Theorem 2.2.1.7. Venn diagrams

		2.2.2. Set notations	2.2.2.1. Union 2.2.2.2. Intersection 2.2.2.3. Disjoint sets 2.2.2.4. Relative compliment 2.2.2.5. Absolute Compliment 2.2.2.6. Power set 2.2.2.7. Power set for a finite set 2.2.2.8. Cartesian product 2.2.2.9. Extensibility over finite set 2.2.2.10. Cartesian product of a set on itself
		2.2.3. Laws of algebra on sets	2.2.3.1. Idempotent laws 2.2.3.2. Associative laws 2.2.3.3. Commutative laws 2.2.3.4. Distributive laws 2.2.3.5. Identity laws 2.2.3.6. Involution laws 2.2.3.7. Complement laws 2.2.3.8. DeMorgan's laws
	2.3. Strings	2.3.1. String basics	2.3.1.1. String variable name conventions 2.3.1.2. Alphabet set Σ 2.3.1.3. Strings over the alphabet set Σ*
		2.3.2. String notations and properties	2.3.2.1. Empty string 2.3.2.2. String length 2.3.2.3. Concatenation 2.3.2.4. String reversal 2.3.2.5. Substring
	2.4. Numbers	2.4.1. Number	2.4.1.1. Number notations

		representations	2.4.1.2. Number bases
	2.5. Relations and functions	2.5.1. Relations	2.5.1.1. Relation notation 2.5.1.2. Inverse relations 2.5.1.3. Composition of relations 2.5.1.4. Reflexive property 2.5.1.5. Symmetric property 2.5.1.6. Anti-symmetric property 2.5.1.7. Transitive property 2.5.1.8. Atransitive property 2.5.1.9. Equivalence relation
		2.5.2. Functions	2.5.2.1. Function notation 2.5.2.2. Domain and codomain 2.5.2.3. Dependent and independent variables 2.5.2.4. Equal functions 2.5.2.5. Cartesian coordinate system 2.5.2.6. Injective function 2.5.2.7. Surjective function 2.5.2.8. Bijective function
	2.6. Graph theory	2.6.1. Graph basics	2.6.1.1. Graph notations 2.6.1.2. Edges and vertices 2.6.1.3. Degree of a vertex 2.6.1.4. Graph representation 2.6.1.5. Tree and forest
		2.6.2. Graph types	2.6.2.1. Multigraphs 2.6.2.2. Finite graphs 2.6.2.3. Complete graphs 2.6.2.4. k-regular graphs 2.6.2.5. Acyclic graph 2.6.2.6. Planar graphs

	2.7. Permutations and combinations	2.7.1. Notations	2.7.1.1. Permutation notation 2.7.1.2. Factorial notation 2.7.1.3. Combinations notation
3. Precise Specifications	3.1. Motivation	3.1.1. Motivation for interfaces 3.1.2. Motivation for precision	3.1.1.1. Information hiding 3.1.1.2. Independent software development 3.1.2.1. Problems with informal specifications 3.1.2.2. Ease of component integration
	3.2. Specification structure	3.2.1. Specification signature 3.2.2. Usage requirements 3.2.3. Use of math theories 3.2.4. Specification inheritance (enhancements)	3.2.1.1. Concept name 3.2.1.2. Generic parameters
	3.3. Abstraction	3.3.1. Math models for conceptualizing objects	3.3.1.1. Booleans 3.3.1.2. Numbers 3.3.1.3. Integers 3.3.1.4. Strings 3.3.1.5. Sets 3.3.1.6. Functions 3.3.1.7. Relations 3.3.1.8. Cartesian products 3.3.1.9. Other discrete structures

			3.3.1.10. Combination of the above
		3.3.2. Constraints 3.3.3. Trade-offs of alternative mathematical models	
	3.4. Specifications of operations	3.4.1. Initialization and finalization specification	
		3.4.2. Operation signature	3.4.2.1. Operation name 3.4.2.2. Formal parameters 3.4.2.3. Return value
		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 in the post-condition
4. Modular Reasoning	4.1. Motivation	4.1.1. Motivation for reasoning	4.1.1.1. Error detection 4.1.1.2. Code tracing and inspection 4.1.1.3. Formal verification
		4.1.2. Motivation for modular reasoning	4.1.2.1. Problems with implementation to implementation coupling

			4.1.2.2. Desirable coupling through contracts
	4.2. Design-by-Contract	4.2.1. Roles of clients and service providers	4.2.1.1. Specifications as external contracts 4.2.1.2. Client 4.2.1.3. Service provider 4.2.1.4. Client implementation 4.2.1.5. Service provider implementation
		4.2.2. Construction of new components from built-in components	4.2.2.1. Implementation with arrays 4.2.2.2. Implementation with records
		4.2.3. Construction of new components using existing components	4.2.3.1. Implementation of a specification (data representation, code for operations) 4.2.3.2. Implementation of enhancement specification
	4.3. Internal contracts and assertions	4.3.1. Internal contracts for data representations	4.3.1.1. Abstraction functions/ relations (correspondence) 4.3.1.2. Representation invariants (conventions)
		4.3.2. Assertions	4.3.2.1. Loop invariants 4.3.2.2. Progress metrics (loops and recursive procedures)
5. Correctness Proofs	5.1. Motivation	5.1.1. Meaning of correctness	5.1.1.1. Semantics 5.1.1.2. Soundness and

			relative completeness
		5.1.2. Motivation for proofs	5.1.2.1. Partial correctness 5.1.2.2. Total correctness
	5.2. Construction of verification conditions (VCs)	5.2.1. States and abstract values of objects	5.2.1.1. Naming conventions
		5.2.2. Connection between specifications and what is to be proved	5.2.2.1. Assumptions 5.2.2.2. Obligations
		5.2.3. Types of statements	5.2.3.1. Sequential statements 5.2.3.2. Conditional statements 5.2.3.3. Loops 5.2.3.4. Operation calls
		5.2.4. Connection between induction and reasoning	5.2.4.1. Inductive case 5.2.4.2. Base case 5.2.4.3. Termination
	5.3. Proof of 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	5.3.2.1. Direct proofs 5.3.2.2. Rules of inference

Appendix C

RESOLVE Background

While the RCI principles can be presented and learning outcomes defined using any number of formalisms, for purposes of presentation and concrete discussion, the RESOLVE notation has been used in this dissertation.

RESOLVE Overview

The integrated environment includes RESOLVE programming language and built-in specification language, compiler, verifier, and a prover. The programming language itself is an object-based language, similar to object-oriented languages, but without the complications of concurrency, inheritance, polymorphism and built-in pointers. RESOLVE provides only static typing. Its clean semantics and simple syntax are not only easy to learn, but also allow to reason about its components modularly and avoid the common problem of aliasing. To make up for the benefits of pointers RESOLVE incorporates swapping - an operation available by default to all its objects. Kulczycki's work provides more details on the language semantics [80, 81]. RESOLVE verifier is an important component of the environment, as fully verified code significantly improves software reliability and decreases failures. The RESOLVE verifier and prover are also being improved, as described in a number of publications [51, 106, 124, 132].

An indispensable feature of RESOLVE is the built-in specification language that uses precise mathematical notation. Formal specification of a component is the mathematical description of the component's behavior, which informs clients about how

this component is to be used, and directs implementers in what functionality the component is expected to provide. Because it is expressed using the language of mathematics, it is precise and unambiguous.

The importance of formal specifications cannot be understated. Developing software according to formal contracts has great implications in all areas of software engineering. Formally specified and verified software is more reliable, experiences fewer failures, and costs less time-wise and effort-wise at the maintenance stage. Given the last decade's shift towards modular development and reusability, it is especially critical to design components that adhere to formal specifications [43, 64, 88, 122]. The concept of ~~"d~~esign-by-contract", coined by Meyer [97, 98] in relation to his design of Eiffel programming language, explains how software components should collaborate. Some programming languages, such as Dafny [14], Eiffel [99] and RESOLVE, have built-in support for specifications, while others have to rely on a separate specification language, such as Java on JML [89] and C# on Spec# [10].

Because RESOLVE has a built-in specification language that uses familiar mathematical notation, it allows students to drastically reduce the learning curve associated with mastering of a new specification language. RESOLVE has many unique specification mechanisms that we are going to discuss in the following sections.

RESOLVE's Support for Mathematical Modeling

A mathematical model is an abstract way of describing behavior of a particular system by using mathematical terms. In 1974 Pieter Eykhoff defined a mathematical

model as a ―representation of the essential aspects of an existing system (or a system to be constructed) which presents knowledge of that system in usable form" [39]. Mathematical models are extensively used in all areas of STEM, and are rapidly becoming an inseparable part of any software engineering course.

RESOLVE uses mathematical models to conceptualize programming objects ―in usable form." For example, a basic Integer data type in RESOLVE is mathematically conceptualized as a value from the set of mathematical integers Z, with appropriate constraints of *Min_Int* and *Max_Int* that limit the minimum and maximum size of this integer within the program to reflect the fact that machine memory is finite. In addition to this, programming operations are associated with the corresponding mathematical operators: a binary infix operator ―+" corresponds to the same ―+" in mathematics, etc. This can be expressed in the language of mathematics to model an Integer data type in RESOLVE programs:

```
Type Integer is modeled by Z;
   exemplar i;
   constraints min_int <= i <= max_int;
   initialization ensures i = 0;
```

After the suitable model (e.g., a set for Integers) is selected, a number of manipulations can be performed on the Integer type by using notations from corresponding mathematics. Addition, division, increment, etc., can be specified using notations from their math counterparts.

Mathematical models are also used to represent finite sequential data structures such as stacks, lists, or queues. To model a data structure mathematically the suitable

mathematical model first needs to be identified. Not all mathematical models are suitable for representing all data structures. The mathematical model itself and its properties need to be evaluated to determine if it can be used to represent the functionality of the data structure being modeled. For example, for modeling a queue one can consider mathematical sets and strings. Sets will not be a suitable model because items in the sets do not have order. If a queue is viewed as a set, the order of *enqueueing* and *dequeueing* will be lost. On the other hand, strings are ordered, and therefore are suitable for modeling queues.

After a suitable mathematical model is identified, the decision is made on the set of operations that the data structure needs to have. All of the components' operations are then specified via pre- and post-conditions using suitable notations from the corresponding mathematical models. All RESOLVE operations have parameter modes for incoming variables and indicate what happens to these parameters after the operation call. A number of built-in mathematical theories serve the purpose. These details are discussed below.

Components of a RESOLVE Program

This subsection demonstrates how a Queue ADT can be modeled using a mathematical string. RESOLVE has several types of entities to support the software engineering principles of component reuse, abstraction, and information hiding. There are several types of modules that comprise a program in RESOLVE, and they are discussed below.

RESOLVE Concepts

A typically parameterized RESOLVE *Concept* provides a template of a specific data abstraction, analogous to a C++ or Java class declaration (e.g.: Stack, Linked List, Queue, etc.), mathematical model used to represent it (e.g.: mathematical string, set, etc.), constraints (for example: min and max ranges of integers), and a list of specified operations typical to that data structure (for example: Push and Pop for Stacks, Enqueue and Dequeue for Queues, etc.).

The declaration of a *Queue_Template* is provided in Figure 24. It is parameterized by a type of an *Entry* held in the Queue, and a maximum length of the queue. Both of these are passed from user's *Facility* at the time of its instantiation, as shown in a later section. One of the requirements of the Queue instantiation is that its *Max_Length* is greater than zero, that is, the Queue should have at least one element in it.

Queue is modeled by a mathematical string of Entries. Using an exemplar *Q*, the constraint states that the length of the Queue at any time is less than or equal to the *Max_Length* specified at the time of instantiation of the Queue_Template. The specification also states that upon initialization, the value of a Queue will be the empty string.

Next the template includes a list of standard Queue operations. The background information necessary for understanding some of the RESOLVE operation specifications is discussed next.

```
Concept Queue_Template(type Entry; evaluates Max_Length: Integer);
   uses Std_Integer_Fac, String_Theory;
   requires Max_Length > 0;

   Type Family Queue is modeled by Str(Entry);
      exemplar Q;
      constraint |Q| <= Max_Length;
      initialization ensures Q = empty_string;

   Operation Enqueue(alters E: Entry; updates Q: Queue);
      requires |Q| < Max_Length;
      ensures Q = #Q o <#E>;

   Operation Dequeue(replaces R: Entry; updates Q: Queue);
      requires |Q| /= 0;
      ensures #Q = <R> o Q;

   Operation Swap_First_Entry(updates E: Entry; updates Q: Queue);
      requires |Q| /= 0;
      ensures Q = <#E> o Prt_Btwn(1, |#Q|, #Q)
                    and  E = DeString(Prt_Btwn(0, 1, #Q));

   Operation Length(restores Q: Queue): Integer;
      ensures Length = (|Q|);

   Operation Rem_Capacity(restores Q: Queue): Integer;
      ensures Rem_Capacity = (Max_Length - |Q|);

   Operation Clear(clears Q: Queue);

end Queue_Template;
```

Figure 24. Queue Template Interface

RESOLVE Operation Specifications

RESOLVE has mechanisms to thoroughly specify every aspect of an operation. Formal specifications should be concise, unambiguous, non-redundant, expressed precisely in the formal notation, and RESOLVE achieves this goal. When a mathematical string is used to model, for example, a linear structure of a queue, a number of mathematical string operators (e.g., concatenation, substring, length, etc.) are employed

to describe its behavior. By using various mathematical theories, the specification language not only provides an unambiguous way to write pre- and post-conditions, but also makes it possible to specify parameter modes for all operations. The three operation specification components are:

- *requires clause* -- a precondition that should be true before an operation is called.
- *ensures clause* – a post-condition that will hold after the operation executes.
- *parameter modes* -- information about each incoming formal parameter that specifies what happens to its value after the operation completes.

RESOLVE Pre- and Post-Conditions

The *requires* clause is the responsibility of the caller, that is, the client that will be calling this operation in his code. The *ensures* clause is the responsibility of the operation implementer, that is, the one who writes the implementation. Below is an example of a RESOLVE operation that demonstrates how pre- and post-conditions work. It is an operation on the Integer data type modeled by Z.

```
Operation Increment (updates Num: Integer);
      requires: Num < Max_Int;
      ensures:  Num = #Num + 1;
```

This Increment operation takes an Integer Num as an incoming parameter, and increments it. Before this operation can be called in the user's code, the user must ensure that the *requires* clause is satisfied, i.e., that the *Num* is less than the maximum integer *Max_Int* allowed in this program. The e*nsures* clause guarantees that when the operation completes, the outgoing value of Num is the incoming value (#Num) incremented by one. The pound sign –#" indicates the incoming value of the variable. Implementation details

are not important here. Whether the implementer of this operation used bit shifting to increment the value, or just a simple mathematical addition will have no influence on the guarantees of the operation - the old value incremented.

RESOLVE Operation Parameter Modes

Parameter modes specify the effect of the operation on the value of the incoming parameters. There are seven parameter modes in RESOLVE, presented in Table 31. The right column explains what happens to the incoming variable after operation completes. As apparent from the discussion above, RESOLVE is very thorough in providing formal specifications. It takes care of operation parameter modes, pre-conditions, and post-conditions. The details of Queue operations are explained next.

The two operations typical to any Queue ADT are *Enqueue* and *Dequeue*. Operation *Enqueue* takes two incoming parameters - an *Entry E* and a *Queue Q* onto which the entry is placed. The parameter modes of *E* and *Q* are *alters* and *updates* respectively, which means that after *E* is enqueued, the *Q* is updated by having *E* added to the queue, and the final value of *E* is now altered in an unspecified way. It is up to the operation implementer to handle this in the manner they see fit.

The *Enqueue*'s pre-condition requires that the length of the existing *Q* is less than the maximum length of the Queue in order for an operation to be called. It ensures that at the end of the operation, the resulting *Q* contains the incoming Entry *E* appended to the old Queue. The # sign in the *ensures* clause indicates an incoming value of the parameter, and the angular brackets <> indicate a single value. So, *Q = #Q o <#E>* simply means

that the new value of the *Queue Q* is the incoming value of the queue *(#Q*) concatenated with an incoming value of Entry *E (<#E>)*.

alters x	the incoming value of x (#x) is used by the operation as specified in the ensures clause, and the outgoing value of x is an unspecified value of the same type;
clears *x*	the incoming value of *x* (#x) is used by the operation as specified in the ensures clause, and the outgoing value of *x* is reset to the initial value as defined in the template of that type;
replaces *x*	the incoming value of *x* (*#x*) is replaced with another value by the operation as specified in the ensures clause, and the outgoing value of *x* is that new value;
updates *x*	the outgoing value of *x* is an update of the input value as specified in the ensures clause;
evaluates *x*	the caller is expected to provide an expression of the type, and it is used by the operation as specified in the ensures clause;
preserves *x*	the outgoing value of *x* is the same at the incoming value (*#x*), the value is not changed during the execution of the operation;
restores *x*	the outgoing value of *x* is the same at the incoming value (*#x*), the value may be changed during the execution of the operation;

Table 31. Summary of operation parameter modes

Operation *Dequeue* takes two parameters - an *Entry R* which holds the dequeued Entry, and the *Queue Q* from which it is dequeued. As explained earlier, the incoming parameter's mode indicates that the value of *R* passed into the operation will be *replaced* by the dequeued value, while the Queue Q will be *updated* to its new state. The pre-condition to this operation requires that the queue has at least one element in it, and the

post-condition guarantees that the updated queue is the value of the old queue with the first element removed.

Some RESOLVE operations return values. Return value types follow the list of operation parameters, and are separated from it with a colon. This is demonstrated by the Queue Operations *Length* and *Rem_Capacity*. *Length* takes a Queue as an incoming parameter, and returns an Integer value of the length of the Queue. If during the operation execution the Queue was modified in any way, it will be restored to its original value after the operation completion, as indicated by the *restores* parameter mode. Length does not have a *requires* clause; it can be called at any time during the program execution and therefore does not need a pre-condition.

Rem_Capacity, as the name implies, returns the remaining capacity of the queue, that is, how many additional elements can be enqueued until the Queue is full. It does not have *requires* clause either, and it *ensures* that the value of the Queue after the operation completion is restored to its original value.

Operation *Clear* takes a *Queue Q* as an incoming parameter, and after the operation completes the Queue is returned to its initial state, defined as an *empty_string* in the initialization clause of the *Queue_Template*. Because the parameter mode *clears* specifies that the queue is returned to its initial state, the operation does not have an *ensures* clause.

Operation *Swap_First_Entry* swaps the first entry in the queue as the name indicates. The parameters are an Entry *E* that is updated to hold the swapped entry after the operation completes, and the queue Q that is updated to its new state. As in the case

with the *Dequeue*, this operation's pre-condition requires that queue contains at least one element to call this operation. The post-condition guarantees that the first entry is swapped, and specifically, that the first Entry of the Queue *Q* is now the incoming value of *E*, and the outgoing value of *E (#E)* is what used to be the first entry. In other words, if the incoming *Q* contains <1, 2, 3> and the incoming *E* is 7, then after the operation completes the *Q* is updated to contain <7, 2, 3>.

Specification Equivalence and Redundancy

There is frequently more than one way to write correct specifications using mathematical notation. The example below shows the *requires* clause of the Queue operation *Enqueue*:

```
Operation Enqueue(alters E: Entry; updates Q: Queue);
   requires |Q| < Max_Length;
   ensures Q = #Q o <#E>;
```

In order to perform an *Enqueue* on the *Q* the maximum length of the Queue should be less than *Max_Length*, the max number of elements allowed in the queue, which is known at time of creation. The same pre-condition can also be expressed in the following way:

```
requires |Q| <= Max_Length + 1;
```

Both the *requires* clauses have exactly the same meaning. They are equivalent, either of the two can be used with the same effect.

Another interesting example is that of an equivalent specification for the operation Swap_First_Entry. Consider the post-condition below:

```
ensures there exists Rem: Str(Entry) such that
          #Q = <E> o Rem and Q = <#E> o Rem;
```

Ptr_Btwn() string function has been used to specify this operation:

```
ensures  Q = <#E> o Prt_Btwn(1, |#Q|, #Q)
                and  E = DeString(Prt_Btwn(0, 1, #Q));
```

Prt_Btwn() is pronounced as ~~p~~art_between", and retrieves a substring located between the two specified positions of the original string. The position of the first element of the string is always 1, not 0. To retrieve the first element of the string the starting position of 0 (*after* which it will start retrieving the substring), and ending position of 1 (which is included in the retrieved substring) are specified. In this example, the resulting queue is the concatenation of the new *Entry E* with the substring from the incoming *Q* starting after position 1, and including the rest of the string, and the new *E* is the substring from the incoming *Q* that contains only the first element. Here, DeString is an operator that transforms a string containing a single entry to an entry.

The two specifications are equivalent. Notice that the second specification does not have an existential clause and avoids declaring a temporary variable *Rem*. Therefore, it is not only easier to understand, but is also easier to prove.

Specifications can also be redundant, as shown in the *ensures* clause of the Queue operation *Enqueue* below.

```
ensures Q = #Q o <#E> and |Q| = |#Q| + 1;
```

The clause states that the new queue is now the incoming element *#E* concatenated with the contents of the initial queue *#Q*, which implies that the length of the new Queue is now one longer. Therefore, the conjunct "*|Q| = |#Q| + 1*" is redundant and unnecessary. Redundancy is undesirable, as it violates the requirement that formal specifications should be non-redundant.

Each template file created in RESOLVE should be saved in a concept file with the file extension ".co" with the file name the same as the name of the concept. In this example, our *Queue_Template.co* informs the compiler and users what the file contains.

RESOLVE Concept Realizations

Concepts do not define how operations are implemented. They only provide specifications in the forms of pre-, post-conditions, parameter modes, initializations, etc. Implementation is done in Concept *Realization* files. Just like in any modern object-oriented programming language, the concept can have one or more implementations. Developers match their implementation to the specific goal they are trying to achieve – simplicity, performance or convenience. In the case of Queue_Template, a realization *Circular_Array_Realization* efficiently enqueues and dequeues elements in a circular fashion, and a space-conscious *Clean_Array_Realization* performs certain manipulations to ensure the dequeued element of an array has been cleared. *Circular_Array Realization* is shown in Figure 25**,** and is detailed next.

The first line of the program declares the name of the realization: *Circular_Array_Realiz* and states that it implements *Queue_Template*. Please note that

the name of the file that contains this realization should be the same as the name of the Realization, with a file extension .rb, such as *Circular_Array_Realiz.rb*. This is the RESOLVE convention and is true for all other types of files, with the exception of the file extensions, which are different for each type of a file. The newly created Type Queue is a *Record* which consists of an *Array Contents* that holds variables of type *Entry* and indices ranging from 0 to *Max_Length* -1, and Integers *Front* and *Length*. *Record* in RESOLVE is similar to a *struct* in other programming languages. The rest of the important concepts are explained in the next subsection.

Internal Contracts

Internal contracts reflect the relationship between internal routines in a component, and specifically deal with the consistency of the internal representation of the data abstraction itself. For example, both the *Enqueue* and *Dequeue* should be implemented such that every time they are executed, the expected item is returned. To ensure this, both the operations should be consistent in their internal implementation.

In the case when an array is used to represent a *Queue*, two distinct implementations are possible. An item can either be placed into the beginning indices of an array in *Enqueue* and is retrieved from the back in the *Dequeue* operation; or it can be placed at the back of the array, and removed from the front in the *Dequeue*. The challenge is to ensure that the two operations *Dequeue* and *Enqueue* work consistently even if developed independently. This is guaranteed by the internal contracts, *conventions* and *correspondence*.

```
Realization Circular_Array_Realiz for Queue_Template;
   Type Queue = Record
      Contents:  Array 0..Max_Length - 1 of Entry;
                 Front, Length: Integer;
   end;

   convention
   0 <= Q.Front < Max_Length  and  0 <= Q.Length <= Max_Length;

   correspondence
   Conc.Q = (Concatenation i: Integer
      where Q.Front <= i <= Q.Front + Q.Length- 1,
      <Q.Contents(i mod Max_Length)>);

   Procedure Enqueue(alters E: Entry; updates Q: Queue);
      Q.Contents[(Q.Front + Q.Length) mod Max_Length] :=: E;
      Q.Length := Q.Length + 1;
   end Enqueue;

   Procedure Dequeue(replaces R: Entry; updates Q: Queue);
      Q.Contents[Q.Front] :=: R;
      Q.Front := (Q.Front + 1) mod Max_Length;
      Q.Length := Q.Length -1;
   end Dequeue;

   Procedure Swap_First_Entry(updates E: Entry;
                              updates Q: Queue);
      Q.Contents[Q.Front] :=: E;
   end Swap_First_Entry;

   Procedure Length(restores Q: Queue): Integer;
      Length := Q.Length;
   end Length;

   Procedure Rem_Capacity(restores Q: Queue): Integer;
      Rem_Capacity := Max_Length - Q.Length;
   end Rem_Capacity;

   Procedure Clear(clears Q: Queue);
      Q.Front := 0; Q.Length := 0;
   end Clear;
end Circular_Array_Realiz;
```

Figure 25. Circular Array Realization for the Queue Template

Conventions are also called *representation invariants*. Conventions are assumptions that are true at the beginning and end of every procedure (except the

beginning of initialization and end of finalization). Every procedure must guarantee that the convention holds after the procedure completes. The convention contains information needed to keep all the operations' implementations consistent. In this Queue implementation the convention is:

```
convention
   0 <= Q.Front < Max_Length and
   0 <= Q.Length <= Max_Length;
```

The front of the Q is always between zero and Max_Length, and Q Length is between zero and Max_Length. Every operation should leave the variables in a state that satisfies conventions.

In the implementation, conventions are usually written before correspondence because the correspondence only needs to be interpreted for the representations that satisfy the conventions. Correspondence is an abstraction function or abstraction relation. In the concept, a Queue is represented as a mathematical string of entries, and in the implementation it is a Record with an array and an Integer. There is a correspondence between the abstract view of the queue presented in the specifications and its representation in the implementation. Without correspondence it is impossible to reason about the correctness of the implementation with respect to specification.

Correspondence explains how to relate the values from the *Content* array to the mathematical string representing it. In this implementation the conceptual string *Conc.Q* corresponds to the string formed by inserting the new item into the array at the location calculated using the modulus arithmetic.

```
Conc.Q = (Concatenation i: Integer
   where Q.Front <= i <= Q.Front + Q.Length - 1,
         <Q.Contents(i mod Max_Length)>);
```

This correspondence is used to check if the code for the *Enqueue* and *Dequeue* procedures based on our circular array representation of a queue, satisfies the specification based on the string models of the queue in the template. To function correctly, both the *Enqueue* and *Dequeue* have to agree to a common correspondence.

Implementing Procedures

When operations defined in a template are implemented, the keyword *Procedure* is used. The procedure defines what actions are performed. The discussion on Queue procedures is preceded by a brief overview of some uncommon RESOLVE operators.

RESOLVE Operators

The operator "~:=:" (previously seen in the implementation of the Procedure *Enqueue)* is a swap operator. This built-in capability swaps the values of two variables of the same type, without the necessity of creating any temporary hold variables. Not declared in any template file, this unique feature of RESOLVE is automatically available for use with any built-in or user-created data type. An assignment operator "~:=" can be used only to assign the return value of a function to a variable.

Queue Procedures

Procedure *Enqueue* updates the *Q* by placing the new entry *E* to a location in the *Contents* array. Because our implementation is circular, the index of the new location is calculated based on where the *Q* front is, how many elements in the *Queue*, and the *Max_Length* of the *Queue.* The parameter mode of E is *alteres*, and as seen in the implementation, the value of *E* is swapped with the value of that array location. Because the value at the location that *E* is swapped with is unknown, the value of *E* is altered in an unspecified way.

Procedure *Dequeue* swaps the front element of the Queue with the Entry *R*, adjusts the *Q.Front* , and updates the queue's length. *Swap_First_Entry* replaces the first entry of the *Q* with a new element *E*. Procedures *Length* and *Rem_Capacity* return the length of *Q* and calculate its remaining capacity, correspondingly. Procedure *Clear* sets the length of the queue and *Q.Front* to zeros to indicate that the Q is now empty.

RESOLVE Enhancements

Data abstraction concepts contain only the number of basic orthogonal operations necessary to ensure the proper functionality of the data structure. Enhancements provide additional functionality to the existing data abstractions by creating new operations not defined in the concept by combining existing operations. As the name indicates, they enhance that data abstraction with a custom feature. For example, though Queue_Template does not contain an operation Append that allows one to append a queue to another, it can be implemented as an enhancement. Enhancement contains

specification for one or more new operations. It supports the principle of reusability, allowing it to reuse the already existing template, but at the same time enhancing it with the special operations we need. Figure 26 shows the *Append_Capability* enhancement for Queue_Template.

```
Enhancement Append_Capability for Queue_Template;
   Operation Append(updates P: Queue; clears Q: Queue);
      requires |P| + |Q| <= Max_Length;
      ensures P = #P o #Q;
end Append_Capability;
```

Figure 26. Queue Enhancement

The second parameter *Q* is appended to the first queues *P*, clearing Queue *Q* and updating queue *P*. The *requires* clause states that the combined lengths of the two queues should be less than the maximum length of a queue. The *ensures* clause guarantees that the outgoing *P* is the contents of the incoming *P* concatenated with the incoming *Q*. This enhancement resides in the file *Append_Capability.en* and is later implemented by an enhancement realization.

RESOLVE Enhancement Realizations

Enhancements are realized by enhancement realizations, the same way concepts are realized by concept realizations. And similar to concept realizations, they can have several implementations. Enhancements support the principle of information hiding, and encourage component reuse which is so important to software engineering. Figure 27

contains a recursive implementation of the *Append* operation. This realization file is named *Recursive_Append_Realization.rb*.

Procedure *Append* is identified by a keyword *recursive*. The first statement indicates that the *Length* of *Q* will be decreased each recursive procedure call. This is a tail-recursive Procedure. As along as the condition remains true, an element is dequeued from *Q* and enqueued onto *P* before making another call to *Append*.

```
Realization Recursive_Append_Realiz for Append_Capability of
                                                Queue_Template;
   uses Std_Boolean_Fac;
   Recursive Procedure Append (updates P: Queue; clears Q: Queue);
      decreasing |Q|;
      Var E: Entry;
      If (Length (Q) /= 0) then
         Dequeue(E,Q);
         Enqueue(E,P);
         Append(P,Q);
      end;
   end Append;
end Recursive_Append_Realiz;
```

Figure 27. Recursive Implementation for Append Enhancement

Another implementation of the same enhancement is an iterative realization shown in Figure 28. *Iterative Implementation for Append Enhancement*does not use recursion. This realization uses a while loop to perform a series of *Dequeues* and *Enqueues.* The while loop uses invariants described in the next subsection.

```
Realization Iterative_Append_Realiz for Append_Capability of
                                        Queue_Template;
   uses Std_Boolean_Fac;
   Procedure Append(updates P: Queue; clears Q: Queue);
      Var E: Entry;
      While (Length(Q) /= 0)
         changing P,Q,E;
         maintaining (P o Q = #P o #Q)
                     and (|P| + |Q| <= Max_Length);
         decreasing |Q|;
         do
            Dequeue(E,Q);
            Enqueue(E,P);
      end;
   end Append;
end Iterative_Append_Realiz;
```

Figure 28. Iterative Implementation for Append Enhancement

Loop invariants

A loop invariant is an invariant used to describe a property of loops. It is also a condition that is necessarily true *before* and *after* each iteration of a loop. The above enhancement implementation uses loop invariants. Certain annotations are required for loops. For example, the proper syntax of a While-do loop is shown below:

```
While (condition)
   changing … ;
   maintaining … ;
   decreasing … ;
do statements;
```

The context of the above example comes from a procedure d*equeueing* an element from *Q* and *enqueueing* it onto the *P*, as long as the *Length* of the *Q* is greater than zero. In each iteration, *P*, *Q* and *E* are modified. In every iteration, the concatenation of *P* and *Q* is the same as the contents of the initial queues concatenated. This is the loop invariant and it is is specified in the *maintaining* clause of a loop. In each loop iteration, the length

of the queue Q is *decreasing* and this is the termination progress metric. The three clauses (changing, maintaining, decreasing) provide a clear specification of a loop and they are used in automated verification.

RESOLVE Facility

A facility may be created from scratch, for example, to present a Main program. A (short) facility may also be created by instantiating a template by passing parameters to it (if any), and indicating which realization is used to implement it. If the data abstraction is enhanced, the facility declaration also indicates which realization implements it.

Figure 29 shows a simple program using the Queue template. The *facility*, named QF, instantiates parameterized Queue_Template. *QF* contains Integers and has a *Max_Length* of 6. Realized by a *Circular_Array Realization*, it uses an *Append_Enhancement* realized by *Recursive_Append_ Realization*. The body of work is done in the operation *Main*. Two Integer variables *C* and *D* are created and initialized, and *enqueued* onto the two queues *Q1* and *Q2*. *Q2* is then appended to *Q1*. Two values *C* and *D* are *dequeue*d from the *Q1* and printed.

Using an *Iterative_Array_Realization*, for example, instead of *Recursive_Array_Realization* does not require user to make any program changes, the execution produces the same correct results. This is true as long as both of the realizations adhere to formal specifications.

```
Facility Queue_Append_Example_Facility;
      uses Std_Boolean_Fac, Std_Integer_Fac, Queue_Template;

      Facility QF is Queue_Template(Integer, 6)
         realized by Circular_Array_Realiz
         enhanced by Append_Capability
         realized by Recursive_Append_Realization;

         Operation Main();
         Procedure
            -- Declare variables
            Var C, D: Integer;
            Var Q1, Q2: QF.Queue;

            -- Initialize Integer variables
            C := 15;   D := 17;

            -- Enqueue Integers onto the two queues
            Enqueue(C, Q1);
            Enqueue(D, Q2);

            -- Append the Queues
            Append(Q1, Q2);

            -- Dequeue variables from Q1 and print them
            Dequeue(C, Q1);
            Dequeue(D, Q1);

            -- Print the values od C and D
            Write(C);
            Write(D);
         end Main;
   end Queue_Append_Example_Facility;
```

Figure 29. User's program using the Queue Template

RESOLVE Program Structure as a CRD

The component relationship diagram (CRD) in Figure 30 depicts relationship among the components discussed in this Appendix.

Queue_Template is the central building block in the CRD. The two implementations (*Circular_Array_Realization* and *Clean_Array_Realization*) are shown with the ~~–i~~mplements" arrows. The *Append* enhancement ~~–e~~nhances" the Queue's

functionality, and is implemented by two enhancement realizations. User's *Facility* uses the *Queue_Template*.

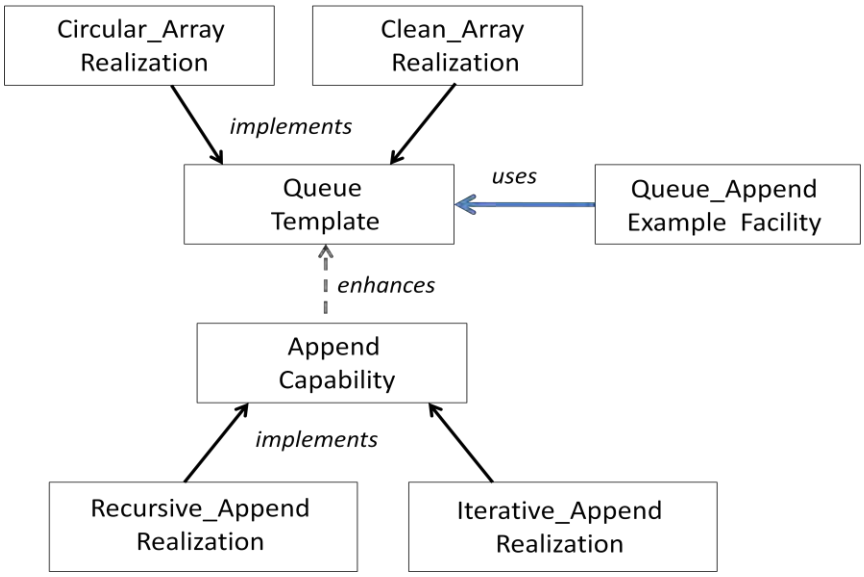


Figure 30. Sample RESOLVE Component Relationship Diagram

RESOLVE Platform Compatibility

RESOLVE is platform-independent. "Behind the scenes" the code is translated to Java code, compiled and interpreted as any Java class files. RESOLVE can be installed on a Windows machine as well as on Linux.

RESOLVE Suitability for Teaching Mathematical Reasoning

RESOLVE has a lot of promise for software engineering and it has a growing community of users [23]. However, the RESOLVE is simply one medium for teaching general mathematical reasoning skills. These skills are not dependent on this environment

and can easily translate to any programming environment that college or university may chose. The benefits of using RESOLVE for our purpose include the presence of the built-in verifier and a native built-in specification language. Instead of trying to master a separate verifier and a separate specification language, students can concentrate on learning the mathematical reasoning skills. Should a college or an institution select RESOLVE as the teaching medium, we offer a Web IDE (discussed in Chapter 3) that speeds up the learning curve by providing a user-friendly interface with on-demand help. It can be accessed on our project home page at http://resolve.cs.clemson.edu/interface.

Appendix D

Experimental Data

Each data table provides several kinds of important information. The leftmost column is the relevant RCI item that was assessed. The second column shows total number of points possible to earn on that particular question, followed by the class average on this question, number of students that scored >= 70% on that question, and the percentage of students who scored >= 70% on that question. We have chosen 70% as the cut-off point, because 70% is the lowest passing grade. The data is separated by semester, course number, and instructor (where applicable).

Spring 2008 Assessment Data

Spring 2008 CPSC215 Final Exam, Clemson University, 13 students

Reasoning Topic	Total Points	Class Avg	Percent	#students >=70	%students >=70
RCI#3.4.3 (2, 3)	3	3.0	100%	13	100%
RCI#3.4.3.2	4	2.9	72%	8	62%
RCI#3.4.3.3	8	5.2	64%	7	54%

Spring 2009 Assessment Data

Spring 2009 CPSC215 Final Exam, Clemson University, 12 students

Reasoning Topic	Total Points	Class Avg	Percent	#students >=70	%students >=70
RCI#3.4.3(2, 3)	3	2.9	96%	11	92%
RCI#3.4.3.2	12	6.9	58%	5	42%
RCI#4.1.1.3	2	1.5	75%	9	75%
RCI#4.2.1.1	2	1.7	83%	10	83%
RCI#5.2.2.1	2	1.8	92%	11	92%
RCI#5.2.2 (1, 2)	14	11.7	84%	10	83%
RCI#5.3.2	4	0.4	10%	1	8%

Fall 2010 Assessment Data

Fall 2010 CPSC372 Quiz11, Clemson University, 17 students

Reasoning Topic	Total Points	Class Avg	Percent	#students >=70	%students >=70
RCI#3.3.1.4	3	2.1	69%	8	47%
RCI#3.3.1.4	4	3.5	88%	15	88%
RCI#3.3.1.5	3	2.6	86%	13	76%

Fall 2010 CPSC372 Quiz13, Clemson University, 18 students

Reasoning Topic	Total Points	Class Avg	Percent	#students >=70	%students >=70
RCI#3.4.2.2	5	4.6	92%	17	94%
RCI#3.4.2.2	5	4.4	89%	16	89%

Fall 2010 CPSC372 Final Exam, Clemson University, 15 students

Reasoning Topic	Total Points	Class Avg	Percent	#students >=70	%students >=70
RCI#3.4 (2, 3)	3	2.8	94%	14	93%
RCI#4.1.1.2	6	5.6	93%	14	93%
RCI#5.2.2 (1, 2)	5	3.1	61%	6	40%
RCI#5.3 (1, 2)	6	5.4	91%	13	87%

Spring 2011Assessment Data

Spring 2011 CPSC215 Final Exam, Clemson University, 25 students

Reasoning Topic	Total Points	Class Avg	Percent	#students >=70	%students >=70
RCI#3.4.3 (2, 3)	3	2.9	96%	24	96%
RCI#3.4.3.2	12	7.2	60%	10	40%
RCI#4.1.1.3	2	1.6	80%	20	80%
RCI#4.2.1.1	2	1.6	80%	20	80%
RCI#5.2.2.1	2	1.8	88%	22	88%
RCI#5.2.2(1, 2)	14	9.6	68%	14	56%
RCI#5.3.2	4	1.3	34%	7	28%

Spring 2011 CS315 Assignment, Alabama University, 18 students

Reasoning Topic	Total Points	Class Avg	Percent	#students >=70	%students >=70
RCI#5.2.2	15	10.7	71%	7	39%
RCI#5.3	15	14.2	94%	17	94%
RCI#5.3	10	9.4	94%	17	94%
RCI#5.3	30	21.9	73%	13	72%
RCI#5.3	15	9.2	61%	11	61%
RCI#5.3	15	13.0	87%	16	89%

Spring 2011 CPSC372 Final Exam, Clemson University, 15 students

Reasoning Topic	Total Points	Class Avg	Percent	#students >=70	%students >=70
RCI#3.4.3 (2, 3)	4	3.1	78%	10	67%
RCI#4.1.1.2	6	4.8	79%	11	73%
RCI#5.2.2 (1, 2)	6	4.4	73%	9	60%
RCI#5.3 (1, 2)	4	3.5	88%	14	93%

Fall 2011 Assessment Data

Fall 2011 CPSC215 Final Exam, Clemson University, 24 students, Instructor 1

Reasoning Topic	Total Points	Class Avg	Percent	#students >=70	%students >=70
RCI#3.3.1.5	5	3.4	68%	15	63%
RCI#3.4.3 (2,3)	4	3.2	80%	18	75%
RCI#3.4.3	8	5.3	66%	12	50%
RCI#4.1.1	4	3.9	98%	23	96%
RCI#5.2.2	10	6.3	63%	11	46%

Fall 2011 CPSC215 Final Exam, Clemson University, 14 students, Instructor 2

Reasoning Topic	Total Points	Class Avg	Percent	#students >=70	%students >=70
RCI#3.3.1.5	2	0.6	29%	2	14%
RCI#3.4.3.2	2	0.6	29%	4	29%
RCI#3.4.3 (2,3)	2	1.7	86%	11	79%
RCI#3.4.3 (2, 3)	4	0.1	4%	0	0%
RCI#4.1.1.3	2	1.1	57%	5	36%
RCI#5.2.2	8	5.1	64%	7	50%

Fall 2011 CS315 Final Exam, Alabama University, 15 students

Reasoning Topic	Total Points	Class Avg	Percent	#students >=70	%students >=70
RCI#3.4.3 (2, 3)	3	2.5	82%	12	80%
RCI#5.2.2	7	4.9	70%	10	67%

Fall 2011 CS315 Assignment, Alabama University, 14 students

Reasoning Topic	Total Points	Class Avg	Percent	#students >=70	%students >=70
RCI#5.2.2	15	14.1	94%	14	100%
RCI#5.3	15	11.8	79%	11	79%
RCI#5.3	10	8.9	89%	12	86%
RCI#5.3	30	24.3	81%	10	71%
RCI#5.3	15	12.9	86%	12	86%
RCI#5.3	15	12.3	82%	11	79%

Fall 2011 CPSC372, Quiz8, Clemson University, 34 students

Reasoning Topic	Total Points	Class Avg	Percent	#students >=70	%students >=70
RCI#3.4	3	2.1	69%	12	35%
RCI#4.2.3	3	5.7	82%	27	79%

Fall 2011 CPSC372, Quiz9, Clemson University, 28 students

Reasoning Topic	Total Points	Class Avg	Percent	#students >=70	%students >=70
RCI#3.4	5	3.1	63%	15	54%
RCI#4.2.3	5	4.4	87%	23	82%

Fall 2011 CPSC372, Quiz10, Clemson University, 31 students

Reasoning Topic	Total Points	Class Avg	Percent	#students >=70	%students >=70
RCI#3.4	3	2.0	68%	17	55%
RCI#4.1.1	3-	2.4	78%	11	35%
RCI#4.2.3	4	3.2	80%	22	71%

Fall 2011 CPSC372, Quiz11, Clemson University, 29 students

Reasoning Topic	Total Points	Class Avg	Percent	#students >=70	%students >=70
RCI#5.2.2	5	4.2	83%	27	93%
RCI#5.3	5	4.2	84%	22	76%

Fall 2011 CPSC372 Final Exam, Clemson University, 33 students

Reasoning Topic	Total Points	Class Avg	Percent	#students >=70	%students >=70
RCI#3.2.4	1	0.8	81%	26	81%
RCI#3.3.1	1	0.8	84%	27	84%
RCI#3.4.3 (2, 3)	4	3.6	89%	28	88%
RCI#3.4.3	3	2.7	89%	26	81%
RCI#4.1.1.2	8	6.9	86%	28	88%
RCI#4.1.2	1	0.6	56%	18	56%
RCI#4.1.2	1	0.9	94%	30	94%
RCI#4.2	4	2.6	64%	20	63%
RCI#4.2	4	2.9	73%	18	56%
RCI#4.2	4	3.4	84%	27	84%
RCI#5.2.2	5	3.8	76%	24	75%
RCI#5.3 (1, 2)	4	1.8	46%	9	28%

Spring 2012 Assessment Data

Spring 2012 CPSC215 Quiz1, Clemson University, 20 students, Instructor 1

Reasoning Topic	Total Points	Class Avg	Percent	#students >=70	%students >=70
RCI# 3.1.2	2	1.7	85%	14	70%
RCI# 3.3.1	4	2.2	55%	7	35%
RCI# 3.3.3	2	1.3	65%	12	60%
RCI# 3.4.3	2	1.6	80%	15	75%
RCI# 3.4.3	2	2.0	100%	20	100%
RCI# 3.4.3	2	1.6	78%	15	75%
RCI# 4.1.1.3	1	0.9	90%	18	90%
RCI# 4.1.1.3	1	0.9	90%	18	90%
RCI# 4.2	2	1.5	75%	14	70%

Spring 2012 CPSC215 Quiz2, Clemson University, 20 students, Instructor 1

Reasoning Topic	Total Points	Class Avg	Percent	#students >=70	%students >=70
RCI# 3.3	1	0.6	55%	11	55%
RCI# 3.4.3.6	1	0.9	90%	18	90%
RCI# 4.1.2	1	0.7	70%	14	70%
RCI# 5.3	2	1.4	68%	10	50%

Spring 2012 CPSC215 Final Exam, Clemson University, 21 students, Instructor 1

Reasoning Topic	Total Points	Class Avg	Percent	#students >=70	%students >=70
RCI# 3.3.1.5	5	2.5	50%	10	48%
RCI# 3.4.3 (1, 2)	5	3.9	77%	11	52%
RCI# 3.4.3 (1, 2)	2	1.6	81%	17	81%
RCI# 4.1.1.3	2	1.4	71%	15	71%
RCI# 4.2	2	2.0	100%	21	100%
RCI# 4.3.2.1	2	2.0	100%	21	100%
RCI# 5.2 (1, 2)	5	2.9	57%	12	57%
RCI# 5.2.2	5	3.0	60%	9	43%
RCI# 5.2.2	2	1.9	95%	20	95%
RCI# 5.3	15	6.4	43%	7	33%
RCI# 5.3.1	1	0.8	76%	16	76%

Spring 2012 CPSC215 Final Exam, Clemson University, 21 students, Instructor 2

Reasoning Topic	Total Points	Class Avg	Percent	#students >=70	%students >=70
RCI#3.3.1	2	0.7	33%	2	10%
RCI#3.4.3	2	1.8	90%	18	86%
RCI#3.4.3.2	2	0.9	43%	9	43%
RCI#3.4.3.3	2	1.8	88%	18	86%
RCI#4.1.1.2	2	1.4	71%	15	71%
RCI#4.1.1.2	2	1.4	71%	9	43%
RCI#4.1.1.3	2	1.4	71%	10	48%
RCI#3.4.3	2	1.5	76%	16	76%
RCI#5.2.2	6	4.8	79%	15	71%

Spring 2012 CS315 Final Exam, Alabama University, 41 students

Reasoning Topic	Total Points	Class Avg	Percent	#students >=70	%students >=70
RCI#3.4.3 (2, 3)	3	2.5	82%	31	76%
RCI#5.2.2	7	3.9	56%	19	46%

Spring 2012 CPSC372, Midterm Exam, Clemson University, 24 students

Reasoning Topic	Total Points	Class Avg	Percent	#students >=70	%students >=70
RCI#3.4 (2, 3)	4	2.4	59%	8	33%
RCI#3.4 (2, 3)	1	0.8	79%	19	79%
RCI#3.4.3	2	0.8	40%	6	25%
RCI#3.4.3	2	1.9	96%	23	96%
RCI#3.4.3.1	2	1.5	75%	13	54%
RCI#3.4.3.3	2	1.3	67%	11	46%
RCI#4.1.2	1	0.8	75%	18	75%
RCI#4.2.1	4	2.7	67%	10	42%
RCI#4.2.3.1	3	1.7	58%	3	13%
RCI#4.2.3.2	5	3.9	78%	15	63%
RCI#4.2.3.2	1	0.9	88%	21	88%
RCI#4.3.1	1	0.7	71%	17	71%
RCI#4.3.1 (1, 2)	2	1.1	56%	9	38%
RCI#4.3.1 (1, 2)	10	7.4	74%	17	71%

Spring 2012 CPSC372, Final Exam, Clemson University, 23 students

Reasoning Topic	Total Points	Class Avg	Percent	#students >=70	%students >=70
RCI#3.4.3	4	3.4	84%	18	90%
RCI#4.1.1.2	6	5.1	84%	12	60%
RCI#5.2.2	4	2.9	74%	11	55%
RCI#5.3 (1, 2)	4	3.5	86%	18	90%

Fall 2012 Assessment Data

Fall 2012 CPSC215 Final Exam, Clemson University, 21 students, Instructor 1

Reasoning Topic	Total Points	Class Avg	Percent	#students >=70	%students >=70
RCI#3.4.3	4	1.9	48%	8	38%
RCI#3.4.3.3	12	5.3	44%	4	19%
RCI#3.4.3 (2, 3)	3	3.0	96%	21	100%
RCI#4.1.1.3	2	1.4	71%	15	71%
RCI#4.2.1.1	2	1.9	95%	20	95%
RCI#5.2.2.1	2	1.6	81%	17	81%
RCI#5.2.2 (1, 2)	14	11.4	81%	17	81%
RCI#5.3.2	4	2.8	69%	12	57%

Fall 2012 CPSC215 Final Exam, Clemson University, 16 students, Instructor 2

Reasoning Topic	Total Points	Class Avg	Percent	#students >=70	%students >=70
RCI#3.4.3.3	12	7.2	60%	6	38%
RCI#4.1.1.3	2	1.5	75%	12	75%
RCI#4.2.1.1	2	1.8	88%	14	88%
RCI#5.2.2.1	2	2.0	100%	16	100%
RCI#5.2.2 (1, 2)	14	11.9	85%	14	55%
RCI#5.3.2	4	2.5	63%	9	56%

Fall 2012 CPSC372 Final Exam, Clemson University, 17 students, Instructor 2

Reasoning Topic	Total Points	Class Avg	Percent	#students >=70	%students >=70
RCI#4.1.1.3	8	5.9	74%	10	59%

Appendix E

Student Consent Form and Survey Questions

Information Concerning Participation in a Clemson-Sponsored Research Study

Overview. You are invited to participate in a research study being conducted by Dr. Jason Hallstrom, Dr. Joseph Hollingsworth with Indiana University, Dr. Joan Krone with Denison University, Dr. Richard Pak, and Dr. Murali Sitaraman. The purpose of this study is to evaluate the teaching and learning benefits of new curriculum materials and software tools that will be introduced as part of your course this semester. If you agree to participate, we will ask you to complete a short survey that will help us understand your attitudes toward Computer Science and related topics. The amount of time required for you to complete the survey is less than fifteen minutes. We will ask you to complete a similar survey at the end of the semester. We will also collect information about your performance on homework assignments, quizzes, and tests. We may also use information about your opinions collected during informal conversations.
Benefits. Participation in this study has the potential to improve Computer Science curriculum across the country. There are no known risks or discomforts associated with this study.

Confidentiality. We will do everything we can to protect your privacy. No identifying information will be associated with the data we collect as part of this study; all data will be stored anonymously. Your identity will never be revealed in any publication or presentation resulting as part of this study.

Participation. Participation in this research study is completely voluntary. You may choose not to participate, and may withdraw your consent to participate at any time. You will not be rewarded for participating. You will not be punished if you decide not to participate.

Contacts. If you have any questions or concerns about this study, or if any problems arise, please contact Dr. Hallstrom at 864.656.0187. If you have any questions or concerns about your rights as a research participant, please contact the Clemson University Office of Research Compliance at 864.656.6460.

Consent

I have read this consent form and have been given the opportunity to ask questions. I give my consent to participate in this study.

Participant's Age:_____

Participant's Signature:_____Date:_____

A copy of this informational letter will be given to you.

Collaborative Reasoning Survey

Part I.
Please provide short answers to each of the following questions. The questions ask you to describe your *opinions*. There are no right or wrong answers.

1. In your opinion, what are the most important characteristics of a software system deemed to be of ***high quality***? Feel free to explain each of the characteristics you identify.

2. What would you do differently or what steps would you take if the quality of your software was a critical requirement, unlike a typical lab assignment?

3. List up to three courses (or course numbers) that have best prepared you to develop high quality software. You may feel free to include even non-CS courses, if any, in this list.

1.
2.
3.

Part II.
Please rate your level of agreement with each of the following statements. You will be asked to select from six options: *strongly disagree, disagree, moderately disagree, moderately agree, agree, strongly agree*. The statements are a matter of opinion. There are no right or wrong answers.

4. If I worked for a company and was asked to develop 10,000 lines of software to solve a problem, I would be capable of designing and implementing that software.

strongly disagree	disagree	moderately disagree	moderately agree	agree	strongly agree
○	○	○	○	○	○

5. My coursework has prepared me well for a software development career.

strongly disagree	disagree	moderately disagree	moderately agree	agree	strongly agree
○	○	○	○	○	○

6. 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.

strongly disagree	disagree	moderately disagree	moderately agree	agree	strongly agree
○	○	○	○	○	○

7. 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 the programming language in which it is written.

strongly disagree	disagree	moderately disagree	moderately agree	agree	strongly agree
○	○	○	○	○	○

8. Software development can benefit from carefully designing each component before coding it, as opposed to quickly coding and experimenting with it.

strongly disagree	disagree	moderately disagree	moderately agree	agree	strongly agree
○	○	○	○	○	○

9. The main challenge of software construction lies in turning the design into code, and not so much in specifying what needs to be done.

strongly disagree	disagree	moderately disagree	moderately agree	agree	strongly agree
○	○	○	○	○	○

10. There are benefits to showing that a software component is correct without running it on a computer.

strongly disagree	disagree	moderately disagree	moderately agree	agree	strongly agree
○	○	○	○	○	○

11. It is possible to show that a software component is correct without actually running it on the computer.

strongly disagree	disagree	moderately disagree	moderately agree	agree	strongly agree
○	○	○	○	○	○

12. The development of reliable software has a lot to do with mathematical reasoning.

strongly disagree	disagree	moderately disagree	moderately agree	agree	strongly agree
○	○	○	○	○	○

13. To understand and reason about a program built using a component, you need to understand all the statements inside the component.

strongly disagree	disagree	moderately disagree	moderately agree	agree	strongly agree
○	○	○	○	○	○

14. Testing software thoroughly is the most important way to ensure software correctness.

strongly disagree	disagree	moderately disagree	moderately agree	agree	strongly agree
○	○	○	○	○	○

15. It is easy to combine components from different team members and produce working software.

strongly disagree	disagree	moderately disagree	moderately agree	agree	strongly agree
○	○	○	○	○	○

16. I believe that there is a strong correlation between a person's mathematical background and their ability to design and implement large systems correctly.

strongly disagree	disagree	moderately disagree	moderately agree	agree	strongly agree
○	○	○	○	○	○

17. To guarantee correctness, it is best to develop a system from scratch.

strongly disagree	disagree	moderately disagree	moderately agree	agree	strongly agree
○	○	○	○	○	○

18. Having precise mathematical descriptions for each software component improves the likelihood that my software will be correct.

strongly disagree	disagree	moderately disagree	moderately agree	agree	strongly agree
○	○	○	○	○	○

19. When working in teams, natural language (e.g., English) descriptions of the different components are sufficient for communication among team members.

strongly disagree	disagree	moderately disagree	moderately agree	agree	strongly agree
○	○	○	○	○	○

20. Before I run my code on a computer, I make it a practice to hand trace through the statements on example inputs to see if it works.

strongly disagree	disagree	moderately disagree	moderately agree	agree	strongly agree
○	○	○	○	○	○

21. Reasoning about programs involving components requires a thorough understanding of pointers and/or references.

strongly disagree	disagree	moderately disagree	moderately agree	agree	strongly agree
○	○	○	○	○	○

22. Working with a friend makes it easier to reason about a program.

strongly disagree	disagree	moderately disagree	moderately agree	agree	strongly agree
○	○	○	○	○	○

23. Working with a friend to complete classroom activities is fun.

strongly disagree	disagree	moderately disagree	moderately agree	agree	strongly agree
○	○	○	○	○	○

24. Reasoning about programs is easier when there is tool support.

strongly disagree	disagree	moderately disagree	moderately agree	agree	strongly agree
○	○	○	○	○	○

25. Tools make program analysis exercises interesting.

strongly disagree	disagree	moderately disagree	moderately agree	agree	strongly agree
○	○	○	○	○	○

26. My conception of what software is has changed significantly over time.

strongly disagree	disagree	moderately disagree	moderately agree	agree	strongly agree
○	○	○	○	○	○

27. My conception of the difficulty associated with developing high quality software has changed significantly over time.

strongly disagree	disagree	moderately disagree	moderately agree	agree	strongly agree
○	○	○	○	○	○

28. My conception of how to build high quality software has changed significantly over time.

strongly disagree	disagree	moderately disagree	moderately agree	agree	strongly agree
○	○	○	○	○	○

Part III
For each block of code, please choose the answer that best describes its functionality.

29. What is the effect of this block of code?

```
I = I + J;
J = I - J;
I = I - J;
```

- ◉ Assigns the value 0 to either I or J
- ○ Makes the values of I and J the same
- ○ Exchanges the values of I and J as long as J is positive
- ○ Exchanges the values of I and J under most (but not all) circumstances
- ○ Always exchanges the values of I and J

30. Please describe what this block of code does to the queue Q, a queue of Integers

```
Integer temp;
temp = Q.dequeue();
Q.enqueue(temp);
```

- Q is not changed
- Q is reversed
- Q is reversed if Q contains 1 or 2 Integers
- Q is reversed if Q contains no more than 2 Integers
- I'm not familiar with these queue operations

31. Please describe what this block of code does to the queue Q, a queue of Integers

```
Integer temp;
Integer count = Q.length();
while(count > 1) {
     temp = Q.dequeue();
     Q.enqueue(temp);
     count = count - 1;
}
```

- Q is not changed
- Q is reversed
- The last Integer in Q becomes the first
- Q is reversed if Q is not empty
- I'm not familiar with these queue operations

Appendix F

CPSC215 and CPSC372 Attitudinal Survey Data

All sections of CPSC215 (summative survey data)						
Survey Question#	**Pre M**	**Pre SD**	**Post M**	**Post SD**	**t**	**Sig. (2-tailed)**
4	4.20	1.11	4.11	1.37	-0.34	0.74
5	4.45	0.93	4.07	1.33	-1.53	0.13
6	4.78	1.05	4.30	1.08	-2.16	**0.03**
7	4.40	1.45	4.00	1.26	-1.43	0.16
8	4.90	1.17	5.02	1.12	0.50	0.62
9	4.35	1.42	3.76	1.27	-2.11	**0.04**
10	4.08	1.33	3.85	1.29	-0.82	0.42
11	4.18	1.32	3.74	1.43	-1.50	0.14
12	4.30	1.20	4.42	0.94	0.55	0.58
13	3.95	1.36	3.73	1.32	-0.78	0.44
14	5.00	0.99	4.81	1.01	-0.91	0.36
15	3.93	1.33	3.50	1.29	-1.55	0.13
16	3.93	1.42	3.87	1.46	-0.20	0.84
17	3.20	1.45	3.48	1.43	0.93	0.36
18	4.20	1.38	4.25	1.15	0.19	0.85
19	4.10	1.22	4.17	1.12	0.30	0.77
20	3.75	1.45	3.37	1.34	-1.32	0.19
21	4.68	0.94	4.54	0.78	-0.76	0.45
22	4.58	1.34	4.42	1.23	-0.57	0.57
23	4.53	1.26	4.42	1.29	-0.38	0.71
24	4.65	1.19	4.79	0.85	0.65	0.52
25	4.33	1.23	4.50	1.00	0.75	0.45
26	4.88	1.14	4.46	1.29	-1.60	0.11
27	4.98	1.17	4.44	1.19	-2.14	**0.03**
28	5.23	0.95	4.71	1.02	-2.47	**0.02**

All sections of CPSC372 (summative survey data)						
Survey Question#	**Pre M**	**Pre SD**	**Post M**	**Post SD**	**t**	**Sig. (2-tailed)**
4	4.30	1.105	4.13	1.180	-0.52	0.61
5	4.17	1.029	4.61	.988	1.46	0.15
6	4.43	.992	4.39	1.196	-0.13	0.89
7	4.35	1.027	4.57	.992	0.73	0.47
8	4.83	1.029	5.09	1.041	0.85	0.40
9	3.57	1.441	3.65	1.434	0.21	0.84
10	4.74	.752	4.35	.935	-1.56	0.12
11	4.87	.869	4.78	.795	-0.35	0.72
12	4.74	.752	4.74	.619	0.00	1.00
13	3.65	1.369	3.30	1.222	-0.91	0.37
14	4.26	1.176	4.70	.765	1.49	0.14
15	4.13	1.325	3.57	1.308	-1.46	0.15
16	4.78	.951	4.00	1.279	-2.35	0.02
17	3.52	1.473	3.91	1.311	0.95	0.35
18	5.04	.767	4.39	.941	-2.58	**0.01**
19	3.48	1.410	4.57	.788	3.23	**0.00**
20	3.87	1.359	3.39	1.196	-1.27	0.21
21	4.39	.941	4.35	.982	-0.15	0.88
22	4.61	1.305	5.00	1.168	1.07	0.29
23	4.96	.928	4.87	1.180	-0.28	0.78
24	4.96	.825	4.57	.992	-1.45	0.15
25	4.65	.832	4.61	1.076	-0.15	0.88
26	4.87	.869	4.65	1.027	-0.77	0.44
27	5.00	.853	4.87	.869	-0.51	0.61
28	4.87	.869	4.78	.850	-0.34	0.73

Appendix G

Transcription of the Focus Group Meeting

Date: 5/1/2012

Moderator: So, you are already done with your exams and everything, right?

Participants: Yep, yeah, ugh-uuh

Moderator: That's a relief, isn't it? Yeah

Instructor2 telling about having to proctor another exam for another professor.

Moderator: We are going to discuss what you have experienced teaching 215. This is not to evaluate your teaching and what you have done right and what you have done wrong, we want to see how we as dept incorporate reasoning principles in the curriculum, how students are responding, how we can improve, and what we researchers can improve so you can use it in class.

(*Moderator* explains that if any statements are used in the research, names will not be used. Telling instructors that participation is voluntary). (Participants nodding in agreement.) The first question I want to ask is … and I am the moderator I will not be giving my opinion, I will just ask question to start the discussion and write down some results. (Participants nodding and agreeing)
So, the first question is what percentage of the semester topics were reasoning topics from RCI?

Supervisor: When you say ~~"r~~easoning" you also mean a part of our formal specifications.

Moderator: Yeas, yes, of course, formal specifications. I know you introduced several items, because I already talked to you, what percentage of the course material were reasoning topics?

Instructor2: I was going to say 3 weeks

Instructor3: Yeah, three weeks, this is like 20%.

Instructor1 walks in.

Moderator: Hi [*Instructor1*], would you please sign the release form that you agree that I can use this discussion in further research. I am not going to use your name. (*Instructor1* signs without reading) So much for the reading it.. I already asked the question and you can pitch into the discussion, approx. what percentage of the course material that taught

in 215 was covering reasoning topics? I am talking about specifications, reasoning tables, and whatever else.

Instructor1 interrupts: Formal specifications? Not just contracts? Ahh.. We spent 2 weeks on that.

Moderator: Were there any difficulties in incorporating reasoning topics into the ~~"n~~ormal" 215 topics?

Instructor3: I would say YES. Things were very sequential at the beginning of the semester, like Java, Java, this is what you do… and then all of a sudden we are going to talk about formal reasoning. So, I think this was like a jump….

Instructor2: Yeah, it was like a hard left turn when you get to that portion of it. Because you know for the 1st quarter of class I explained Java to them, and then for the second or so quarter of the class we were talking about SE contracts in Java and stuff like that, and then I remember I said ok, we are going to do something very weird now, we gonna take a hard left turn, we going to talk about math. So, I mean I think this is a fine topic to look at, and in fact this is one of the benefits of our 215 course, but it ….does not actually transition well given the material in this way …

Moderator: [Instructor1], what do you think?

Instructor1: I did not think this was much of a left turn. What became the weird part is when we started looking at RESOLVE and that stuff. But formal specification, I mean to me this just flows naturally, we were already looking at contracts, we were already looking at the informal specifications. This was not a big deal, I did not think. Now when we started looking at RESOLVE code and building proof tables, yeah, this was a little bit out there.

Instructor2: I guess in this way I did not put this much emphasis on informal specifications, or contracts outside of formal contracts. So, I talked about contract programming early in the semester, because I feel like you can talk about exceptions and defensive programming without bringing up other side of the coin and at least introducing it, but I did not focus as much on informal specifications, in fact we did not even really cover that till we got to the formal part where I said well here is informal specification and here is why they are bad, or not as good as they could be.

Moderator: So you already mentioned how you handled that sudden introduction of that. Hampton, how did you handle that jump from one topic to the other? From talking about Java and everything to talking about RESOLVE?

Instructor1: Well, it sounds like I spent a fair amount of time talking about formal contracts, I do not know what *Instructor3* did in her class. But from *Instructor2* is saying,

we too had a full … on contracts, informal… informal, and so, the jump was more like obvious, let's look at this Java API here this really leaves a lot out there, right? We are still guessing a lot, in fact a lot of talented software engineers would use contract here. So I used that to motivate it.

Instructor3: For me I started with doing regular Java and then implementation, and then what about implementation, can't really account for every possible invalid input, it's not practical. And here is the design by contract, and this is a pre- and post-condition, was very informally introduced, and then later on we, I introduced formal specifications and reasoning tables.

Supervisor: Before I ask you more specifically, what week number was that, when you first told them about formal precondition or formal mathematical model, or anything? What week was that? Does not have to be that exact..

Instructor3 does not remember, wants to log in to her website and tell that.

Supervisor: Do not worry about it.

Instructor2: It was the week before spring break.

Instructor1: How many weeks do we have roughly? 15. Probably..

Instructor2: I remember yours was towards the end.

Instructor1: Yeah.. Last ¼ of the class.

Instructor2: When I did that guest lecture, this is great, we get to take a break from math. I did mine the week b4 spring break, and the two weeks sort of following it. And every now and then there would be a lecture on formal stuff, and some informal stuff. Or split lecture in half.

Supervisor: So, it's like about week number 10?

Instructor2: Something like that.

Instructor3: For me, I introduced it much earlier, probably like week 5 or something.

Supervisor: Ok, and then showed various specifications.

Instructor3: Yeah, I devoted 2 or 3 weeks, maybe 2.5 weeks for the formal specs and the RTs. And then I continued on with Java stuff. And then I gave them some later too.

Moderator: This brings up to the question: did you talk among themselves at what time you introduced specific topics?

Instructor2: Yeah, I was aware of the stuff. She introduced it a little earlier, and I waited a little while.

Moderator: So, did you have to comm. with other instructors and ask questions like ―Have you already introduced this topic? When are you going to do that? ―

Instructor3: We have labs that Jason already prepared as our guide.

Instructor1: There was a fair amount of communication as well. *Instructor3* and I really worked teaching the same stuff at the same time.

Moderator: Next I wanted to look at the RCI and if you have not seen it yet…

Instructor2: It's huge..
(Laughing..)

Moderator: It is detailed… ☺ I know I asked you to mark it yest, but I do not know if you all got to it yet. Let's look at number 3.

Supervisor suggests we start with number 1.

Moderator: Yes, Have you guys look at the logic part? Have you talked about it anough for students to be accountable for the topic, say, they have to remember it, talk about it, etc.

Instructor3: ughuuh

Moderator: If you just asked you guys know what existential quantifiers are? Ok, let's move on. This does not count..

Instructor3: I covered exist. and universal quantifiers in class and it was incorporated in the quiz questions. Like there exists something in the pre-condition ..

Supervisor: Mark that then.

Instructor3: But it was not a question just for that, it was a part of a question..

Moderator: It was incorporated….

Instructor3: Yeah.

Moderator: to *Instructor1*: You are nodding your head, does it mean you did the same thing?

Instructor1: no, we saw exist and universal quantifiers in the context of specifications. I took a second do you know what this is and we went over it, I would not say I taught it, but I spent 5 minutes on it. It was not on test or anything..

Moderator: ok, how about you *[Instructor2]?*

Instructor2: I covered exist quant. And there was a quiz question and a question on the final exam that they just took. I covered it only in the realm of formal reasoning.

Moderator: ok. (Explaining how to mark it on the rci.) Now, let's go down the list and see if you see anything that you got covered. Whether the entire section from column 2, or just a concept on the rightmost side

Instructor1: Next thing I marked here is proof (*Instructor3* says yeah). This of course was tested.

Instructor2: yeah.

Moderator: Ok, let's mark this as yes.

Instructor1: asking a question about proof section, whether it is reasoning tables.

Moderator: Proof techniques.

Instructor1: We did forward reasoning, they saw that, and maybe did not realize that this was a special thing, proof.

Moderator: Same for you guys? Got it covered?

Instructor2 and *Instructor3* yay in confirmation

Instructor2: I did not go into the mechanics of going about proving something, because I surveyed my class and every single person in there took 207 (Discrete math class) , and I had one math grad student, so we did not specifically talk about proofs, except the fact that we are doing proofs, and exist quant.

Moderator: Ok, let's move down the list. Anything else you guys covered, mentioned, or taught?

Instructor2: I did the set basics,

Instructor3: Yeah

Instructor2: empty sets, subset, subset theorem, I some algebra on sets. Associativity and commutativity, union, intersection..

Moderator: How about you guys, anything else u see?

Instructor1: Cartesian product

Instructor3: We did basically just the union, intersection, disj sets, subsets…

Moderator: So your selection of these topics was based on that this is a prerequisite knowledge of something you are about to teach?

Unanimous Yeah from all instructors.

Moderator: you were not told teach this and teach this, you just knew this is the prereq part, so you covered it.

Instructor2: yeah, and in fact if we did not know this is prerequisite to what we are doing, I did not think we would get into hash tables or anything like that.

Supervisor: You taught it in the context of the specification of Stack template or something like that.

Instructor2: No, I did this as straight mathematics. Let's see how this is used in formal reasoning. I showed set, subset, intersection, union… in fact when I tested them on that it was just a pure math set, not say sets from a formal reasoning standpoint.

Instructor3: I did this in the context of Java sets, ok, this is set in Java, let's see the equivalent of it in math.. so there is no order if you want to get an element from a java set then there is no index, no order, so..

Instructor2: I indirectly taught about that when I introduced hash maps early on, that there is no specified order, and u get no guarantees on the order that comes out, I presented these ideas.

Supervisor: What about strings?

Unanimous yes

Instructor3: lots of strings

Moderator: Let's mark that

Instructor2: Alpha over sigma, sigma star

Instructor1: Full background

Instructor3: Yeah

Instructor2: I used one of the projects straight-line parser that generates a language with no branches. It's just s very simple grammar for doing arithmetic operations, statements, and printing. It has formal grammar mapped to a set of classes. O I spend a day or a day and a half talking about formal grammars. So we had to go over alpha over sigma, sigma star, then I also tested them on empty strings, string concatenation. We talked briefly about string reversal. Yeah, lots of strings.

Moderator: all right…

Supervisor: So it seems like *Instructor2* talked about this in mathematical context. And you guys talked about it more in the context of a component, or a

Instructor1: Mine was in the context of formal specifications. So in this sense it was math, but it did not come out like let's talk about sets. .. Intersection was to permit multiple entries.

Supervisor: were there any questions, did you all have questions about their understanding of sets, or maybe …

Instructor3: It was like hard, but for example if I wanted, I think we had a set on sets..

Instructor1: Only indirectly

Instructor3: Yes, it was not like here is a question about mathematical sets.

Instructor1: Here is a stack that uses sets….

Instructor3: Yes., exactly,

Instructor2: [*Moderator*], if you want I have all the notes I have prepared to teach .. you can photocopy this if you want..

Moderator: That would be nice.. I wanted to see it. Let's move on to item number 3, precise specifications. When you introduce precise specs, what type of motivation did you give, did you tell them why they need to know that?

Instructor2: I talked about informality of natural languages, and I used the classic ―little boy touched a little girl with the flower" kind of example to show that if we use natural languages it can be ambiguous. And we need to be precise, we need to use mathematics, because math precise and not ambiguous.

Instructor3: I did the same form the Java API, pull up some methods and ask do you undserstand that, and I used a fact that I come from another country, and that I can understand it this way, and I said that this is why natural langs are ambig, so we use math.

Moderator: Yeah, even aliens use math to communicate with humans (this was a joke)

Instructor1: I used Java API as motivation.

Moderator: so everyone motivated why we need to use precise specifications.

Unanimous yes.

Supervisor: Did you ask a question about problems with informal specs on a test?

Unanimous No.

Moderator: Did you have anything on the test that said why …

Instructor3: I think I may have had one question like why do we use formal specs. I think I may have one. I am not 100% sure.

Instructor2: I don't think I did qualitative analysis question on it.

Instructor1: Early on we talked about formal specs, I presented them with a really bad spec, really high level…

Moderator: When you were introducing formal specs were there any students that really hated it, or somebody who completely did not understand it, any negative reaction from students that you noticed, or did everybody go ―oh wow, this is cool"?

Instructor2: I think the only person who was ―oh wow, this is cool" was the math grad student I had. Because this was very easy for her.

Moderator: But did students react in the positive way to this, because this is new in the 215 course?

Instructor2: I'd say it was neutral or they were hiding their negativity.

Instructor3: My class was very quiet, so I did not get much of the reaction from them.

Instructor1: I had some people who thought it was cool. Not like WOW, but let's see if it is useful. And most people felt the same way they felt about the rest of the material: how am I going to be tested on this.
Others agreeing.

Instructor1: If they was negative reaction like *Instructor2* said, they were hiding it.

Instructor2: People were vocally frustrated with this at times, but …

Moderator: How were they vocally frustrated?

Instructor2: I think this is a difficult subject for sophomores, and I think the biggest thing was when they were doing proofs. They did not know what was sufficient.

*Mod*erator: Why do you think this is a difficult subject for sophomores?

Instructor2: Because they only had one rigorous course in math, and that class, I talked to some people, this class probably comes a little early, I think this stuff will be much easier to teach, after kids took a 350, and have a lot more experience with rigorous proofs.

Instructor3: And another thing is. They come from this mental model? that math is hard. When we tell them it's math then they will just block it. Like this is hard, so.. Some are like, this is math, it is hard, so I better pay attention, others are like ok, I am done, I don't like it. I don't wanna do anth with math so they decided early on that they are not going to understand this.

Instructor 2: And not just the word math, it's the word ~~"p~~roof".

Instructor1: Yes, proof is a dirty word.

Instructor2: It's a dirty word for undergrads.

Instructor1: And I think, part of it is also that they got a new CS to tack around with, and they have not yet at the part of the career where they figured out what CS is. This is one of their first wakeup calls. Hey, this is the sort of thing we do in CS. We are talking about kids that are hard enough to convince that it matters how they structure their code. It works who cares. It is hard to get them to that level, and then we throw them into this math. It's hard for them enough, their brains aren't yet organized.

Instructor3: And another thing I noticed is that it was either black or white, they either get it or not. So a lot of them are like, yeah, I kind of get it. They either do really well or miss it.

Moderator: Back to where you said –proof is a dirty word". Where and how did it happen that they consider proof a dirty word. Should it be addressed somewhere before they come to 215? How would we make they don't think that proof is a dirty word, but something familiar and ..?

Instructor1: Without overhauling the public school system of the United States?

Instructor3: Yeah, I was just going to say the same thing.

Unanimous Laughter...

Moderator: So they come with this notion from school?

Instructor3: Yeah, even when I was an undergrad in a class they started talking about proofs, it was bad… Because in high school generally speaking the only proofs you see are (in my education) geometry, sides, complimentary angles and stuff like that. Proof that something is of certain degree… and given that proofs do not really have a mechanical way of doing it there are no deterministic steps to do the proof, it is hard for kids to wrap their head around it.

Moderator: So we got them already spoiled?

Unanimous yes!

Moderator: Just trying to determine if this is something we should do…

Instructor1: What we needed to do is a class on theory of computation where we present proofs as what proofs are, trying to show something that is, not this stupid thing you have to do to jump thru the hoops to get smth done…

Supervisor: I think if somehow they could see some software connected to something, and you are saying that correctness of this little piece is important,

Instructor3: Yeah, I have shown them exceptions, if you do not need my requires clause then we end up with run time exception. So they can see visually this is what happens when you do not follow the contract basically.

Moderator: Ok, lets look down the table now. Specifications structure. You all guys covered that in your class? Usage requirements, ..spec. inheritance..

Instructor2: I talked about signature, But I do not quite know what do you mean by usage requirements.

Instructor1: What are usage requirements?

Supervisor: Yeah, you probably do not even deal with bounded stacks.. that's when you create a bounded stack. Then you pass a max depth greater than 0.

Instructor1: we did something like that.

Instructor1: when I introduced it, I very much stuck to Dr. J's and Dr. H's tutorial, and they do not use bounded, and I said why don't we have bounds on that because we can't do any reasoning about running out of bounded memory, we can sort of prevent it in our specs, but we can't really reason about running out of memory. So I said let's not worry about that right now. And when they got to RESOLVE I said well they actually put a bound on it because this is a real system, and they wanna make sure they do not explode the stack or whatever.

Moderator: How about models for conceptualizing components? U intro'd strings and sets?

Instructor3: Yeah

Instructor2: Yeah

Instructor1: Boolean Ints, functions briefly, definitely Cart products, that's it.

Moderator: So here students actually saw why you need to know math models to model software components.

Instructor2: Yeah, I talked about sets, strings, and sequences, and used test question. Which math models are used to model abstract data types. Can a math set be used to model a math DS? Why or why not?

Moderator: I see, I recognize the question lol

Instructor2: and there is another question we can represent any ds with a string. T or F?

Moderator: Ok, please mark it as we go down the list.. Was there also a question about set vs. string…

Instructor1: yes

Instructor3: yes

Moderator: Next specifications of operations? Pre- and post- conditions? What did you guys cover and what you did not?

Instructor2: oper name, formal parameter, return value,

Instructor3: I covered the whole thing.

Instructor2: I talked about soec parameter modes, but only once we had that part on RESOLVE. And only by necessity . to keep as much mystery out of it as we could.

Instructor1: I may have touched briefly on that one.

Instructor2: responsibilities of caller and implementer.

Moderator: Were there any negative reactions why you need to know all that?

Unanimous no.

Instructor2: When I presented it, I presented it as an alternative to defensive programming. I said it makes it easy to have a very distributed team, when we hand them specs and make them implement it any way they want. They were fine with that. I do not think they like ar appreciate defensive programming any more than they do contract programming. I do not think they are far along to prefer one or the other.

Instructor1: When I presented it, it was more in the content of you are doing it all the time without realizing it. By doing this we are only making it explicit. Every time you type —. —you have it in your head that you prove it is not a null…

Moderator: How about modular reasoning. You guys had any for motivation?

Instructor1: yeah ahhhhh

Moderator: You guys go down the list and call if you see something you covered.

Instructor1: All of 4.1.1. and tested on it.

Moderator: Formal verification. From talking to you guys aI know you covered it.

Unanimous yes.

Instructor3: They had it on the test.

Instructor2: I had them proof all kinds of stuff on test.

Instructor3: I never had them proof anything on the test. We did a ton on the board in class

Instructor2: And we did a ton on the board in class.

Moderator: Ok, go ahead and mark that, I would like to keep your copies after you mark that.

Instructor2: What do you mean by error detection, like throwing exception?

Supervisor explains.

Instructor2: ahh, ok, unit testing, and stuff like that.

Moderator: How about DBC? Roles? You teach Java so you did cover this one way or the other.

Instructor3: yes

Moderator: Can you give an example?

Instructor3: An assignment with formal specs and they implemented it following the specifications. And they did JUnit testcases.

Instructor2: I did similar one. I gave them a vector class , with specs, told them to find javadocs, pre and post conditions, and unit test for it. We know if it failed it uncovers a bug. And another where I gave them informal specs and they have to develop formal specs. And write unit test for it, but only 2 people did that, and they both did horrible.

Supervisor: This is a very hard question for them.

Moderator: Instructor1pton, how about you?

Instructor1: I am lost as to what we are talking about.

Instructor1: Construction of new comp from existing components.

Instructor1: Sure.

Instructor1: Ok, move next?

Supervisor: Wait, the thing is you probably did not talk to them about formal reasoning of correctness?

Instructor2: No, no , no

Supervisor: See this is what it is about. Not about them, you are not teaching them how to reason about them. Si I think what you are saying if you probably did 4.2.1, but not necessarily 4.2.2., or 4.2.3 for example.

Instructor1: we did 4.2.3

Supervisor: Formal reasoning?

Instructor1: Informal reasoning. This is how I motivated modules.

Supervisor: Oh, that is very good.

Instructor1: Enhancement specification does not mean anything in Java so I did not touch that.

Instructor2: And now that I think about it, 4.2.3 I touched on, I talked about how we get a bunch of modular data structures, and it is easier to build code because we know that ds is correct. Small components as building blocks. I did kinda cover that.

Supervisor: 4.2.3.2 ion the next page you may not have used it as it, but if you ask them to do a reasoning table this is where you used 4.2.3.2. Even if you did not use the term enhancement or anything, if you used some specs and some code and ask them to reason about it, which is pretty much what rt have done, it's just that classification, it's only so many ways to put it.

Instructor1: Anything else from 4.3 maybe?

Instructor1: 4.3.1. covered, and 4.2.2.

Instructor2: I talked about loop invariants, from definitional standpoint, true before we enter each iteration, …, I talked about it again as a progress metric as something simple. And then I showed them one loop invariant that was stack reversal, and showed why it's true at each iteration. But I never tested them on it.

Moderator: Are we done with number 4, should we move on to correctness proofs?

Supervisor: So, *Instructor3* and *Instructor2*, you prob did not do 4.3.1? Internal contracts?

Instructor2: no.

Instructor3: no.

Supervisor: *Instructor1*, you did 4.3.1, but not 4.3.2, *Instructo31*, you did not do either of them?

Instructor3: no.

Instructor2: I did talk about loop invariants, but not that, I believe it is a little too hard for kids at that level. But they should at least know what the concept is. And then I also asked them about loop invariants, but it was only give me a definition of a loop invariant.

Supervisor: Ok, Correctness proofs, last item. Did you give them motivation for correctness proofs for SE, not just mathematical proofs.

Instructor1: Not in all these smaller bullet point senses. I did not talk about semantics…

Supervisor: But did you motivate why….

Instructor1: yes

Instructor3: yes

Instructor2: I guess I talked about it from the point of view of mission critical software. And it is impossible to test all the invalid inputs, so at some point we need something a little bit better than that.

Supervisor: So what we have here is, if you mark with the C under 5.1 , it does not mean C for all the subordinate things. You may need to clarify that you just did a 5.1.1 informally.

Instructor1: I am going to use C* for that.

Supervisor: You can make special notes there too, if you'd like. Next item verification conditions. I know you built proof tables. Did you do that?

Instructor2: Yeah,

Instructor1: Yeah

Instructor3: Yeah

Supervisor: did you make them watch proof tables videos?

Instructor3: they got some of them confused.

Instructor2: The 3rd one was what all my students have complained about.

Supervisor: The third one is the most difficult to watch.

Instructor2: It introduced something they know, but they did not expect to see it this way. And it just thru them off.

Instructor3: The table in Jason's tutorial is different, and there is not column for code, and that through them off. They are used to have a table with the certain number of columns, and the video has an extra column. And some things you put in, some things you don't.

Moderator: Did you discuss this in class when students complained that video showed it one way and tutorial another?

Instructor3: It is just a different column, and I told them that the video will be different from what they have seen in class.

Moderator: A lot of students got confused?

Instructor3: I could not tell how many till the quiz. On the quiz I had one students that did not fill the table correctly. He filled the code in the facts column. And facts in obligations column, and obligations were not even there.

Moderator: Did you experience other difficulties with proof tables?

Instructor2: I had a very similar experience as *Instructor3*. I got irate with students saying this is merely a syntactic difference. We have a presentation difference, we have a code column, it is there to help you, so you look at it and say yes of course. And eventually they were like, oh, ok, we get it, so it was mostly them being uncomfortable with something being a slightly different.

Instructor1: This is sophomore class, something is required.

Instructor2: And on the test I put the code in but blacked out what they did not need to fill in. Because after the video kids put stuff in the wrong columns as the result of that.

Moderator: So the difference in the table look confused the students?

Instructor1: yes

Instructor3:; yes

Instructor2: I like the code column myself because then you do not even explicitly need to give the code. But we need to revise it in one specific way to make it uniform.

Instructor1: I think students should be able to figure it out.

Instructor3: They should, yeah

Instructor1: People have different compassions about things…

Instructor2: When we gave the RESOLVE lab there was a lot of complaining and resistance about that because they said what is that, and I said it is the same we are teaching you techniques that work across the languages, across syntaxes, it's a way of thinking, not a specific tool to use.

Supervisor: Did you show them the Java button. There is a button there that you could make it look like Java. But I do not know if you told them about it.

Instructor2: I was not aware of it. About 2/3 of the class got confused by the third video, 1/3 of the class got confused by the second video, I do not know why.

Supervisor: Third video is one step too far.

Supervisor: So, under 5.2 you probably covered 5.2.1. and 5.2.2., and 5.2.3. maybe just sequential statements.

Unanimous yes to all of the items.

Instructor1: Looks like you did some loop*, Instructor2*.

Instructor2: Just because they asked and I had to sho what is involved in proving a loop, so this is what I do, let me write you out a stack reversal.

Instructor1: Did you do a proof table there?

Instructor2: No, I did not do that. They wanted to see what an invariant looks like, so I wrote out the spec for it. Then I wrote it and said, here is what it looks like, do not worry about it, it will not be on the test.

Moderator: How about 5.2.4.

Unanimous no.

Moderator: Proof of VCs. Did you have students prove any of the generated VCs?

Instructor3: We did it in class, but I did not test them on it.

Supervisor: So you did like 5.2.1?

Instructor3: I did the whole thing, just that I did not test them on it. We did it in class and they participated.

Moderator: So, mark this as covered.

Instructor1: I did not do inference, but I did the rest of it.

Instructor3: I did not mention the term.

Instructor2: I did not specifically talk about direct proofs or rules of inference. But they did at least two proof on quizzes and one proof on this test. And we had a home work on it. And I did lots in class. They wanted me to keep doing examples.

Supervisor: One observation that I will make. If you look at the number of topics being covered, there are a lot

Instructor1: yeah, a lot

Instructor2: I did not realize there are that many topics to talk about.

Supervisor: It seems that you have covered so much!! And again, our goal , is the introduction of the idea, they will see it again in a later course. It seems we have done quite a bit.

Moderator: On a personal note, you guys are doing a really great job. And I am looking at this RCI, and even if you only mentioned you have only slightly reviewed stuff, students are still aware of that. And this is an important thing. I also wanna ask you a couple more concluding questions. Was there some techniques that you used to intro some topic that really was a success? Or something that was a complete failure? So we know what works what does not for certain things.

Instructor1: The first time I taught the class, this will be both a success story and a failure story, I do all my projects, I gave them an almost list-like language and asked them to make a mechanical proof table generator. First time I taught the class they bested it, did it great, did it awesome, I thought it is a really hard project, it sounds hard even to me. Next time it was a disaster, they hated it, almost nobody got it right, I had to curve it like 20 points just to get them to the point…

Moderator: Did you teach them induction the same way as the first time?

Instructor2: Yeah

Moderator: Why do you think they failed it the second time, and aced the first time?

Instructor1: I asked and it was an issue of having other projects due at the same time. And they just did not have enough time to look at it.

Moderator: Was it spring sem you taught it second time?

Instructor1: Fall, both times it was fall.

Instructor3: It is very hard to compare students. Because last semester when I taught the dynamics of the class was very different. The class was very quiet, and onlu one or two people ever speaking, this semester it is completely different.

Moderator: How different? More outspoken? ..

Instructor3: First I had more students, they ask a lot more questions that last semester, and like, even their grades were very close to each other. And last semester were big gaps in grades.

Instructor2: One thing that I like doing when we are talking about reasoning and math stuff, we went and did stack spec, and I told them to raise the hand and tell me what pre- and post condition would be, and when someone answers we would discuss why this is a good one, and why this one is bad one. Like, a length of the stack increases by one as a post-condition, rather than having a concatenation, because it implies that. That worked well, as far as class dynamic, left side of the room were all bc students, right side were all A students. Goint into final there were 5 As, 7 Bs, 6 Cs, 3 Ds.

Moderator: Did you exempt any students from final exam?

Instructor3: yes, exempted 5.

Instructor2: no.

Moderator: Last question: if you were to teach this class again, what would you like us to do more or better so that we provide you with teaching materials, exam questions.

Instructor3: More intro of RESOLVE syntax.

Instructor2: Yeah, that was the big problem they had with the lab, it was not that the lab was too hard, it is just that it took them a while to get syntax right, and it even took me 15 mins because I was struggling with the syntax.

Supervisor: Can you be more specific, was it specifications, or concept…

Instructor3: Enhancement, and then the syntax itself, because all they have in their minds is Java or C. First they never heard of RESOLVE before, so they can't realize hey this is the language with its own grammar. So they can't figure this out yet. .

Instructor1: They are still in that mode too that they came in with C, and then they learn java that is almost exactly like C, and it is pretty intimidating.
Instructor2: We have not talked about templates, concepts, realizations, enhancements, and stuff like that. So it's the very structure of it, and then literally the syntax. Variable name, type, or type, then name, colon…

Moderator: Did you know there is RESOLVE manual online, though outdated.

Instructor2: What would be better is having a compiler button for a cheat sheet syntax.

Unanimous agreement.

Instructor2: Remember, this is undergraduate students, and as good as my grade students were, no one wants to sit around and read the manual. They do not want to read any reading assignments. I was surprised when they told me they watched the 15 minutes worth of videos. They do not want to look at the manual, and they were upset the help button in IDE did not work.

Instructor1: It is just there to tease.

Instructor2: This was one of the issues with it. Especially that we did not explain the underlying math stuff that you guys do in RESOLVE that fuels the concepts, so they asked why ints have to be between max_int and min_int, and all the extra stuff.

Supervisor: One thing I did at the workshop, I did it for the first time, and this is something we might try again in 215, is all the reasoning we did there, we just wrote a little operation, just a facility, there were no enhancement, or anything, and we cpuld press the button show it is Java so it even looked like Java. Then you could press the VC and show VCs. So, you never really looked at concepts, never looked at anything else, all it was one facility with everything you wanted to see.

Instructor2: I think this would be a great way to show to the very introductory students because that very much veers at least how I presented to them which in some cases, here is the function with pre and post conditions..

Supervisor: Ok , I think this is something that we may think of doing.

Moderator: Is there anything else you wanted to let us know.

Instructor2: You asked what we all could do better, and this is timing, we introduced RESOLVE, and then we had a spring break. And I originally planned to give them a homework that spring break, but then I decided tit was a very bad idea. So, part of the reason why I only spent 3 weeks on it, was a timing issue.

Instructor1: This class shd be two classes, a Java class and a reasoning class (*Instructor3* says yes).

Instructor2: I even see this as three parts. Reasoning part is SE.

Instructor1: I still think you can give this a decent intro in one. The biggest problem with 215 is 101, 102, and 212, mu kids come and they cannot program. They do not understand pointers,

Instructor3: recursion is very bad.

Instructor1: Recursion I understand, but if you tell me you were programming for 3 classes, and I will tell you this is pointer, you've got it. And I do not need to say anything more than that. And I understand I could give a refresher, but every year I give 6 quizzes, and every quiz there is a question about passing by reference, and at the end of the semester I still got 5-6 kids who do not get it.

Supervisor: I guess you guys are doing better than that, every time I ask a question in 372, and there is a question about stack in Java, when I ask then when you push n a stack, is this is reference, and two years ago only 2 out of 20 thought it copies a reference, and the last time I asked 16 or 17 have got it right. This is like on specific data point but it is very precise, is what is being copied when you push on a stack. So, maybe they are understanding coming out of 215 that there are references and references.

Instructor1: Good

Instructor2: The other thing about 215 is that there are a bunch of people in ccit who took 215 with Dr. H, and with us TAs in it, when they come out of the class taught by PhD students, they feel much better about the class, but they may not have learned as much, where, when they take it with Dr. H, he murders kids. But the ones that happen to surf to the surface, end up being very good programmers. I do not know why this is, not sure why, I know I am more approachable than some of the professors here and I tend to be more compassionate because ..

Moderator: Some students feel better approaching another student that a professor.

Instructor2: And Hapton brought up a good point that a lot of kids who come to class do not know how to program.

Supervisor: This is a huge surprise to me.

Instructor2: Mine was not that they do not know how to program, just a bad code.

Instructor3: They are not taught to write good code.

Instructor2 tells about bad singleton code examples he saw. People with 15 return statements.

Instructor3: 102 TAs do not look at the code. When they come to 215 they do not know about bad code.

Supervisor: Do u think they see some connection between mathematics and software by the time they leave? Because this is the big picture. There are a lot of things we teach them along the way.

Unanimous yes.

Instructor2: Yes they do.

Supervisor: yes I think this is the thing.

Instructor2: 215 seems to be an undergraduate coming of age class. So they get thru the 4 weed out classes, and then they get to a more interesting stuff. So this is the final big set of tools they got to help them

Moderator: Thank you very much guys, this is over now.

Appendix H

Instructor-Marked RCIs Indicating Topic Coverage

CPSC215 Instructor 1

Reasoning Concept Inventory

The purpose of this inventory is to identify the basic set of principles that are central for analytical reasoning about correctness of software and that must be taught in undergraduate computing education.

Major Reasoning Topic	Subtopic Summary	Concept Term Highlights	Concept Details
1. Logic	1.1. Motivation	1.1.1. Motivation for Boolean Logic	
	1.2. Standard logic symbols	1.2.1. Connectives including implication	1.2.1.1. Simple statement 1.2.1.2. Connectives (NOT, AND, OR, IF... THEN, IFF) and compound statements 1.2.1.3. Truth tables 1.2.1.4. Logically equivalent statements
		1.2.2. Quantifiers	1.2.2.1. Universal quantifier 1.2.2.2. Existential quantifier
	1.3. Standard terminology	1.3.1. Proposition	1.3.1.1. Propositional variables 1.3.1.2. Compound propositions 1.3.1.3. Proof arguments
		1.3.2. Predicate logic	1.3.2.1. Predicate calculus
		1.3.3. Proof	1.3.3.1. Informal proof 1.3.3.2. Axiom 1.3.3.3. Premise 1.3.3.4. Conclusion
	1.4. Standard proof techniques	1.4.1. Supposition Deduction	
	1.5 Methods for proving	1.5.1. Direct proof 1.5.2. Proof by contradiction 1.5.3. Vacuous proof 1.5.4. Trivial proof 1.5.5. Proof by cases 1.5.6. Exhaustive proof 1.5.7. Proof by induction	
	1.6. Proof strategies	1.6.1. Forward reasoning 1.6.2. Backward reasoning	
	1.7. Rules of inference	1.7.1. Modus ponens 1.7.2. Simplification Conjunction 1.7.3. Universal	

		instantiation 1.7.4. Universal generalization 1.7.5. Existential instantiation 1.7.6. Existential generalization	
2. Discrete Math Structures	2.1. Motivation	2.1.1. Motivation for Discrete Mathematics	
	2.2. Sets	2.2.1. Set basics	2.2.1.1. Set naming conventions 2.2.1.2. Universal set U 2.2.1.3. Empty set φ 2.2.1.4. Element naming conventions 2.2.1.5. Subset 2.2.1.6. Subset Theorem 2.2.1.7. Venn diagrams
		2.2.2. Set notations	2.2.2.1. Union 2.2.2.2. Intersection 2.2.2.3. Disjoint sets 2.2.2.4. Relative Compliment 2.2.2.5. Absolute Compliment 2.2.2.6. Power set 2.2.2.7. Power set for a finite set 2.2.2.8. Cartesian product 2.2.2.9. Extensibility over finite set 2.2.2.10. Cartesian product of a set on itself
		2.2.3. Laws of Algebra on Sets	2.2.3.1. Idempotent Laws 2.2.3.2. Associative Laws 2.2.3.3. Commutative Laws 2.2.3.4. Distributive Laws 2.2.3.5. Identity Laws 2.2.3.6. Involution Law 2.2.3.7. Complement Laws 2.2.3.8. DeMorgan's Laws
	2.3. Strings	2.3.1. String basics	2.3.1.1. String variable name conventions 2.3.1.2. Alphabet set Σ 2.3.1.3. Strings over the alphabet set Σ*
		2.3.2. String notations and Properties	2.3.2.1. Empty string 2.3.2.1. String length 2.3.2.1 Concatenation 2.3.2.1. String reversal

			2.3.2.1. Substring
	2.4. Numbers	2.4.1. Number Representations	2.4.1.1. Number notations 2.4.1.2. Number bases
	2.5. Relations and Functions	2.5.1. Relations	2.5.1.1. Relation notation 2.5.1.2. Inverse relations 2.5.1.3. Composition of Relations 2.5.1.4. Reflexive property 2.5.1.5. Symmetric property 2.5.1.6. Anti-symmetric property 2.5.1.7. Transitive property 2.5.1.8. Atransitive property 2.5.1.9. Equivalence Relation
		2.5.2. Functions	2.5.2.1. Function notation 2.5.2.2. Domain and codomain 2.5.2.3. Dependent and independent variables 2.5.2.4. Equal functions 2.5.2.6. Cartesian coordinate system 2.5.2.7. Injective function 2.5.2.8. Surjective function 2.5.2.9. Bijective function
	2.6. Graph Theory	2.7.1. Graph Basics	2.7.1.1. Graph notations 2.7.1.2. Edges and vertices 2.7.1.3. Degree of a vertex 2.7.1.4. Graph representation 2.7.1.5. Tree and forest
		2.7.2. Graph Types	2.7.2.1. Multigraphs 2.7.2.2. Finite graphs 2.7.2.3. Complete graphs 2.7.2.4. k-regular graphs 2.7.2.5. Acyclic graph 2.7.2.6. Planar graphs
	2.7. Permutations and Combinations	2.8.1. Notations	2.8.1.1. Permutation notation 2.8.1.2. Factorial notation 2.8.1.3. Combinations notation
3. Precise Specifications (Mathematical descriptions of interfaces)	3.1. Motivation	3.1.1. Motivation for interfaces	3.1.1.1. Information hiding 3.1.1.2. Independent software development
		3.1.2. Motivation for precision	3.1.2.1. Problems with informal specifications 3.1.2.2. Ease of component integration

	3.2. Specification structure	3.2.1. Specification signature 3.2.2. Usage Requirements 3.2.3. Use of math theories 3.2.4. Specification inheritance (enhancements)	3.2.1.1. Concept name 3.2.1.2. Generic parameters
	3.3. Abstraction	3.3.1. Math models for conceptualizing objects	3.3.1.1. Booleans 3.3.1.2. Numbers 3.3.1.3. Integers 3.3.1.4. Strings 3.3.1.5. Sets 3.3.1.6. Functions 3.3.1.7. Relations 3.3.1.8. Cartesian products 3.3.1.9. Other discrete structures 3.3.1.10. Combination of the above
		3.3.2. Constraints	
		3.3.3. Trade-offs of alternative mathematical models	
	3.4. Specifications of operations	3.4.1. Initialization and finalization specification	
		3.4.2. Operation signature	3.4.2.1. Operation Name 3.4.2.2. Formal parameters 3.4.2.3. Return Value
		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 in the post-condition
4. Modular Reasoning	4.1. Motivation	4.1.1. Motivation for Reasoning	4.1.1.1. Error detection 4.1.1.2. Code tracing/inspection 4.1.1.3. Formal Verification
		4.1.2. Motivation for Modular Reasoning	4.1.2.1. Problems with implementation to implementation coupling 4.1.2.2. Desirable coupling

			through contracts
	4.2. Design-by-Contract	4.2.1. Roles of clients and service providers	4.2.1.1. Specifications as external contracts 4.2.1.2. Client 4.2.1.3. Service Provider 4.2.1.4. Client implementation 4.2.1.5. Service provider implementation
		4.2.2. Construction of new components from built-in components	4.2.2.1. Implementation with arrays 4.2.2.2. Implementation with records
		4.2.3. Construction of new components using existing components	4.2.3.1. Implementation of a specification (data representation, code for operations) 4.2.3.2. Implementation of enhancement specification
	4.3. Internal contracts and Assertions	4.3.1. Internal contracts for data representations	4.3.1.1. Abstraction functions/ relations (correspondence) 4.3.1.2. Representation invariants (conventions)
		4.3.2. Assertions	4.3.2.1. Loop invariants 4.3.2.2. Progress metrics (loops and recursive procedures)
5. Correctness Proofs	5.1. Motivation	5.1.1. Meaning of correctness	5.1.1.1. Semantics 5.1.1.2. Soundness and relative completeness
		5.1.2. Motivation for proofs	5.1.2.1. Partial correctness 5.1.2.2. Total correctness
	5.2.Construction of verification Conditions (VCs)	5.2.1. States and abstract values of objects	5.2.1.1. Naming conventions
		5.2.2. Connection between specifications and what is to be proved	5.2.2.1. Assumptions 5.2.2. 2. Obligations
		5.2.3. Types of Statements	5.2.3.1. Sequential statements 5.2.3.2. Conditional statements 5.2.3.3. Loops 5.2.3.4. Operation calls
		5.2.4. Connection between induction and reasoning	5.2.4.1. Inductive case 5.2.4.2. Base case

			5.2.4.3. Termination
	5.3. Proof of 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	5.3.2.1. Direct proofs 5.3.2.2. Rules of inference

C = Covered T = tested

Reasoning Concept Inventory

The purpose of this inventory is to identify the basic set of principles that are central for analytical reasoning about correctness of software and that must be taught in undergraduate computing education.

Major Reasoning Topic	Subtopic Summary	Concept Term Highlights	Concept Details
1. Logic	1.1. Motivation	1.1.1. Motivation for Boolean Logic	
	1.2. Standard logic symbols	1.2.1. Connectives including implication	1.2.1.1. Simple statement 1.2.1.2. Connectives (NOT, AND, OR, IF... THEN, IFF) and compound statements 1.2.1.3. Truth tables 1.2.1.4. Logically equivalent statements
		1.2.2. Quantifiers C	1.2.2.1. Universal quantifier 1.2.2.2. Existential quantifier
	1.3. Standard terminology	1.3.1. Proposition	1.3.1.1. Propositional variables 1.3.1.2. Compound propositions
		1.3.2. Predicate logic	1.3.1.3. Proof arguments
		1.3.3. Proof C T	1.3.2.1. Predicate calculus 1.3.3.1. Informal proof 1.3.3.2. Axiom 1.3.3.3. Premise 1.3.3.4. Conclusion
	1.4. Standard proof techniques	1.4.1. Supposition Deduction	
	1.5 Methods for proving	1.5.1. Direct proof 1.5.2. Proof by contradiction 1.5.3. Vacuous proof 1.5.4. Trivial proof 1.5.5. Proof by cases 1.5.6. Exhaustive proof 1.5.7. Proof by induction	
	1.6. Proof strategies	1.6.1. Forward reasoning C T 1.6.2. Backward reasoning	
	1.7. Rules of inference	1.7.1. Modus ponens	

		1.7.2. Simplification Conjunction 1.7.3. Universal instantiation 1.7.4. Universal generalization 1.7.5. Existential instantiation 1.7.6. Existential generalization	
2. Discrete Math Structures	2.1. Motivation	2.1.1. Motivation for Discrete Mathematics	
	2.2. Sets	2.2.1. Set basics	2.2.1.1. Set naming conventions 2.2.1.2. Universal set U 2.2.1.3. Empty set ϕ $\subset T$ 2.2.1.4. Element naming conventions 2.2.1.5. Subset $\subset T$ 2.2.1.6. Subset Theorem 2.2.1.7. Venn diagrams
		2.2.2. Set notations	2.2.2.1. Union $\subset T$ 2.2.2.2. Intersection $\subset T$ 2.2.2.3. Disjoint sets 2.2.2.4. Relative Compliment 2.2.2.5. Absolute Compliment 2.2.2.6. Power set 2.2.2.7. Power set for a finite set 2.2.2.8. Cartesian product $\subset T$ 2.2.2.9. Extensibility over finite set 2.2.2.10. Cartesian product of a set on itself
		2.2.3. Laws of Algebra on Sets	2.2.3.1. Idempotent Laws 2.2.3.2. Associative Laws 2.2.3.3. Commutative Laws 2.2.3.4. Distributive Laws 2.2.3.5. Identity Laws 2.2.3.6. Involution Law 2.2.3.7. Complement Laws 2.2.3.8. DeMorgan's Laws
	2.3. Strings	2.3.1. String basics	2.3.1.1. String variable name conventions 2.3.1.2. Alphabet set Σ 2.3.1.3. Strings over the alphabet set Σ^*
		2.3.2. String notations	

		and Properties	2.3.2.1. Empty string 2.3.2.1. String length 2.3.2.1 Concatenation 2.3.2.1. String reversal 2.3.2.1. Substring } CT
	2.4. Numbers	2.4.1. Number Representations	2.4.1.1. Number notations 2.4.1.2. Number bases
	2.5. Relations and Functions	2.5.1. Relations	2.5.1.1. Relation notation 2.5.1.2. Inverse relations 2.5.1.3. Composition of Relations 2.5.1.4. Reflexive property 2.5.1.5. Symmetric property 2.5.1.6. Anti-symmetric property 2.5.1.7. Transitive property 2.5.1.8. Atransitive property 2.5.1.9. Equivalence Relation
		2.5.2. Functions	2.5.2.1. Function notation 2.5.2.2. Domain and codomain 2.5.2.3. Dependent and independent variables 2.5.2.4. Equal functions 2.5.2.6. Cartesian coordinate system 2.5.2.7. Injective function 2.5.2.8. Surjective function 2.5.2.9. Bijective function
	2.6. Graph Theory	2.7.1. Graph Basics	2.7.1.1. Graph notations 2.7.1.2. Edges and vertices 2.7.1.3. Degree of a vertex 2.7.1.4. Graph representation 2.7.1.5. Tree and forest
		2.7.2. Graph Types	2.7.2.1. Multigraphs 2.7.2.2. Finite graphs 2.7.2.3. Complete graphs 2.7.2.4. k-regular graphs 2.7.2.5. Acyclic graph 2.7.2.6. Planar graphs
	2.7. Permutations and Combinations	2.8.1. Notations	2.8.1.1. Permutation notation 2.8.1.2. Factorial notation 2.8.1.3. Combinations notation

3. Precise Specifications (Mathematical descriptions of interfaces)	3.1. Motivation	3.1.1. Motivation for interfaces 3.1.2. Motivation for precision	3.1.1.1. Information hiding 3.1.1.2. Independent software development 3.1.2.1. Problems with informal specifications 3.1.2.2. Ease of component integration
	3.2. Specification structure	3.2.1. Specification signature 3.2.2. Usage Requirements 3.2.3. Use of math theories 3.2.4. Specification inheritance (enhancements)	3.2.1.1. Concept name 3.2.1.2. Generic parameters
	3.3. Abstraction	3.3.1. Math models for conceptualizing objects 3.3.2. Constraints 3.3.3. Trade-offs of alternative mathematical models	3.3.1.1. Booleans 3.3.1.2. Numbers 3.3.1.3. Integers 3.3.1.4. Strings 3.3.1.5. Sets 3.3.1.6. Functions 3.3.1.7. Relations 3.3.1.8. Cartesian products 3.3.1.9. Other discrete structures 3.3.1.10. Combination of the above
	3.4. Specifications of operations	3.4.1. Initialization and finalization specification 3.4.2. Operation signature 3.4.3. Pre- and post-conditions	3.4.2.1. Operation Name 3.4.2.2. Formal parameters 3.4.2.3. Return Value 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 in the post-condition
4. Modular Reasoning	4.1. Motivation	4.1.1. Motivation for Reasoning	4.1.1.1. Error detection 4.1.1.2. Code tracing/inspection 4.1.1.3. Formal Verification
		4.1.2. Motivation for Modular Reasoning	4.1.2.1. Problems with implementation to implementation coupling 4.1.2.2. Desirable coupling through contracts
	4.2. Design-by-Contract	4.2.1. Roles of clients and service providers	4.2.1.1. Specifications as external contracts 4.2.1.2. Client 4.2.1.3. Service Provider 4.2.1.4. Client implementation 4.2.1.5. Service provider implementation
		4.2.2. Construction of new components from built-in components	4.2.2.1. Implementation with arrays 4.2.2.2. Implementation with records
		4.2.3. Construction of new components using existing components	4.2.3.1. Implementation of a specification (data representation, code for operations) 4.2.3.2. Implementation of enhancement specification
	4.3. Internal contracts and Assertions	4.3.1. Internal contracts for data representations	4.3.1.1. Abstraction functions/ relations (correspondence) 4.3.1.2. Representation invariants (conventions)
		4.3.2. Assertions	4.3.2.1. Loop invariants 4.3.2.2. Progress metrics (loops and recursive procedures)
5. Correctness Proofs	5.1. Motivation	5.1.1. Meaning of correctness	5.1.1.1. Semantics 5.1.1.2. Soundness and relative completeness

		5.1.2. Motivation for proofs C*	5.1.2.1. Partial correctness 5.1.2.2. Total correctness
	5.2.Construction of verification Conditions (VCs)	5.2.1. States and abstract values of objects 5.2.2. Connection between specifications and what is to be proved 5.2.3. Types of Statements 5.2.4. Connection between induction and reasoning	5.2.1.1. Naming conventions C 5.2.2.1. Assumptions C T 5.2.2. 2. Obligations C T 5.2.3.1. Sequential statements C T 5.2.3.2. Conditional statements 5.2.3.3. Loops 5.2.3.4. Operation calls C T 5.2.4.1. Inductive case 5.2.4.2. Base case 5.2.4.3. Termination
	5.3. Proof of VCs	5.3.1. VCs as mathematical implications 5.3.2. Application of proof techniques on VCs	5.3.1.1. Givens C T 5.3.1.2. Goals C T 5.3.2.1. Direct proofs C T 5.3.2.2. Rules of inference

Reasoning Concept Inventory
The purpose of this inventory is to identify the basic set of principles that are central for analytical reasoning about correctness of software and that must be taught in undergraduate computing education.

Major Reasoning Topic	Subtopic Summary	Concept Term Highlights	Concept Details
1. Logic	1.1. Motivation	1.1.1. Motivation for Boolean Logic	
	1.2. Standard logic symbols	1.2.1. Connectives including implication	1.2.1.1. Simple statement 1.2.1.2. Connectives (NOT, AND, OR, IF... THEN, IFF) and compound statements 1.2.1.3. Truth tables 1.2.1.4. Logically equivalent statements
		1.2.2. Quantifiers	1.2.2.1. Universal quantifier 1.2.2.2. Existential quantifier
	1.3. Standard terminology	1.3.1. Proposition	1.3.1.1. Propositional variables 1.3.1.2. Compound propositions
		1.3.2. Predicate logic	1.3.1.3. Proof arguments 1.3.2.1. Predicate calculus
		1.3.3. Proof	1.3.3.1. Informal proof 1.3.3.2. Axiom 1.3.3.3. Premise 1.3.3.4. Conclusion
	1.4. Standard proof techniques	1.4.1. Supposition Deduction	
	1.5 Methods for proving	1.5.1. Direct proof 1.5.2. Proof by contradiction 1.5.3. Vacuous proof 1.5.4. Trivial proof 1.5.5. Proof by cases 1.5.6. Exhaustive proof 1.5.7. Proof by induction	
	1.6. Proof strategies	1.6.1. Forward reasoning 1.6.2. Backward reasoning	
	1.7. Rules of inference	1.7.1. Modus ponens	

		1.7.2. Simplification Conjunction 1.7.3. Universal instantiation 1.7.4. Universal generalization 1.7.5. Existential instantiation 1.7.6. Existential generalization	
2. Discrete Math Structures	2.1. Motivation	2.1.1. Motivation for Discrete Mathematics	
	2.2. Sets	2.2.1. Set basics	2.2.1.1. Set naming conventions 2.2.1.2. Universal set U 2.2.1.3. Empty set ϕ 2.2.1.4. Element naming conventions 2.2.1.5. Subset 2.2.1.6. Subset Theorem 2.2.1.7. Venn diagrams
		2.2.2. Set notations	2.2.2.1. Union 2.2.2.2. Intersection 2.2.2.3. Disjoint sets 2.2.2.4. Relative Compliment 2.2.2.5. Absolute Compliment 2.2.2.6. Power set 2.2.2.7. Power set for a finite set 2.2.2.8. Cartesian product 2.2.2.9. Extensibility over finite set 2.2.2.10. Cartesian product of a set on itself
		2.2.3. Laws of Algebra on Sets	2.2.3.1. Idempotent Laws 2.2.3.2. Associative Laws 2.2.3.3. Commutative Laws 2.2.3.4. Distributive Laws 2.2.3.5. Identity Laws 2.2.3.6. Involution Law 2.2.3.7. Complement Laws 2.2.3.8. DeMorgan's Laws
	2.3. Strings	2.3.1. String basics	2.3.1.1. String variable name conventions 2.3.1.2. Alphabet set Σ 2.3.1.3. Strings over the alphabet set Σ^*

		2.3.2. String notations and Properties	2.3.2.1. Empty string 2.3.2.1. String length 2.3.2.1 Concatenation 2.3.2.1. String reversal 2.3.2.1. Substring
	2.4. Numbers	2.4.1. Number Representations	2.4.1.1. Number notations 2.4.1.2. Number bases
	2.5. Relations and Functions	2.5.1. Relations	2.5.1.1. Relation notation 2.5.1.2. Inverse relations 2.5.1.3. Composition of Relations 2.5.1.4. Reflexive property 2.5.1.5. Symmetric property 2.5.1.6. Anti-symmetric property 2.5.1.7. Transitive property 2.5.1.8. Atransitive property 2.5.1.9. Equivalence Relation
		2.5.2. Functions	2.5.2.1. Function notation 2.5.2.2. Domain and codomain 2.5.2.3. Dependent and independent variables 2.5.2.4. Equal functions 2.5.2.6. Cartesian coordinate system 2.5.2.7. Injective function 2.5.2.8. Surjective function 2.5.2.9. Bijective function
	2.6. Graph Theory	2.7.1. Graph Basics	2.7.1.1. Graph notations 2.7.1.2. Edges and vertices 2.7.1.3. Degree of a vertex 2.7.1.4. Graph representation 2.7.1.5. Tree and forest
		2.7.2. Graph Types	2.7.2.1. Multigraphs 2.7.2.2. Finite graphs 2.7.2.3. Complete graphs 2.7.2.4. k-regular graphs 2.7.2.5. Acyclic graph 2.7.2.6. Planar graphs
	2.7. Permutations and Combinations	2.8.1. Notations	2.8.1.1. Permutation notation 2.8.1.2. Factorial notation 2.8.1.3. Combinations notation

3. Precise Specifications (Mathematical descriptions of interfaces)	3.1. Motivation	3.1.1. Motivation for interfaces 3.1.2. Motivation for precision	3.1.1.1. Information hiding 3.1.1.2. Independent software development 3.1.2.1. Problems with informal specifications 3.1.2.2. Ease of component integration
	3.2. Specification structure	3.2.1. Specification signature 3.2.2. Usage Requirements 3.2.3. Use of math theories 3.2.4. Specification inheritance (enhancements)	3.2.1.1. Concept name 3.2.1.2. Generic parameters
	3.3. Abstraction	3.3.1. Math models for conceptualizing objects 3.3.2. Constraints 3.3.3. Trade-offs of alternative mathematical models	3.3.1.1. Booleans 3.3.1.2. Numbers 3.3.1.3. Integers 3.3.1.4. Strings 3.3.1.5. Sets 3.3.1.6. Functions 3.3.1.7. Relations 3.3.1.8. Cartesian products 3.3.1.9. Other discrete structures 3.3.1.10. Combination of the above
	3.4. Specifications of operations	3.4.1. Initialization and finalization specification 3.4.2. Operation signature 3.4.3. Pre- and post-conditions	3.4.2.1. Operation Name 3.4.2.2. Formal parameters 3.4.2.3. Return Value 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 in the post-condition
4. Modular Reasoning	4.1. Motivation	4.1.1. Motivation for Reasoning 4.1.2. Motivation for Modular Reasoning	4.1.1.1. Error detection 4.1.1.2. Code tracing/inspection 4.1.1.3. Formal Verification 4.1.2.1. Problems with implementation to implementation coupling 4.1.2.2. Desirable coupling through contracts
	4.2. Design-by-Contract	4.2.1. Roles of clients and service providers 4.2.2. Construction of new components from built-in components 4.2.3. Construction of new components using existing components	4.2.1.1. Specifications as external contracts 4.2.1.2. Client 4.2.1.3. Service Provider 4.2.1.4. Client implementation 4.2.1.5. Service provider implementation 4.2.2.1. Implementation with arrays 4.2.2.2. Implementation with records 4.2.3.1. Implementation of a specification (data representation, code for operations) 4.2.3.2. Implementation of enhancement specification
	4.3. Internal contracts and Assertions	4.3.1. Internal contracts for data representations 4.3.2. Assertions	4.3.1.1. Abstraction functions/ relations (correspondence) 4.3.1.2. Representation invariants (conventions) 4.3.2.1. Loop invariants 4.3.2.2. Progress metrics (loops and recursive procedures)
5. Correctness Proofs	5.1. Motivation	5.1.1. Meaning of correctness	5.1.1.1. Semantics 5.1.1.2. Soundness and relative completeness

		5.1.2. Motivation for proofs ✓	5.1.2.1. Partial correctness 5.1.2.2. Total correctness
	5.2.Construction of verification Conditions (VCs)	5.2.1. States and abstract values of objects ✓ 5.2.2. Connection between specifications and what is to be proved T 5.2.3. Types of Statements 5.2.4. Connection between induction and reasoning	5.2.1.1. Naming conventions ✓ T 5.2.2.1. Assumptions } T 5.2.2. 2. Obligations 5.2.3.1. Sequential statements ✓ T 5.2.3.2. Conditional statements 5.2.3.3. Loops 5.2.3.4. Operation calls 5.2.4.1. Inductive case 5.2.4.2. Base case 5.2.4.3. Termination
	5.3. Proof of VCs	5.3.1. VCs as mathematical implications ✓ 5.3.2. Application of proof techniques on VCs ✓	5.3.1.1. Givens] ✓ 5.3.1.2. Goals 5.3.2.1. Direct proofs ✓ 5.3.2.2. Rules of inference ✓

✓ = covered in class

A = " " + assignment

T = " " + test (quiz, midterm, or final)

Appendix I

Basic Reasoning Principles Survey

Part I

Please consult the Basic Reasoning Inventory document for related terms. Focusing only on what these reasoning principles are (and not on how they might or should be taught), state your agreement with the following statements.

1. CS students need to understand how to use Boolean logic not only for understanding how computers work, but for establishing correctness of programs.

strongly disagree	disagree	moderately disagree	moderately agree	Agree	strongly agree
○	○	○	○	○	○

2. CS students need to understand and use standard logic symbols (including implication).

strongly disagree	disagree	moderately disagree	moderately agree	Agree	strongly agree
○	○	○	○	○	○

3. CS students need to understand and use sets and related notations (e.g., subsets, unions, and intersections).

strongly disagree	disagree	moderately disagree	moderately agree	Agree	strongly agree
○	○	○	○	○	○

4. CS students need to understand and use the notion of strings over a given alphabet and the sets associated with such strings.

strongly disagree	disagree	moderately disagree	moderately agree	Agree	strongly agree
○	○	○	○	○	○

5. CS students need to understand the connections between software specifications and basic discrete math structures, such as sets, strings, integers and other number systems, relations, and functions.

strongly disagree	disagree	moderately disagree	moderately agree	Agree	strongly agree
○	○	○	○	○	○

6. CS students need to understand the distinct roles of clients (users) and implementers of components, and the use of interfaces.

strongly disagree	disagree	moderately disagree	moderately agree	Agree	strongly agree
○	○	○	○	○	○

7. CS students need to understand pre- and post-conditions of operations.

strongly disagree	disagree	moderately disagree	moderately agree	agree	strongly agree
○	○	○	○	○	○

8. CS students need to understand formal descriptions of pre- and post-conditions of operations.

strongly disagree	disagree	moderately disagree	moderately agree	agree	strongly agree
○	○	○	○	○	○

9. CS students need to understand internal code assertions such as class representation invariants and loop invariants.

strongly disagree	disagree	moderately disagree	moderately agree	agree	strongly agree
○	○	○	○	○	○

10. CS students need to understand formal descriptions of internal code assertions such as class representation invariants and loop invariants.

strongly disagree	disagree	moderately disagree	moderately agree	agree	strongly agree
○	○	○	○	○	○

11. CS students must be familiar with at least one formal specification language.

strongly disagree	disagree	moderately disagree	moderately agree	agree	strongly agree
○	○	○	○	○	○

12. CS students must understand the need for precise (mathematical) specifications for software in order to reason about the software and establish its correctness.

strongly disagree	disagree	moderately disagree	moderately agree	agree	strongly agree
○	○	○	○	○	○

13. CS students need to understand the concept of modular reasoning, which allows for individual components to be certified as correct without a need to re-verify when those components are placed in a larger program.

strongly disagree	disagree	moderately disagree	moderately agree	agree	strongly agree
○	○	○	○	○	○

14. CS students need to know that testing a program cannot verify its correctness.

strongly disagree	disagree	moderately disagree	moderately agree	agree	strongly agree
○	○	○	○	○	○

15. CS students need to understand how to reason about correctness of straight line code (no branches) formally.

strongly disagree	disagree	moderately disagree	moderately agree	agree	strongly agree
○	○	○	○	○	○

16. CS students need to understand how to reason about termination formally.

strongly disagree	disagree	moderately disagree	moderately agree	agree	strongly agree
○	○	○	○	○	○

17. CS students need to understand how to reason about correctness of code involving loops (using invariants) formally.

strongly disagree	disagree	moderately disagree	moderately agree	agree	strongly agree
○	○	○	○	○	○

18. CS students need to understand how to reason about recursive code formally.

strongly disagree	disagree	moderately disagree	moderately agree	agree	strongly agree
○	○	○	○	○	○

19. CS students should be aware that correctness of a component can be reduced to proving a set of mathematical verification conditions.

strongly disagree	disagree	moderately disagree	moderately agree	agree	strongly agree
○	○	○	○	○	○

20. CS students need to be able to apply their proof techniques from Boolean logic, such as induction, modus ponens, etc. to the challenge of proving the verification conditions (VC's) generated from the specifications.

strongly disagree	disagree	moderately disagree	moderately agree	agree	strongly agree
○	○	○	○	○	○

21. CS students must be familiar with at least one tool for mathematical reasoning.

strongly disagree	disagree	moderately disagree	moderately agree	agree	strongly agree
○	○	○	○	○	○

Part II

Please indicate which principles you think should be a part of particular courses in the curriculum. The numbers are associated with the list of 6 principles. Check all that apply.

1. Beginning Programming (CS1)
   ○ 1 ○ 2 ○ 3 ○ 4 ○ 5 ○ 6

2. Intermediate Programming (CS2)
   ○ 1 ○ 2 ○ 3 ○ 4 ○ 5 ○ 6

3. Discrete Math
   ○ 1 ○ 2 ○ 3 ○ 4 ○ 5 ○ 6

4. Data Structures & Algorithms
   ○ 1 ○ 2 ○ 3 ○ 4 ○ 5 ○ 6

5. Theory of Programming Languages
   ○ 1 ○ 2 ○ 3 ○ 4 ○ 5 ○ 6

6. Software Engineering
   ○ 1 ○ 2 ○ 3 ○ 4 ○ 5 ○ 6

7. Theory of Computation
   ○ 1 ○ 2 ○ 3 ○ 4 ○ 5 ○ 6

8. AI
   ○ 1 ○ 2 ○ 3 ○ 4 ○ 5 ○ 6

Should any additional principles be a part of the CS curriculum? Please list them in the order of your priority.

What other courses might be appropriate for introducing mathematical reasoning principles?

Please write any comments you have about teaching mathematical reasoning in the CS curriculum.

Appendix J

End of Workshop Feedback (SIGCSE 2012)

- Read each of the workshop topic areas in the list below.
- Circle 0 - 5 to indicate how much you knew about each topic at the start of the workshop.
- Circle 0 - 5 to indicate how much you know now at the end of the workshop.

	Workshop Topic	You Knew at the Start 0 = A Little 5 = A Lot	You Knew at the End 0 = A Little 5 = A Lot
1	The meaning of *design (or programming) by contract*, and the responsibilities of callers and implementers of operations.	0 1 2 3 4 5	0 1 2 3 4 5
2	That a component (or class) interface describes abstractly what it does but not concretely how it does it.	0 1 2 3 4 5	0 1 2 3 4 5
3	How to use mathematical types (e.g., sets, functions, strings, mathematical integers) to mathematically model computing types such as stacks, queues, computing integers, etc.	0 1 2 3 4 5	0 1 2 3 4 5
4	How to write an operation's pre and post conditions based on a mathematical type, e.g., mathematical strings, sets, functions, etc.	0 1 2 3 4 5	0 1 2 3 4 5
5	That mathematical *strings* (with notations the empty string, and the length and the concatenate operators) are suitable for describing a variety of programming concepts.	0 1 2 3 4 5	0 1 2 3 4 5
6	The distinction between mathematical and computational integers as it applies to the proof process.	0 1 2 3 4 5	0 1 2 3 4 5
7	How to prove correctness of a piece of code using a *reasoning table*, in conjunction with each operation's *requires* and *ensures* clauses.	0 1 2 3 4 5	0 1 2 3 4 5
8	That the client program is responsible for guaranteeing that the *requires* clause to hold prior to the call.	0 1 2 3 4 5	0 1 2 3 4 5

9	The connection between an inductive proof in mathematics and the proof of a recursive implementation.	0 1 2 3 4 5	0 1 2 3 4 5
10	Use of loop invariants in proving correctness of iterative code.	0 1 2 3 4 5	0 1 2 3 4 5
11	That termination of loops and recursion can be proved formally using progress metrics (decreasing clauses).	0 1 2 3 4 5	0 1 2 3 4 5
12	What a verification condition (VC) is and how to prove a VC.	0 1 2 3 4 5	0 1 2 3 4 5

With respect to teaching mathematical reasoning to your students, list by topic number:

- What you believe to be the three *most* important topics:

- What you believe to be the three *least* important topics:

- Which topics you will attempt to incorporate into your teaching this coming academic year:

- For which topics you most need additional instructional materials:

In which computer science classes do you believe these mathematical reasoning concepts can be taught? (*For example*: CS1, CS2, advanced data structures, computer science discrete math, analysis of algorithms, etc.)

Appendix K

Results from Attitudinal Assessments in the Professional Community

Two types of attitudinal surveys were conducted to discover what the professional community of computer science educators thinks about the usefulness of the Reasoning Concept Inventory.

Survey to Determine if the RCI matches the Expectations of the Professional Community

A 21-question survey was administered to eleven professors with a variety of teaching experiences from different institutions, mostly four year liberal arts colleges. The survey's goal was to discover how well the principles of the reasoning concept inventory match the expectations of these reasoning experts and educators. A 6-point scale was used, ranging from 1 (strongly agree) to 6 (strongly disagree). A number of observations along with some survey questions are presented below. They were initially addressed in [280], and are reproduced below.

Observation #1

The statements ranged from the most basic, such as:

> **Q#1:** CS students need to understand how to use Boolean logic not only for understanding how computers work, but for establishing correctness of programs.

To the more advanced, such as:

Q#10: CS students need to understand formal descriptions of internal code assertions, such as class representation invariants and loop invariants.

In Question#1 (Q#1) 10 out of 11 respondents agreed or strongly agreed with the statement. On Q#2, there was still agreement from 10 out of 11 respondents. There was, however, a shift away from strong agreement in Q#2; only 5 of them agreed or strongly agreed, whereas 6 agreed only moderately.

Clearly the time necessary to cover the latter topics is greater than the first (loop invariants are discusses later in this section). Teaching the types of representation invariants necessary to prove the correctness of data abstraction implementations is certainly non-trivial. In a software engineering course, students are introduced to formal assertions of representation invariants and abstraction functions, but were not taught to reason about the correctness of an implementation using those assertions. In other words, ―understanding" of a representation invariant may have strong agreement, whereas ―application" of the principle may have less agreement.

Observation #2

Another set of questions concern the verification of code involving loops and recursion. Representative examples include:

Q#16: CS students need to understand how to reason about termination formally.

Q#17: CS students need to understand how to reason about correctness of code involving loops (using invariants) formally.

> **Q#18:** CS students need to understand how to reason about recursive code formally.

Two out of 11 respondents disagreed with all three questions, perhaps because the statements (by design) insisted on formality. The others agreed with the statements. The level of agreement was highest for the termination question. Also, there was slightly more agreement about teaching reasoning in relationship to loops, relative to reasoning about recursion. This is possibly because educators often feel that students encounter more iterative code in practice than recursive code.

Observation #3

The concept of modular reasoning and the use of interfaces in reasoning were explored in two questions:

> **Q#6:** CS students need to understand the distinct roles of clients (users) and implementers of components, and the use of interfaces.

> **Q#13:** CS students need to understand the concept of modular reasoning, which allows for individual components to be certified as correct without a need to re-verify when those components are placed in a larger program.

The level of agreement for both questions was uniformly high; about half the respondents strongly agreed with both statements.

Observation #4

Convinced of the importance of formal reasoning (observation #2) and modular reasoning (observation #3), both dependent on component specifications, respondents were also in agreement on the importance of teaching specifications. There was a slight shift in the level of agreement when –formality" was introduced, but agreement was still high. Specifically, responses are considered to the following two questions, which differ only in their formality requirement.

> **Q#7:** CS students need to understand pre- and post-conditions of operations.
>
> **Q#8:** CS students need to understand descriptions of pre- and post-conditions of operations.

For both questions, all respondents were in agreement. For the first, 5 respondents agreed strongly with the statement, and 5 agreed. For the second, 3 respondents agreed strongly, 4 agreed, and 4 agreed moderately. The responses were similar when asked about loop and representation invariants.

Observation #5

Some questions were designed to measure the perceived importance of directly connecting mathematics and computer science. One question focused on making this connection through specifications, whereas another focused on making this connection by applying logical reasoning principles in dispatching the verification conditions that arise in proofs of code correctness:

Q#2: CS students need to understand the connections between software specifications and basic discrete math structures, such as sets, strings, integers and other number systems, relations, and functions.

Q#20: CS students need to be able to apply their proof techniques from Boolean logic, such as induction, modus ponens, etc., to the challenge of proving the verification conditions (VCs) generated from the specifications.

All respondents agreed with the first statement; 9 of 11 agreed or strongly agreed. Two disagreed with the second statement. The agreement level among other respondents shifted towards moderate agreement. These responses are attributed to the difficulty of teaching the associated reasoning exercises without suitable reasoning tools. More precisely, in the absence of widely available tools accessible to undergraduate CS students, the task of generating and proving verification conditions can be difficult.

Observation #6

The survey also asked educators to provide feedback on where the six principles might be taught in the computing curriculum. They were given eight course subjects and asked to identify the principles that should be included as part of each course. The course subjects are: Beginning Programming (CS1), Intermediate Programming (CS2), Discrete Math, Data Structures and Algorithms, Theory of Programming Languages, Software Engineering, and Theory of Computation. The results are summarized in Figure 31.

The most important observation is that respondents generally agreed that the reasoning principles are relevant to a broad range of computer science courses. Indeed,

approximately half of the respondents indicated that principles 1 and 2 should be included in at least half of the courses. Further, with the exception of CS1 and AI, each course was associated with at least 4 of the reasoning principles. Data structures and algorithms and software engineering, in particular, were identified as excellent candidates for integrating a majority of the identified reasoning principles.

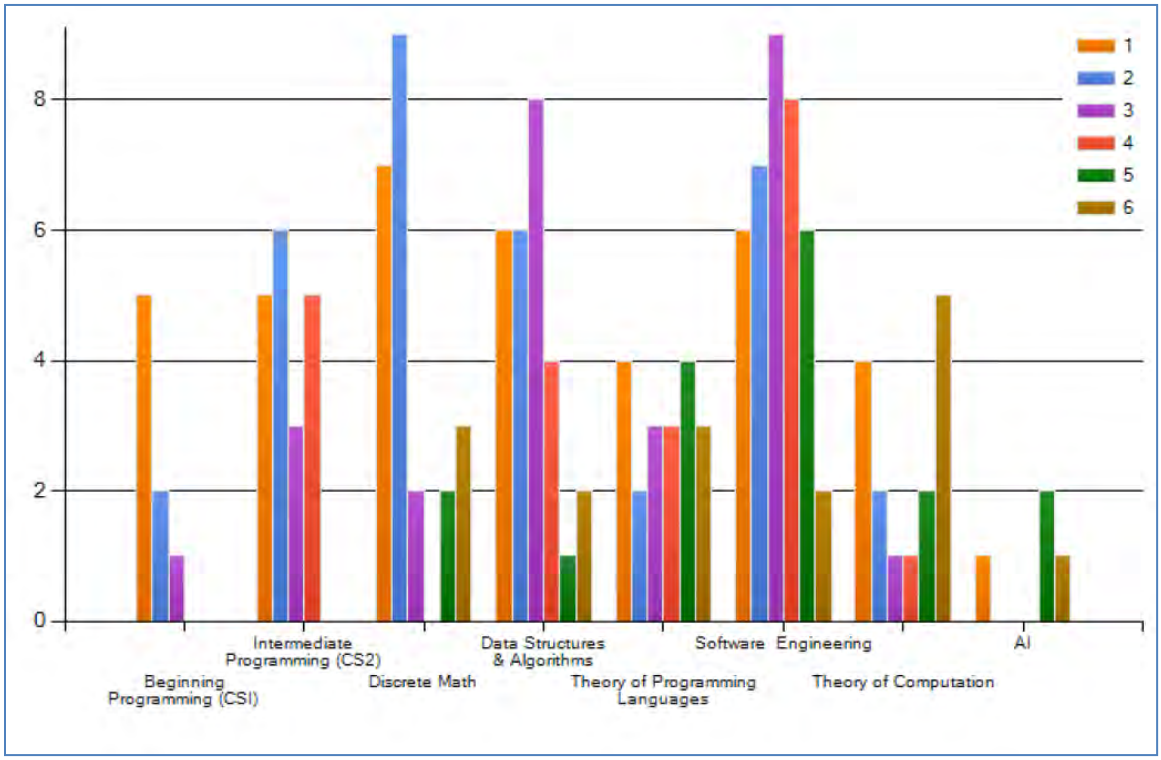


Figure 31. Which principles should be a part of particular courses in the curriculum? (survey results)

SIGSCE 2012 Computer Science Educators' Survey

Another community survey was conducted after the reasoning workshop at the SIGSCE 2012 conference. The workshop's goal was to introduce the participants to a small number of reasoning topics and to demonstrate how they can be taught in the

classroom. At the end of the two and a half hour workshop the participants were asked to indicate on the scale from 0 (a little) to 5 (a lot) how their knowledge level has changed by the end of the workshop for each of the twelve topics. Among other questions, they were also asked to select the most important topics. The full survey is available in Appendix J. The data summary is presented in Table 32[2]. The left column lists each of the twelve topic numbers that correspond to the twelve questions of the survey. The column labeled –Average Change" indicates the average change in the participants' knowledge for each of the topics by the end of the workshop. The maximum change is 5, as in the case where participant who knew a little on the topic before the workshop (0), learned a lot by the end of the workshop (5). –No Change" column is the number of participants without any knowledge change, and the last column provides the number of workshop attendees who selected particular topics as the most important.

For example, topic#7 (How to prove correctness of a piece of code using a *reasoning table*, in conjunction with each operation's *requires* and *ensures* clauses) indicates the highest average knowledge change among the participants, with only one attendee having learned nothing new, and three of them selecting this topic as one of the most important.

Topic #	Average Change	No Change	Most Important
1	1.2	2	3
2	0.7	4	3
3	1.5	0	1

[2] The data analysis and the survey questions were provided by Dr. Hollingsworth, Indiana University SouthEast.

4	1.0	3	0
5	1.7	1	0
6	0.8	3	0
7	2.3	0	2
8	0.3	4	2
9	0.0	6	0
10	0.0	6	3
11	0.0	6	1
12	2.3	0	1

Table 32. Results of the SIGSCE 2012 Workshop Survey

The charts in Figure 32 and Figure 33 depict the average change of the participants' knowledge by topic, and the count for each topic selected as most important.

The highest average change in knowledge occurred for the topics 7 (How to prove correctness of a piece of code using a *reasoning table*, in conjunction with each operation's *requires* and *ensures* clauses) and 12 (What a verification condition (VC) is and how to prove a VC). No knowledge change occurred for the topics 9 (The connection between an inductive proof in mathematics and the proof of a recursive implementation), 10 (Use of loop invariants in proving correctness of iterative code), and 11 (That termination of loops and recursion can be proved formally using progress metrics (decreasing clauses)).

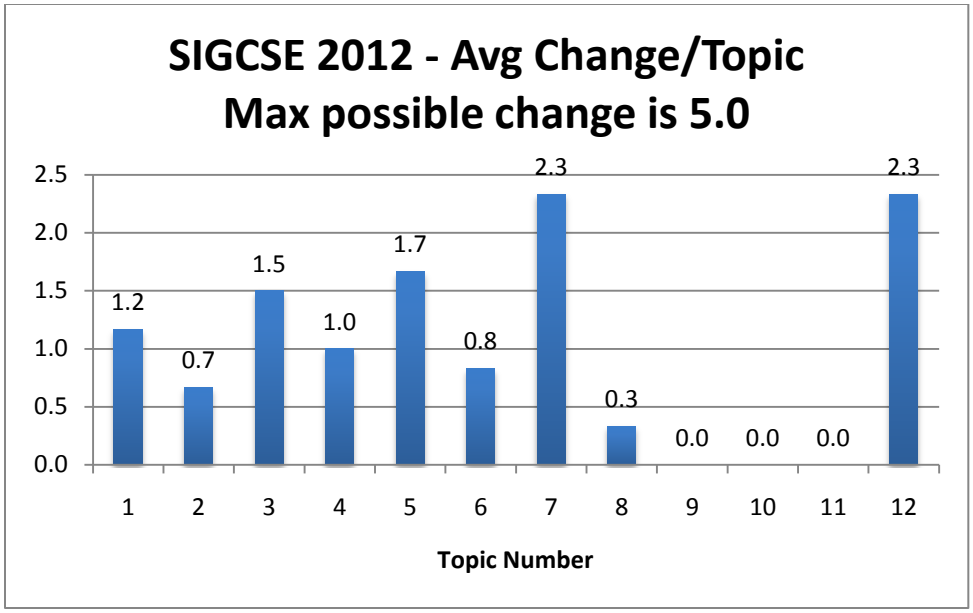
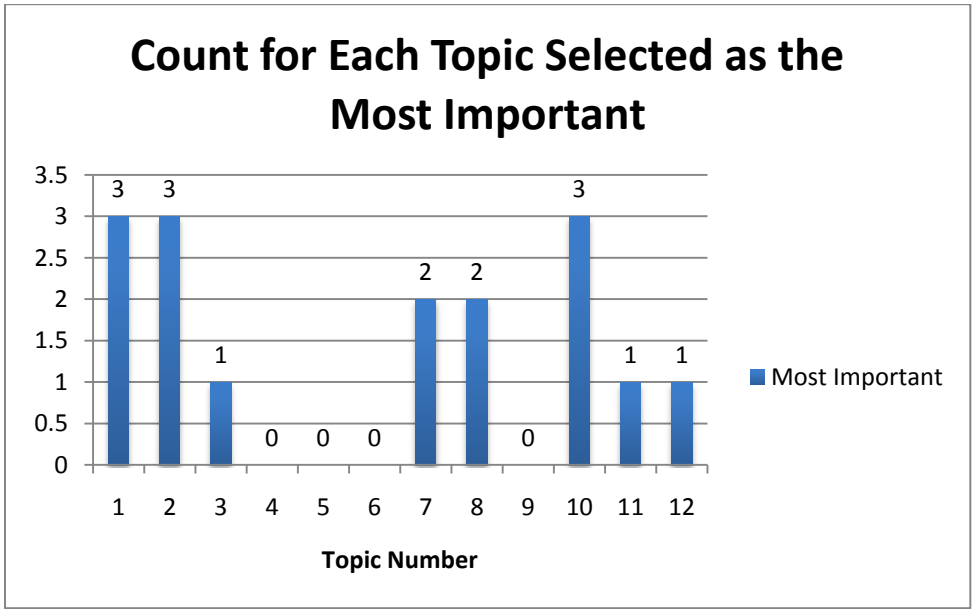


Figure 32. Average knowledge change by topic (SIGSCE 2012 workshop survey)



-

Figure 33. Topics indicated as most important (SIGSCE 2012 workshop survey)

Three of the topics: 1 (The meaning of *design (or programming) by contract*, and the responsibilities of callers and implementers of operations), 2 (That a component (or class) interface describes abstractly what it does but not concretely how it does it), and 10 (Use of loop invariants in proving correctness of iterative code) got the highest counts of being the most important.

Appendix L

Learning Outcomes and Sample Exercises for a Subset of RCI Topics

RCI# 3. PRECISE SPECIFICATIONS
RCI# 3.1. Motivation
RCI# 3.1.1. Motivation for interfaces
RCI# 3.1.1.1. *Information hiding*

KC	**LO:** **#1. Define the principle of information hiding.** **#2. State how information hiding supports software design.** **#3. State the role of information hiding in object interface design.** **#4. Identify programming language features that support information hiding.** **Exercise 1:** For each of the statements below indicate whether they are true or false: T/F Clients need to know only interface, not implementation; T/F Implementations capture decisions likely to change; T/F Interfaces capture decisions likely to change. **Exercise 2:** In 1972 Professor D. Parnas wrote: ―Information hiding is perhaps the most important intellectual tool developed to support software design.‖ State what information hiding is, and why it is important in software engineering.

RCI# 3.1.1.2. *Independent software development*

KC	**LO: Summarize how interfaces enable independent software development.** **Exercise 1:** Explain why it is important for components developed by different software engineers to adhere to formal specification.

RCI# 3.1.2. Motivation for precision
RCI# 3.1.2.1. *Problems with informal specifications*

KC	**LO:** **#1. Outline the drawbacks of informal specifications.** **#2. List the advantages of formal specifications.** **Exercise 1:** a. Explain what problems can arise from using informal specifications to specify a

	software component. b. Name 5 good characteristics of a well-designed formal specification. c. Name 2 specification languages. **Exercise 2:** Indicate if the statement below is true or false T/F The language of mathematics is best suited for modeling and descriptions of modules, because of its precision, expressiveness, and common knowledge of the notations and terms.

RCI# 3.1.2.2. ***Ease of component integration***

KC	**LO: State how using precise formal specifications affect the processes of component integration and team development.**

RCI# 3.2. Specification structure
RCI# 3.2.1. Specification signature
RCI# 3.2.1.1. ***Concept name***
RCI# 3.2.1.2. ***Generic parameters***

KC	**LO: Define a specification signature and name its syntactic elements.**

RCI# 3.2.3. Use of math theories

KC	**LO:** **#1. Distinguish a concept specification from an interface description.** **#2. Define the central role of abstraction in concept specification.** **#3. Define the central role of mathematical modeling for abstraction.** **#4. Give an example of how a mathematical theory is useful for mathematical modeling in software engineering.** **#5. State how a math theory is used when developing a new component specification.**

RCI# 3.2.4. Specification inheritance (enhancements)

KC	**LO:** **#1. Define what is a specification inheritance (enhancement).** **#2. Give an example of how enhancements are useful.**

	Exercise 1: Explain what specification inheritance (enhancement) is and give an example. **Exercise 2:** Explain what is the benefit of writing an enhancement that uses existing operations instead of including an additional operation into a template specification?
AA	**LO:** **#1. Demonstrate how an existing enhancement can extend the functionality of the component.** **#2. State the benefits of specification inheritance for team development as compared with implementation inheritance.**
SE	**LO: Write an enhancement that extends the functionality of a component.** **Exercise 1:** Write an enhancement specification Flipping_Capability for Stack_Template that allows to flip a Stack.

RCI# 3.3. Abstraction
RCI# 3.3.1. Math models for conceptualizing objects
RCI# 3.3.1.1. *Booleans*

KC	**LO: Identify a software component behavior that can be modeled by mathematical booleans.**

RCI# 3.3.1.2. *Numbers*

KC	**LO: Identify a software component specification that can be modeled by numbers in mathematics.**

RCI# 3.3.1.3. *Integers*

KC	**LO: State how computer integers can be modeled by mathematical integers.** **Exercise 1:** What is the main difference between Integers in mathematics and the Integers we use in programming?

RCI# 3.3.1.4. ***Strings***

KC	**LO: Identify a software component specification that can be modeled by mathematical strings.** **Exercise 1:** For each of the components below please indicate if they can be modeled by a mathematical string. Y/N Queue Y/N Stack Y/N Map Y/N Linked List Y/N Integer **Exercise 2:** Can a mathematical string be used to model a Queue? Explain your answer.
AA	**LO: Apply basic string properties and operations to strings.** **Exercise 1**: For the statements below, please indicate whether they are true or false. T/F For all α: String(α), Reverse (Prt_Between(0, | α |, α)) = α o λ; T/F If (Reverse (α o β) = Reverse (β) o <x> then α = <x>; T/F Prt_btwn (| a |, | a | + 1, a o <x>) = Prt_btwn (0, 1, <x> o a) = <a>; T/F Prt_btwn (m, n, Reverse(α)) = Reverse(Prt_btwn (| α | - n, | α | - m, α)); T/F If α = <x> then α ≠ Λ; T/F |α o Λ| < |α o <x>|; T/F If α = β and |α o γ| = |γ| + 1 then β = Λ; T/F If α = <x> o <y> then |α| = 2 * |<x>|; T/F Reverse(α o <x>) = <x> o Reverse(α); T/F Reverse(α o <x>) = Reverse(<x> o α); T/F |Reverse(α o β)| ≠ |Reverse(α)| + |Reverse(β)|; T/F If Reverse(α o β) = Reverse(β) o <x> then α = <x>;
SE	**LO: Write code using advances string properties.**

RCI# 3.3.1.5. ***Sets***

KC	**LO: Give an example of a software component specification that can be modeled by a mathematical set.** **Exercise 1:** Can a mathematical set be used to model a Stack? Explain your answer.

RCI# 3.3.1.6. ***Functions***
RCI# 3.3.1.7. ***Relations***
RCI# 3.3.1.8. ***Cartesian products***
RCI# 3.3.1.9. ***Other discrete structures***
RCI# 3.3.1.10. ***Combination of the above***

KC	**LO: Determine which software component behaviors can be modeled by various mathematical models.** **Exercise 1:** Match which components can be modeled by the following mathematical models. You can draw lines. a. Queue — integer b. Stack — string c. Map — set d. Linked List — function e. Integer — cartesian product

RCI# 3.3.2. Constraints

KC	**LO: Define the role of constraints and give an example.**

RCI# 3.3.3. Trade-offs of alternative mathematical models

SE	**LO: Evaluate the trade-offs of mathematical models used for conceptualizing the behavior of a software component.** **Exercise 1:** You are given two mathematical models: a string and a set. Explain which one you will select to model a queue, and why.

RCI# 3.4. Specification of operations
RCI# 3.4.1. Initialization and finalization specification

KC	**LO: State the role of initialization and finalization specifications.**
AA	**LO: Analyze a piece of code to determine if initialization and finalization specifications are satisfied.**
SE	**LO: Write code that satisfies initialization and finalization specifications.**

RCI# 3.4.2. Operation signature
RCI# 3.4.2.1. *Operation Name*
RCI# 3.4.2.2. *Formal Parameters*
RCI# 3.4.2.3. *Return Value*

KC	**LO: Identify the elements of an operation signature.**

RCI# 3.4.3. Pre- and post-conditions
RCI# 3.4.3.1. *Specification parameter modes*

KC	**LO:** **#1. State the purpose of operation parameter modes.** **#2. State the meaning of various parameter modes.** **Exercise 1:** Circle all the true statements with respect to the operation parameter modes. T/F clears - returns the variable to its initial state as specified in the initialization clause of that type; T/F alters - the outgoing value is altered in the way specified in the ensures clause; T/F evaluates - the outgoing value of the incoming variable is unchanged; T/F updates - the outgoing value is slightly different from the incoming value; T/F preserves - the value of the incoming variable was changed as specified in the ensures clause; T/F restores expects an expression; **Exercise 2:** 1. Explain what an operation parameter mode is, and why it is useful. 2. What happens to the outgoing value of the incoming variable when we use the parameter mode [*alters*] (use parameter mode of instructor's choice)?

AA	**LO: Determine the result of an operation's execution by examining operation parameter modes.**
	
	Exercise 1:
	Determine which statement is true with respect to the code below:
	
	`Num  = 3;`
	`Num2 = 7;`
	`Stk  = <1, 2, 5>;`
	
	`Operation Guess(preserves Num: Integer;`
	`alters Num2: Integer;`
	`updates Stk: Stack);`
	`ensures Stk = #Stk o #Num2;`

	a. Num may be modified by the operation, but the final value is 3;
	b. Num = 3, S = < 1, 2, 5, 7 >, Num2 is some integer;
	c. Stk will be updated in some way, Num will be 7;
	d. Num2 is still 7 after the operation completes;
	
	Exercise 2:
	Determine the values of S1 and S2 after operation AppendStack executes.
	
	`S1 = <5, 7, 9>; S2 = <11, 13>;`
	
	`Operation AppendStack(updates S1: Stack;`
	`clears S2: Stack);`
	`requires |S1| + |S2| <= Max_Depth;`
	`ensures S1 = #S1 o Reverse(#S2);`
	
	a. S1 is updated as stated in the ensures clause, and S2 = λ;
	b. S1 is updated as stated in the ensures clause, and S2 = <0>;
	c. S1 will be updated in some way, S2 is unknown;

SE	**LO: Produce operation specifications using appropriate operation parameter modes.**
	
	Exercise 1:
	
	Choose correct parameter modes for the specification of operation RetrieveFirst for Stack Template. The operation reverses a stack, removes the first element, and reverses it again.
	
	`Operation RetrieveFirst (???? S1: Stack, ???? R: Entry)`
	`requires |S| >= 1;`
	`ensures There exists T1: Stack, such that`
	`T1 = <R> o Reverse(#S1) and S1 = Reverse(T1);`

RCI# 3.4.3.2. ***Responsibility of the caller***

KC	**LO: State the responsibilities of the caller of an operation with respect to its pre-condition.** **Exercise 1:** Define the term pre-condition. **Exercise 2:** Circle all the true statements with respect to the pre-conditions. a. Pre-condition is the responsibility of the caller. b. Pre-condition is the responsibility of the implementer. c. Pre-condition is the responsibility of both the caller and the implementer.

AA	**LO: Analyze a piece of code to determine if the operation's pre-condition is violated.**

SE	**LO: Write an operation specification with a suitable pre-condition.**
	
	Exercise 1:
	Examine the operation code and its specification, and provide a suitable pre-condition.
	
	`Operation Multiply(alters I, J: Integer; replaces K: Integer);`
	`  requires  ?????`
	`  ensures K = #I * #J;`
	
	`  Procedure`
	`  While (Is_Not_Zero(I))`
	`  changing I, K;`
	`  maintaining K = (#I - I) * #J;`
	`  decreasing I;`
	`  do`
	`  K := Sum(K, J);`
	`  Decrement(I);`
	`  end;`
	`  end;`
	`end Multiply;`
	
	Exercise 2:
	
	Examine the operation code and its specification, and provide a suitable pre-condition.
	
	`Operation Remove_Last(updates Q: Queue; replaces E: Entry);`
	`  requires ?????`
	`  ensures #Q = Q o <E>;`
	
	`  Procedure`
	`  Var T: Queue;`
	`  Dequeue (E, Q);`
	
	`  While (Length(Q) /= 0)`
	`  changing Q,T,E;`
	`  maintaining #Q = T o <E> o Q;`
	`  decreasing |Q|;`
	`  do`
	`  Enqueue(E,T);`
	`  Dequeue(E,Q);`
	`  end;`
	`  Q :=: T;`
	`  end;`
	`end Remove_Last;`

RCI# 3.4.3.3. *Responsibility of the implementer*

<table>
<tr><td>KC</td><td>LO: State the responsibilities of the implementer of an operation with respect to its post-condition.

<u>Exercise 1:</u>
Define an operation’s post-condition.

<u>Exercise 2:</u>
Circle all the true statements with respect to post-conditions:

a. Post-condition is the responsibility of the implementer.
b. Post-condition is the responsibility of the caller.
c. Post-condition is the responsibility of both the caller and the implementer.</td></tr>
<tr><td>AA</td><td>LO: Analyze a piece of code to determine if the operation’s post-condition is violated.

<u>Exercise 1:</u>
Select valid test points from the list below to show your understanding of the following specifications.

Operation Mystery3 (restores I: Integer): Boolean;
requires: I > 1;
ensures: Mystery3 = (for all J, K: N, J * K = I
implies J = 1 or K = 1);

test set 1:
inputs: I = 1;
outputs: Mystery3 = True;

test set 2:
inputs: I = 5;
outputs: Mystery3 = True;

test set 3:
inputs: I = 10;
outputs: Mystery3 = False;

test set 4:
inputs: I = 15;
outputs: Mystery3 = True;

<u>Exercise 2:</u>
Give two test points to show your understanding of the following specifications:

Operation Mystery2 (updates Q: Queue; evaluates I: Integer);</td></tr>
</table>

	requires 0 <= n <= |Q| **ensures There exists** α: Str(Entry) **such that** |α| = n **and** #Q = α ○ Q;
SE	**LO: Write an operation specification with suitable post-condition.** **Exercise 1:** Write the ensures clause to capture the behavior of code precisely. **Operation** Add_to (**updates** i: Integer; **evaluates** j: Integer); **requires** min_int <= i + j **and** i + j <= max_int **and** j >= 0; **ensures** ???? **Procedure** (**updates** i:Integer; **evaluates** j:Integer); **While** (**not** Is_Zero(j)) **changing** i, j; **maintaining** (i + j = #i + #j) **and** j >= 0; **decreasing** j; **do** Increment (i); Decrement (j); **end;** **end;** **end** Add_to; **Exercise 2:** Write the ensures clause to capture the behavior of code precisely. **Operation** Mystery_3 (**updates** S: Sequence); **requires** 1 <= |S| **ensures** ????? **Procedure** Var E: Entry; Remove_After (0, E, S); Insert_After (Length(S), E, S); **end;** **end** Mystery_3;

RCI# 3.4.3.4. *Equivalent specifications*

KC	**LO:** **#1. Define what is an equivalent specification.** **#2. State if equivalent specifications are desirable.** **Exercise 1:** Explain what is an equivalent specification and give an example.

	Exercise 2: Indicate if the statement below is true or false. T/F In general, it is possible to write the same specification in several different ways.

AA	**LO: Identify an equivalent specification for a given operation.**
	
	Exercise 1:
	Identify which of the three specifications given below are equivalent to the specification of the operation.
	
	`Operation Do_Nothing(restores I: Integer);`
	`  requires (I + 1 <= max_int);`
	
	a. `requires I + 2 <= max_int + 1;`
	b. `requires I <= max_int - 1;`
	c. `requires I + 1 < max_int - 1 ;`
	
	Exercise 2:
	Identify which of the three specifications given below are equivalent to the specification of the operation.
	
	`Operation PushOne( alters E: Entry; updates S: Stack);`
	`  requires: |S| < Max_Depth;`
	`  ensures  S = <#E> o #S;`
	
	a. `requires |S| <= Max_Depth + 1;`
	b. `requires |S| + 1 < Max_Depth +1;`
	c. `requires |S| <= Max_Depth;`

SE	**LO: Write an operation and provide equivalent specifications.**

RCI# 3.4.3.5. *Redundant specifications*

KC	**LO:** **#1. Define what is a redundant specification.** **#2. State why redundant specifications are not desirable.** **Exercise 1:** Explain what redundant specification is and give an example.

AA	**LO: Identify an operation with redundant specifications.**
	
	Exercise 1:
	Determine if the specification below is redundant.
	
	`Operation Dequeue(replaces R: Entry; updates Q: P_Queue);`
	`  requires |Q| /= 0;`
	`  ensures #Q = <R> o Q and |Q| = |#Q| - 1;`

SE	**LO: Write an operation and provide a non-redundant specification.**
	
	Exercise 1:
	Determine if the specification below is redundant. If so, then rewrite it so that it does not contain redundancy.
	
	`Operation Dequeue(replaces R: Entry; updates Q: P_Queue);`
	`  requires |Q| /= 0;`
	`  ensures #Q = <R> o Q and |Q| = |#Q| - 1;`

RCI# 3.4.3.6. ***Notation to distinguish an incoming value in the post-condition***

KC	**LO: State why a special notation to designate the incoming value is only used in post-condition of an operation.**

RCI# 4. MODULAR REASONING
RCI# 4.1. Motivation
RCI# 4.1.1. Motivation for reasoning
RCI# 4.1.1.1. *Error Detection*
RCI# 4.1.1.2. *Code tracing/inspection*

KC	**LO:** **#1. State the goals of code tracing/inspection.** **#2. State how code tracing/inspection differs from verification.** **Exercise 1:** For the statements below please indicate whether they are true or false. T/F Tracing (or formal inspection) is an approach that combines the principles of testing and verification; T/F In tracing, an implementation is analyzed statically on a sample set of valid inputs; T/F Tracing involves execution of the implementation; T/F Tracing helps check the logical correctness of the implementation on selected inputs. **Exercise 2:** What is the similarity between the objectives of testing and formal inspection? What is the main difference between these two approaches?
AA	**LO: Trace given code to determine its output.** **Exercise 1:** Suppose P and Q are two queues of trees and T is a tree. A tree can have one of the three values: β, γ, ϕ. The initial value of a tree is ϕ. For each of the three questions below, assume that at the beginning, P = < β, ϕ>, Q = < γ, γ >, and T = γ. Trace the given code and show the values of P and Q for each (a) and (b). a. P :=: Q; Enqueue (T, P); *Answer:* P = Q= Clear (T); b. Suppose the statement P := Q copies the value of Q to P. Also, suppose that Enqueue restores its entry. P := Q; Enqueue (T, P); *Answer:* P = Q= Clear (T);

RCI# 4.1.1.3. *Formal Verification*

KC	**LO: State the goals of formal verification.** **Exercise 1:** Circle the phrase that best completes the sentence. The goal of formal verification is: a. prove that a piece of software works on all valid inputs; b. reveal the presence of software bugs; c. show that a piece of software is syntactically correct; d. improve code efficiency.

RCI# 4.1.2. Motivation for modular reasoning
RCI# 4.1.2.1. *Problems with implementation to implementation coupling*

KC	**LO:** **#1. Define what is an implementation to implementation coupling.** **#2. State why implementation to implementation coupling is not desirable.**

RCI# 4.1.2.2. *Desirable coupling through contracts (specifications)*

KC	**LO:** **#1. Define what is coupling through contracts.** **#2. State why coupling through contracts is desirable.** **#3. State the role of contracts for modular reasoning.**

RCI# 4.2. Design-by-Contract
RCI# 4.2.1. Roles of clients and service providers
RCI# 4.2.1.1. *Specifications as external contracts*

KC	**LO: Describe a good external contract specification.** **Exercise 1:** Circle the best answer to the following question: If a method p( ) calls a method q( ), what facts must the developer know to determine whether p() placed the call correctly? a. The post-condition for p( ).

	b. The post-condition for q( ). c. The pre-condition for p( ). d. The pre-condition for q( ). **Exercise 2:** Answer true or false to the following statement: T/F In general, it is possible to write the same specification in many different ways, and it is possible to implement a given specification in many different ways. **Exercise 3:** Draw a UML diagram to show the relationship between the above facility and the concepts, enhancements, and implementations it uses.

RCI# 4.2.1.2. ***Client***

KC	**LO: State the responsibilities of clients (component users) in design-by-contract.**

RCI# 4.2.1.3. ***Service provider***

KC	**LO: State the responsibilities of service providers (component implementers) in design-by-contract.**

RCI# 4.2.1.4. ***Client implementation***

AA	**LO: Examine a client code to determine if it violates a component's external contracts**
SE	**LO: Write an example of client code without violating a component's external contracts.**

RCI# 4.2.1.5. ***Service provider implementation***

AA	**LO: Examine a service provider code to determine if it violates a component's external contracts**
SE	**LO: Write an example of service provider code without violating a component's external contracts.**

RCI# 4.2.2. Construction of new components from built-in components
RCI# 4.2.2.1. *Implementation with arrays*

SE	**LO: Write an implementation of a software component using an array.**

RCI# 4.2.2.2. *Implementation with records (structures)*

SE	**LO: Write an implementation of a software component using a record.**

RCI# 4.2.3. Construction of new components using existing components
RCI# 4.2.3.1. *Implementation of a specification (data representation, code for operations)*

SE	**LO: Implement a given specification using existing components.**

RCI# 4.2.3.2. *Implementation of enhancement specification*

SE	**LO: Implement an enhancement specification.**
	
	Example 1:
	Write an enhancement specification for Flipping_Capability for Stack_Template, and provide its implementation.
	
	Example 2:
	Write an enhancement specification for Find_Min_Int_Capability for Integer_Template, and provide its implementation.
	
	Exercise 3:
	Write a realization of the following enhancement for Queue_Template.
	
	`Enhancement Rotating_Capability for Queue_Template;`
	`  Operation Rotate (evaluates n: Integer; updates Q: Queue);`
	`  requires  0 <= n <= |Q|`
	`  ensures  Q = Prt_Btwn(n, |#Q|, #Q) o Prt_Btwn (0, n, #Q);`
	`end Rotating_Capability;`
	
	`Realization Rotating_Realiz for Rotating_Capability;`
	`  Procedure Rotate (evaluates n: Integer; updates Q: Queue);`
	`  ....`
	`  end Rotate;`
	`end Rotating_Realiz;`

RCI# 4.3. Internal contracts and assertions
RCI# 4.3.1. Internal contracts for data representations
RCI# 4.3.1.1. *Abstraction functions/relations (correspondence)*

KC

LO: Define the term abstraction function (correspondence) and explain its role.

Exercise 1:
Explain what a correspondence is and what purpose does it serve.

Exercise 2:
For the statements below please indicate whether they are true or false.

T/F Correspondence is an assumption that is true at the beginning and end of every procedure;
T/F Without correspondence it is impossible to reason about the correctness of the implementation with respect to specification.
T/F Correspondence is an abstraction function, or abstraction relation.

AA

LO: Determine if the implementation of a component violates the correspondence assertion.

Exercise 1:
Examine the code below and determine if operations Push and Pop for this implementation of the Stack template violate the correspondence assertion.

```
Realization Init_Array_Realiz for Stack_Template;
  Type Stack is represented by Record
          Contents: Array 1..Max_Depth of Entry;
          Top: Integer;
     end;
     convention
          1 <= S.Top <= Max_Depth+1;
     correspondence
          Conc.S = Reverse(Concatenation i: Integer
          where 1 <= i <= S.Top-1, <S.Contents(i)>);
     initialization
          S.Top := 1;
     end;
  Procedure Push(alters E: Entry; updates S: Stack);
     E :=: S.Contents[S.Top];
     S.Top := S.Top + 1;
  end Push;

  Procedure Pop(replaces R: Entry; updates S: Stack);
     S.Top := S.Top - 1;
     R :=: S.Contents[S.Top];
  end Pop;
  ....
```

	Exercise 2:
	Examine the code below and determine the correct correspondence for this implementation.
	
	`Realization Init_Array_Realiz for Stack_Template;`
	`  Type Stack is represented by Record`
	`  Contents: Array 1..Max_Depth of Entry;`
	`  Top: Integer;`
	`  end;`
	`  convention`
	`  1 <= S.Top <= Max_Depth+1;`
	`  correspondence`
	`  ????`
	`  initialization`
	`  S.Top := 1;`
	`  end;`
	`  Procedure Push(alters E: Entry; updates S: Stack);`
	`  E :=: S.Contents[S.Top];`
	`  S.Top := S.Top + 1;`
	`  end Push;`
	
	`  Procedure Pop(replaces R: Entry; updates S: Stack);`
	`  S.Top := S.Top - 1;`
	`  R :=: S.Contents[S.Top];`
	`  end Pop;`
	`  ....`
	
	a. `Conc.S = Reverse(Concatenation i: Integer`
	`  where 1 <= i <= S.Top-1, <S.Contents(i)>);`
	
	b. `Conc.S = Concatenation i: Integer`
	`  where 1 <= i <= S.Top-1, <S.Contents(i)>;`

SE	**LO: Generate a realization for a component without violating the component's correspondence assertion.**
	
	Exercise 1:
	Complete the following implementation of Preemptable_Queue_Template, without violating the conventions of correspondence assertions. Notice that the Stack facility has been enhanced by the flipping capability; your code needs to take appropriate use of this enhancement for full credit.
	
	`Realization Stack_Based_Realiz for Preemptable_Queue_Template;`
	`  uses Stack_Template;`
	`  Facility Entry_Stack_Fac is Stack_Template (Entry, Max_Length)`
	`  realized by Array_Realiz`
	`  enhanced by Flipping_Capability`
	`  realized by Iterative_Realiz;`
	`  Type P_Queue = Record`
	`  Contents: Entry_Stack_Fac.Stack;`

```
end;
   convention true;
   correspondence Conc.Q = Q.Contents;

Procedure Enqueue (alters E: Entry; updates Q: P_Queue);
     ...
end Enqueue;

Procedure Inject (alters E: Entry; updates Q: P_Queue);
     ...
end Inject;

Procedure Dequeue (replaces R: Entry; updates Q: P_Queue);
     ...
end Dequeue;

Procedure Swap_First_Entry (updates E: Entry;
                            updates Q: P_Queue);
     ...
end Swap_First_Entry;
   end Stack_Based_Realiz;
```

RCI# 4.3.1.2. ***Representation invariants (conventions)***

KC	**LO: Define the term representation invariant (convention) and explain its role.** **Exercise 1:** For the statements below please indicate whether they are true or false. T/F Convention is an assumption that is true at the beginning and end of every procedure; T/F Every procedure must guarantee that the convention holds after the procedure completes; T/F Convention is an abstraction function, or abstraction relation. **Exercise 2:** Explain why conventions are written before correspondence.

AA	**LO: Determine if an implementation violates the convention assertion.**
	
	Exercise 1:
	Convention is an assumption that is true at the beginning and end of every procedure. Examine the code below for operations *Enqueue()* and *Dequeue()* and determine if the code violates the convention.
	
	`Realization Circular_Array_Realiz for Queue_Template;`
	`  Type Queue = Record`
	`  Contents: Array 0..Max_Length - 1 of Entry;`

```
        Front, Length: Integer;
    end;
    convention
        0 <= Q.Front < Max_Length and 0 <= Q.Length <= Max_Length;
    correspondence
        Conc.Q = (Concatenation i: Integer
        where Q.Front <= i <= Q.Front + Q.Length - 1,
        <Q.Contents(i mod Max_Length)>);

    Procedure Enqueue(alters E: Entry; updates Q: Queue);
        Q.Contents[(Q.Front + Q.Length)mod Max_Length] :=: E;
        Q.Length := Q.Length + 1;
    end Enqueue;

    Procedure Dequeue(replaces R: Entry; updates Q: Queue);
        Q.Contents[Q.Front] :=: R;
        Q.Front := (Q.Front + 1) mod Max_Length;
        Q.Length := Q.Length -1;
    end Dequeue;

       .....
```

SE

LO: Write code without violating the convention.

Exercise 1:

Below is part of a Stack Template Realization that uses a clean array. Please provide a suitable convention for the code below.

```
Realization Clean_Array_Realiz for Stack_Template;

   Definition Array_is_Clean(SR: Stack): B =
        For all i: Integer, if SR.Top < i <= Max_Depth
        then Entry.Is_Initial(SR.Contents(i));

   Type Stack is represented by Record
        Contents: Array 1..Max_Depth of Entry;
        Top: Integer;
   end;
   convention
        ?????
```

Exercise 2:

Below is the implementation for the queue template that uses circular array. Convention is usually written before the correspondence because the correspondence only needs to be interpreted for the representations that satisfy the conventions. In this case the convention was accidentally deleted, after correspondence was written and all operation were implemented. You will need to work backwards to restore the missing convention.

```
Realization Circular_Array_Realiz for Queue_Template;
   Type Queue = Record
      Contents: Array 0..Max_Length - 1 of Entry;
      Front, Length: Integer;
```

```
end;
convention
   ??????

correspondence
   Conc.Q = (Concatenation i: Integer
   where Q.Front <= i <= Q.Front + Q.Length - 1,
   <Q.Contents(i mod Max_Length)>);

Procedure Enqueue(alters E: Entry; updates Q: Queue);
   Q.Contents[(Q.Front + Q.Length)mod Max_Length]:=:E;
   Q.Length := Q.Length + 1;
end Enqueue;

Procedure Dequeue(replaces R: Entry; updates Q: Queue);
   Q.Contents[Q.Front] :=: R;
   Q.Front := (Q.Front + 1) mod Max_Length;
   Q.Length := Q.Length -1;
end Dequeue;

   ....
```

RCI# 4.3.2. Assertions
RCI# 4.3.2.1. *Loop invariants*

KC	**LO:** **#1. Define what a loop invariant is.** **#2. State the role of loop invariants in determining the loop termination.** **Exercise 1:** For each statement below indicate if it is true or false. T/F Loop invariant is a statement of the relationship that a loop maintains among the loop variables; T/F To show that a loop terminates, as in the case of recursive implementations, a *maintaining* assertion is needed; T/F The *maintaining clause* is true at the beginning and at the end of every iteration; T/F This is an example of a loop invariant: |P| + |Q| ≤ Max_Length;
AA	**LO:** **#1. Complete a piece of code by filling in the suitable loop invariants.** **#2. Complete a piece of code according to the given loop invariants.**
SE	**LO: Write code for a procedure using suitable loop invariants.**

RCI# 4.3.2.2. ***Progress metrics (loops and recursive procedures)***

KC	**LO:** **#1. Define what is a progress metric.** **#2. State the role of progress metrics in ensuring that a loop or a recursive procedure terminates.** **Exercise 1:** For each of the statements below please indicate whether they are true or false. T/F A progress metric is a natural number which reduces on each pass through the loop. T/F The loop must terminate because there are no infinite decreasing sequences of natural numbers. T/F Every recursive procedure needs a decreasing metric in order to terminate.
AA	**LO: Select suitable progress metrics to show termination of a loop or recursive procedure.**
SE	**LO: Write a loop or a recursive procedure with a suitable progress metric.**

RCI# 5. CORRECTNESS PROOFS
RCI# 5.1. Motivation
RCI# 5.1.1. Meaning of correctness
RCI# 5.1.1.1. *Semantics*
RCI# 5.1.1.2. *Soundness and relative completeness*

KC	**LO: State what it means for a piece of code to be sound and relatively complete.**
	Exercise 1: Circle all correct statements below. Code correctness means: a. An operation works exactly as specified in the pre- and post conditions. b. The code terminates and produces expected output for all the valid inputs. b. The code is professionally written using the language chosen in the project documentation. c. The code produces expected output for all the inputs that you have used to test it. **Exercise 2:** For the two statements below please indicate if they are true or false: T/F When we say the code is correct, it means that the code adheres to its formal specifications; T/F Code correctness is a highly desirable characteristic, especially in safety-critical and high-assurance software.

RCI# 5.1.2. Motivation for proofs
RCI# 5.1.2.1. *Partial correctness*

KC	**LO: State what is a partial correctness of code.**

RCI# 5.1.2.2. *Total Correctness*

KC	**LO: State what is a total correctness of code.** **Exercise 1:** Please indicate whether the following statement is true or false. T/F If the code is incorrect, the base case may never be reached. **Exercise 2:** Circle the correct answer. Total correctness means:

	a. the code is correct. b. the code terminates. c. the code is correct and it terminates. d. the code has both the requires and ensures clauses.

RCI# 5.2. Construction of verification conditions (VCs)
RCI# 5.2.1. States and abstract values of objects
RCI# 5.2.1.1. *Naming Conventions*

KC **LO: State the naming conventions used in VCs.**

Example1:
How many states are there in the procedure below? How do we determine this?

```
Operation Mystery (updates i: Integer);
     requires  i > 0 and I <= max_int - 2;
     ensures  i = #i + 2;
  Procedure
     Increment (i);
     Increment (i);
  end;
end Mystery;
```

AA **LO: Determine the values of variables at different states of a piece of code.**

Example 1:
Below is the tracing table for the operation Do_Nothing. For each state in the table please indicate to which state the variables belong. What variables are we dealing with in state 2?

```
Operation Do_Nothing (restores I: Integer)
     requires  Min_Int <=1 and (I + 1) <= Max_Int;
     ensures  I = #I ;
  Procedure
     Increment ( I );
     Decrement ( I );
  end;
end Do_Nothing;
```

State	Statement	Assume	Confirm
0		Min_Int <= I and (I + 1) <= Max_Int	(I + 1) <= Max_Int
	Increment (I);		
1		I = #I + 1	Min_Int <= (I -1)
	Decrement (I);		
2		I = #I + 1	I = #I

Exercise 2:
Examine the code below. The tracing table is provided. For each state in the table label to which state variables belong. What variables are we dealing with in state 1?

```
Operation Reverse2(updates Q: Queue);
     requires |Q| = 2;
     ensures Q=Rerverse(Q);
  Procedure
     Var Temp: Entry;
     Dequeue (Temp, Q);
     Enqueue (Temp, Q);
  end;
end Reverse2;
```

State	Statement	Assume	Confirm
0		|Q | < Max_Length, |Q | = 2;	|Q | > 0
	Dequeue (Temp, Q);		
1		Q = <Temp> o Q	|Q | < Max_Length
	Enqueue (Temp, Q);		
2		Q = Q o <Temp>	Q = Reverse (Q)

RCI# 5.2.2. Connection between specifications and what is to be proved
RCI# 5.2.2.1. *Assumptions*

KC

LO:
#1. State what are the assumptions of an operation.
#2. State the role of assumptions in proving correctness of code.

Exercise 1:
Look at the code below. Where do assumptions of the operation Clear2 come from?

```
Operation Clear2(updates S: Stack);
     requires (|S| = 2);
     ensures S = empty_string;
  Procedure
     Pop (E, S);
     Pop (E, S);
  end;
end Clear2;
```

a. requires clause of operation Clear2()
b. ensures clause of operation Clear2()
c. both the requires and ensures clauses of operation Clear2()
d. requires clause of operation Pop()

AA	**LO: Explain from where the assumptions come.**
	
	Example 1:
	For the procedure Clear_2 show the assumptions, if any, in state 0. Specification of operation Pop() is provided for your convenience.
	
	`Operation Pop(replaces R: Entry; updates S: Stack);`
	`  requires |S| /= 0;`
	`  ensures #S = <R> o S;`
	
	`Operation Clear_2(updates S: Stack);`
	`  requires (|S| = 2);`
	`  ensures S = empty_string;`
	`Procedure Clear_2(updates S: Stack);`
	`  Var Next_Entry: Entry;`
	`0`
	`  Pop(Next_Entry, S);`
	`1`
	`  Pop(Next_Entry, S);`
	`2`
	`end;`
	`end Clear_2;`
	
	Example 2:
	For operation Add_2 show assumptions, if any, in states 0 and 1.
	
	`Operation Add_2(updates I: Integer);`
	`  requires I = 0;`
	`  ensures I = #I + 2;`
	`Procedure`
	`0`
	`  Increment (I);`
	`1`
	`  Increment (I);`
	`2`
	`end;`
	`end Clear_2;`

SE	**LO: Fill in the assumptions for verification of a procedure in different states.**
	
	Example 1:
	Complete the reasoning table below by filling in the assumptions. Specifications of operations Increment () and Decrement () are shown below.
	
	`Procedure Inc_Dec (I: Integer)`
	`  requires  Min_Int <= I and (I + 1) <= Max_Int;`
	`  ensures  I = #I;`
	`Procedure`
	`  Increment(I);`
	`  Decrement(I);`

```
end;

Operation Increment(updates I: Integer);
     requires I + 1 <= Max_Int;
     ensures     I = #I + 1;

Operation Decrement(updates I: Integer);
     requires Min_Int <= I - 1;
     ensures     I = #I - 1;
```

State	Code	Assume True	Confirm True
0			$I_0 + 1 <= Max\_Int;$
	Increment (I)		
1			$Min\_Int <= I_1 - 1;$
	Decrement (I)		
2			$I_2 = I_0;$

Example 2:

For the operation below please fill in the assumptions in the reasoning table. Specifications of operations Push() and Pop() are provided.

```
Operation Manipulate (restores S: Stack);
     requires |S| > 0;
     ensures S = #S;
  Procedure
     Var E: Entry;
     Pop(E, S);
     Push(E, S);
  end;
end Manipulate;

Operation Pop (replaces E: Entry; updates S: Stack);
     requires |S| /= 0;
     ensures #S = <E> o S;

Operation Push(alters E: Entry; updates S: Stack);
     requires |S| + 1 <= Max_Depth;
     ensures  S = <#E> o #S;
```

State	Code	Assume True	Confirm True
0			$\|S_0\| /= 0;$
	Pop (E, S)		
1			$\|S_1\| + 1 <= Max\_Depth;$
	Push (E, S)		
2			$S_2 = S_0$

RCI# 5.2.2.2. *Obligations*

KC

LO:
#1. Define what are obligations of an operation.
#2. State the role of obligations in proving correctness of code.

Exercise 1:
What are the obligations of the operation Calculate?

```
Operation Calculate (updates Result: Integer, alters I: Integer);
      requires Result >= 0 and Result + I <= Max_Int );
      ensures Result = #Result+#I;
   Procedure
      Var K: Integer;
      K := Sum(Result,I);
      Result :=: K;
   end;
 end Calculate;
```

Exercise 2:
What are the obligations of the operation Clear_3?

```
 Operation Clear_3 (updates S: Stack);
      requires (|S| = 3);
      ensures S = empty_string;
   Procedure
      Pop (E, S);
      Pop (E, S);
      Pop (E, S);
   end;
 end Clear_3;
```

AA

LO: State from where the assumptions come.

Example 1:
For operation Add_2 show obligations, if any, in states 1 and 2.

```
 Operation Add_2(updates I: Integer);
      requires   I = 0;
      ensures   I = #I + 2;
   Procedure
   0
      Increment (I);
   1
      Increment (I);
   2
   end;
 end Add_2;
```

Example 2:

For the following procedure Move_One show the obligations, if any, in states 1 and 2.

```
Operation Move_One (updates Q: P_Queue);
     ensures Q = Prt_Btwn(1,|Q|-1,#Q) o Prt_Btwn(0,1,#Q);
  Procedure
     Var E: Entry;
  0
     Dequeue (E, Q);
  1
     Enqueue (E, Q);
  2
  end;
end Flip;
```

SE

LO: Fill in the obligations for verification of a procedure in different states, and generate verification conditions.

Exercise 1:

Complete the reasoning table below by filling in the obligations. Specifications of operations Increment () and Decrement () are shown below.

```
Procedure Inc_Dec (restores I: Integer)
     requires  Min_Int <= I and (I + 1) <= Max_Int;
     ensures   I = #I;
  Procedure
     Increment(I);
     Decrement(I);
  end;
end Inc_Dec;

Operation Increment(updates I: Integer);
     requires I + 1 <= Max_Int;
     ensures      I = #I + 1;

Operation Decrement(updates I: Integer);
     requires  Min_Int <= I - 1;
     ensures   I = #I - 1;
```

State	Code	Assume True	Confirm True
0		Min_Int <= I_0 and (I_0 + 1) <= Max_Int;	
	Increment (I)		
1		$I_1 = I_0 + 1$;	
	Decrement (I)		
2		$I_2 = I_1 - 1$;	

Example2:

Complete the reasoning table below by filling in the obligations. Specifications of

operations Pop () and Push () are shown below.

```
 Operation Manipulate (restores S: Stack);
      requires |S| > 0;
      ensures S = #S;
   Procedure
      Var E: Entry;
      Pop(E, S);
      Push(E, S);
 end Manipulate;

Operation Pop (replaces E: Entry; updates S: Stack);
      requires |S| /= 0;
      ensures #S = <E> o S;

Operation Push(alters E: Entry; updates S: Stack);
      requires |S| + 1 <= Max_Depth;
      ensures  S = <#E> o #S;
```

State	Code	Assume True	Confirm True
0		$\|S_0\| > 0$;	
	Pop (E, S)		
1		$S_1 = <E> \text{ o } S_0$;	
	Push (E, S)		
2		$S_2 = <E_1> \text{ o } S_1$;	

RCI# 5.2.3. Types of statements
RCI# 5.2.3.1. *Sequential Statements*

SE

LO: Construct verification conditions (VCs) for a procedure that contains sequential statements.

Exercise 1:
For the code below, construct the reasoning table and generate verification conditions.

```
Operation Inc_Dec (i: Integer);
   requires min_int < i  and  i + 1 <= max_int;
   ensures i = #i;
 Procedure
      Increment (I);
      Decrement (I);
 end Inc_Dec;
```

State	Code	Assume	Confirm
0			
	Increment (i)		
1			
	Decrement (i)		

2			

Exercise 2:
For the code below, fill out the tracing table. Assume the initial value of I is 5.

```
Operation Inc_Dec (i: Integer);
   requires min_int < i  and  i + 1 <= max_int;
   ensures i = #i;
Procedure
      Increment (I);
      Decrement (I);
end Do_Nothing;
```

State	Statement	Assume	Confirm
		I	
0		5	i + 1 ≤ max_int
	Increment (I)		
1			min_int ≤ i - 1
	Decrement (I)		
2			i = #i;

RCI# 5.2.3.2. ***Conditional Statements***

AA	**LO: State how conditions give rise to assumptions.**

SE	**LO: Construct verification conditions (VCs) for a procedure that contains conditional statements.**
	
	Exercise 1:
	For the code below please construct a reasoning table and generate VCs.
	
	`Operation Manipulate (i: Integer);`
	`  requires min_int < i  and  i + 1 <= max_int;`
	`  ensures i = #i;`
	`Procedure`
	`  Increment (I);`
	`  Increment (I);`
	`  If(I > 10) then`
	`  Decrement (I);`
	`  end;`
	`end Manipulate;`
	
	State | **Code** | **Assume** | **Confirm**
	0 | | |

1			
2			
3			

Exercise 2:

For the code below, please complete the tracing table. Assume the initial value of I is 9. Generate verification conditions.

```
Operation Manipulate (i: Integer);
   requires min_int < i  and  i + 2 <= max_int;
   ensures i >= #i + 1;
Procedure
      Increment (I);
      Increment (I);
      If(I >= 10) then
          Decrement (I);
      end;
end Manipulate;
```

State	Statement	Assume	Confirm
		I	
0		9	i + 1 ≤ max_int
	Increment (I)		
1			i + 1 ≤ max_int
	Increment (I)		
2			
	If (I >10)		
3			min_int ≤ i - 1
	Decrement		
4			i >= #i + 1;

RCI# 5.2.3.3. ***Loops***

AA	**LO:** **#1. State how loop conditions give rise to assumptions.** **#2. State how loop invariants give rise to assumptions.** **#3. State how loop invariants give rise to obligations.** **#4. State how progress metrics give rise to obligations.**

SE

LO: Construct verification conditions (VCs) for a procedure that contains loops.

Exercise 1:

For the code below construct a reasoning table and generate VCs

```
Procedure Remove_Last (updates Q: Queue; replaces E: Entry);
     Var T: Queue;
     Dequeue (E, Q);
     While (Length(Q) /= 0)
          changing Q,T,E;
          maintaining #Q = T ∘ <E> ∘ Q;
          decreasing |Q|;
     do
          Enqueue(E,T);
          Dequeue(E,Q);
     end;
     Q :=: T;
end Remove_Last;
```

Exercise 2:

For the code below, please fill out the tracing table. Assume the initial value of I is 5.

```
Procedure Remove_Last (updates Q: Queue; replaces E: Entry);
     Var T: Queue;
     Dequeue (E, Q);
     While (Length(Q) /= 0)
          changing Q,T,E;
          maintaining #Q = T ∘ <E> ∘ Q;
          decreasing |Q|;
     do
          Enqueue(E,T);
          Dequeue(E,Q);
     end;
     Q :=: T;
end Remove_Last;
```

State	Statement	Assume	Confirm
0			
1			
2			
3			
4			

RCI# 5.2.3.4. *Operation Calls*

AA	**LO:** **#1. State how an operation call gives rise to obligations in the state before the call.** **#2. State how an operation call gives rise to assumptions in the state after the call.**
SE	**LO: Construct verification conditions (VCs) for a procedure that contains operation calls.**

RCI# 5.2.4. Connection between induction and reasoning
RCI# 5.2.4.1. *Base case*

KC	**LO:** **#1. State what is a base case of mathematical induction.** **#2. Point out how base cases are used in proving correctness of a loop, given its invariant, or a recursive procedure.**

RCI# 5.2.4.2. *Inductive Case*

KC	**LO:** **#1. Distinguish between a base case and an inductive case of mathematical induction.** **#3. Point out how induction is used in proving correctness of a loop, given its invariant, or a recursive procedure**

RCI# 5.2.4.3. *Termination*

KC	**LO:** **#1. State why it is important to prove the termination of a piece of code.** **#2. State the connection between the progress metric and the validity of inductive supposition.**
AA	**LO:** **#1. Determine if a recursive procedure terminates.** **#2. Determine if a given progress metric is appropriate to show the termination of a loop or recursive procedure.**
SE	**LO: Write a progress metric for a loop or a recursive procedure that terminates and construct VCs to prove it terminates.**

RCI# 5.3. Proof Of VCs
RCI# 5.3.1. VCs as mathematical implications
RCI# 5.3.1.1. *Givens (Assumptions)*

KC	**LO: State what are the givens.**
AA	**LO: Identify the givens relevant to establishing a goal.**

RCI# 5.3.1.2. *Goals (Obligations)*

KC	**LO: State what are the goals.**
AA	**LO: Identify provable goals.** **Exercise 1:** Identify provable goals. **VC_1:** *Goal:* (|New_Queue'| < Max_Length) *Given:* 1: (min_int <=0) 2: (0 < max_int) 3: (Max_Length > 0) 4: (min_int <= Max_Length) and (Max_Length <= max_int) 5: (|Q| <= Max_Length) 6: |Q| /=0 7: Q = (<Min'> o Q''') 8: Is_Permutation (((New_Queue' o Q'') o <Min'>), Q) 9: (|Q''| > 0) 10: Q'' = (Considered_Entry'> o Q') **VC_2:** *Goal:* |New_Queue'| = (|Q| - 1) *Given:* 1: (min_int <=0) 2: (0 < max_int) 3: (Max_Length > 0) 4: (min_int <= Max_Length) and (Max_Length <= max_int) 5: (|Q| <= Max_Length) 6: |Q| /=0 7: Q = (<Min'> o Q''') 8: Is_Permutation (((New_Queue' o Q'') o <Min'>), Q) 9: not ((|Q'| = 0))
SE	**LO: Prove goals when presented with mathematical implications (VCs), containing suitable givens.**

Exercise 1:
Prove the goals that are provable.

VC_1:
Goal: (|New_Queue'| < Max_Length)
Given:
1: (min_int <=0)
2: (0 < max_int)
3: (Max_Length > 0)
4: (min_int <= Max_Length) and (Max_Length <= max_int)
5: (|Q| <= Max_Length)
6: |Q| /=0
7: Q = (<Min'> o Q''')
8: Is_Permutation (((New_Queue' o Q'') o <Min'>), Q)
9: (|Q''| > 0)
10: Q'' = (Considered_Entry'> o Q')

VC_2:
Goal: |New_Queue'| = (|Q| - 1)
Given:
1: (min_int <=0)
2: (0 < max_int)
3: (Max_Length > 0)
4: (min_int <= Max_Length) and (Max_Length <= max_int)
5: (|Q| <= Max_Length)
6: |Q| /=0
7: Q = (<Min'> o Q''')
8: Is_Permutation (((New_Queue' o Q'') o <Min'>), Q)
9: not ((|Q'| = 0))

RCI# 5.3.2. Application of proof techniques on VCs
RCI# 5.3.2.1. *Direct proofs*
RCI# 5.3.2.2. *Rules of Inference*

SE	**LO: Prove a given VC using various rules of inference.**

WORKS CITED

1. Abrial, J. R. (2007). Formal methods: Theory becoming practice. *Journal of Universal Computer Science*, *13*(5), 619–628.

2. Abrail, J. R. (2006). Formal methods in industry: Achievements, problems, future. *Proceedings of the 28th International Conference on Software Engineering*.

3. Almstrum, V. L., Henderson, P. B., Harvey, V., Heeren, C., Marion, W., Riedsel, C., Soh, L.-K., Tew, A. E. (2006). Concept inventories in computer science for the topic Discrete Mathematics. *Proceedings of the 11th Annual Conference on Innovation and Technology in Computer Science Education, Working Group Reports*.

4. Almstrum, V. L., Hilburn, T., Dean, C. N., Smith, J., Goelman, D. (2000). Support for teaching formal methods. *Proceedings of the 5th Annual Conference on Innovation and Technology in Computer Science Education, Working Group Reports*.

5. Alpert, S. R., Singley, M. K., Fairweather, P. G. (1999). Deploying intelligent tutors on the web: An architecture and an example. *International Journal of Artificial Intelligence in Education*, *10*(2), 183-197.

6. Anastasakis, K., Bordbar, B., Georg, G., Ray, I. (2007). UML2Alloy: A challenginh model transformation. *Proceedings of the 10th international conference on Model Driven Engineering Languages and Systems.*

7. Baldwin, D. *Mathematical (and Other) Reasoning in Computer Science Education*. Birds-Of-a-Feather Session, 42nd ACM Technical Symposium, Special Interest Group on Computer Science Education, 2011.

8. Baldwin, D., Henderson, P., Sitaraman, M. *Mathematical Reasoning in Computer Science Education*. Birds-Of-a-Feather Session, ACM Technical Symposium, Special Interest Group on Computer Science Education, 2008.

9. Baldwin, D., Marion, B., Sitaraman, M., Heeren, C. (2010). Some developments in mathematical thinking for computer science education since computing curricula 2001, 392-393. *Proceedings of the 41st ACM Technical Symposium on Computer Science Education*.

10. Barnett, M., Leino, K. R. M., Schulte, W. (2004). The Spec# programming system: An overview. *Proceedings of Construction and Analysis of Safe, Secure and Interoperable Smart Devices.*

11. Bayley, I. (2006). Teach z by reverse engineering specifications from real-life implementations. *Teaching Formal Methods: Practice and Experience*.

12. Bloom, B. S., Engelhart, M. D., Furst, E. J., Hill, W. H., Krathwohl, D. R. (Eds.) *Taxonomy of Educational Objectives. The Classification of Educational Goals, Handbook I: Cognitive Domain.* New York, David McKay Company, Inc., 1956.

13. Bowen, J., Hinchey, M. G. (2005). Ten Commandments revisited: A ten-year perspective on the industrial application of formal methods. *Proceedings of the 10th International Workshop on Formal Methods in Industrial Critical Systems.*

14. Bronish, D., Weide, B. W. (2010). A review of verification benchmark solutions using Dafny. *Proceedings of the Verified Software: Theories, Tools, and Experiments Workshops*.

15. Brookes, S. D., Hoare, C. A. R., Roscoe, A. W. (1984). A Theory of communicating sequential processes. *Journal of the ACM, 31*(3).

16. Bucci, P., Long, T., Weide, B. (2001). Do we really teach abstraction? *Proceedings of the 32nd SIGCSE Technical Symposium on Computer Science Education, 33*(1), 26-30.

17. Bucci, P., Hollingsworth, J. E., Krone, J., Weide, B. W. (1994). Part III: implementing components in RESOLVE. *SIGSOFT Software Engineering Notes, 19*(4), 40-51.

18. Bucci, P., Long, T. J., Weide, B. W., Hollingsworth, J. E. (2000). Toys are us: Presenting mathematical concepts in CS1/CS2. *Proceedings of the 30th Frontiers in Education Conference.*

19. Bucur, Doina, Kwiatkowska, M. (2010). Software verification for TinyOS. *Proceedings of the 9th ACM/IEEE International Conference on Information Processing in Sensor Networks.*

20. Carmona, J., Cortadella, J., Takada, Y., Peper, F. (2008). Formal methods for the analysis and synthesis of nanometer-scale cellular arrays. *Journal on Emerging Technologies in Computing Systems, 4*(2).

21. Chen, J. C., Kadlowec, J., Whittinghill, D. (2004). Work In progress: combining concept inventories with rapid feedback to enhance learning. *34th ASEE/IEEE Frontiers in Education Conference.*

22. Cheon, Y., Leavens, G. T. (1994). The Larch/Smalltalk specification language. *Transactions on Software Engineering and Methodology*, *3*(3).

23. Chiang, C.-C. (2004). Teaching a formal method in a software engineering course. *Proceedings of the 2nd Annual Conference on Mid-South College Computing.*

24. Cho, I.-H., McGregor, J. (2000). A formal approach to specifying and testing the interoperation between components. *Proceedings of the 38th on Annual Southeast Regional Conference.*

25. Cook, C.T. (2011). A Web-integrated environment for component-based reasoning. *Master of Science Thesis, Clemson University School of Computing.*

26. Cook, C.T., Drachova, S., Hallstrom, J.O., Hollingsworth, J., Jacobs, J.P., Krone, J., and Sitaraman, M. (2012). A Systematic Approach to Teaching Abstraction and Mathematical Modeling. *Proceedings of the 17th ACM annual conference on Innovation and technology in computer science education.*

27. Cook, C. T., *A Web-Integrated Environment for Component-Based Reasoning*, M. S. Thesis, Clemson, September 2011

28. Cooper, S., Cassel, L., Cunningham, S. , Moskal, B. (2005). Outcomes-based computer science education. *Proceedings of the 36th SIGCSE Technical Symposium on Computer Science Education.*

29. Crow, J., Owre, S., Rushby, J., Shankar, N., Srivas, M. (1995). A tutorial introduction to PVS. *Proceedings of the Workshop on Industrial-Strength Formal Specification Techniques*.

30. Davies, J. , Gibbons, J. (2009). Formal methods for future interoperability. *SIGCSE Bulletin*, *41*(2).

31. Davis, J. F. (2005). The affordable application of formal methods to software engineering. *Proceedings of the 2005 annual ACM SIGAda international conference on Ada.*

32. Del Vade Virseda, R., Fernandez, P., Munos, S., Murillo, A. (2009). Intelligent tutoring system for interactive learning of data structures. *Proceedings of Third IEEE International Conference on Semantic Computing*.

33. Devlin, K. (2001). Viewpoint: the real reason why software engineers need math. *Communications of the ACM*, 44(10).

34. Dony, I., Le Chalier, B. (2006). A tool for helping teach a programming method. *Proceedings of the 11th annual SIGCSE conference on Innovation and technology in computer science education.*

35. Drachova-Strang, S. (2009). A hands-on approach for teaching design by contract to software engineering students. *RESOLVE Workshop, Clemson University School of Computing, Clemson, SC, 2009.*

36. Drachova-Strang, S. V, Hollingsworth, J. E., Sitaraman, M. (2011). Experimentation with tutors for teaching mathematical reasoning and specifications. *Proceedings of the 2011 International Conference on Frontiers in Education: Computer Science and Computer Engineering, WorldComp 2011*.

37. East, J. P. (2006). On models of and for teaching: Toward theory-based computing education. *Proceedings of the 2006 international workshop on Computing education research.*

38. Evans, D., Guttag, J., Horning, J., Tan, Y. M. (1994). LCLint - a tool for using specifications to check code. *ACM SIGSOFT Software Engineering Notes*, *19*(5).

39. Eykhoff, P. (1974). *System identification: parameter and state estimation*. Chichester, England: Wiley.

40. Edwards, S. H., Heym, W. D., Long, T. J., Sitaraman, M., Weide, B. (1994). Part II: Specifying Components in RESOLVE. *SIGSOFT Software Engineering Notes*, *19*(4), 29-39.

41. Finnet, K. (1996). Mathematical notation in formal specification: too difficult for the masses? *IEEE Transactions on Software Engineering*, *22*(2), 158-159.

42. Fitzgerald, J., Larsen, P. G., Sahara, S. (2008). VDMTools: advances in support for formal modeling in VDM. *ACM SIGPLAN Notices*, *43*(2).

43. Fraser, M. D., Kumar, K., Vaishnavi, V. K (1994). Strategies for incorporating formal specifications in software development. *Communications of the ACM*, *37*(10), 74-86

44. Garis, A., Paiva, A.C.R., Cunha, A., Riesco, D. (2012). Specifying UML protocol state machines in alloy. *Proceedings of the 9th international conference on Integrated Formal Methods.*

45. Gopalakrishnan, G. L., Kirby, R. M. (2010). Top ten ways to make formal methods for HPC practical. *Proceedings of the FSE/SDP workshop on Future of SE Research.*

46. Gee, J. P. (2005). Learning by design: Good video games as learning machines. *E-Learning and Digital Media*, *2*(1).

47. Glasman, L. R., ALbarracin, D. (2006). Forming attitudes that predict future behavior: a meta-analysis of the attitude-behavior relation. *Psychological Bulletin, 132*, 778-822.

48. Goldman, K., Gross, P., Hereen, C., Herman, G., Kaczmarczyk, L., Loui, M. C., Zilles, C. (2008). Identifying important and difficult concepts in introductory computing courses using a Delphi process. *Proceedings of the 39th SIGCSE technical symposium on Computer science education.*

49. Gries, D., Marion, B., Henderson, P., Schwartz, D. (2001). How mathematical thinking enhances computer science problem solving. *Proceedings of the 32nd SIGCSE technical symposium on Computer Science Education.*

50. Gries. D. (1981). *The Science of Programming*. Springer-Verlag.

51. Hallstrom, J., Hollingsworth, J., Krone, J., Sitaraman, M. *Making Mathematical Reasoning Fun: Tool-Assisted, Collaborative Learning Techniques. Birds-Of-a-Feather Workshop,* Special Interest Group in Computer Science Education, 2012.

52. Hallstrom, J., Hollingsworth, J., Krone, J., Sitaraman, M. *Making Mathematical Reasoning Fun: Well-Integrates,Collaborative, and "Hands-On" Techniques.* Birds-Of-a-Feather Session. Special Interest Group in Computer Science Education, 2013.

53. Harton, H. K., Sitaraman, M., Krone, J. (2008). Formal Program Verification. In: Wah B (ed.), *Wiley Encyclopedia of Computer Science and Engineering*. John Wiley & Sons.

54. Harton, L. (1997). What is a formal method, (and what is an informal method)? *Computer Assurance*.

55. Henderson, P., Baldwin, D., Sitaraman, M. *Mathematical Reasoning in Computer Science.* Birds-Of-a-Feather Session, 40th ACM Technical Symposium, Special Interest Group in Computer Science Education, 2009.

56. Henderson, P. B. (2003). Mathematical reasoning in software engineering education. *Communications of the ACM*, *46*(9), 45-50.

57. Henderson, P. B., Baldwin, D., Dasigi, V., ..., Walker, H. (2001). Striving for mathematical thinking. *SIGSCE Bulletin, 33*(4), 114-124

58. Herman, G. L., Loui, M.C., Zilles, C. (2010). Creating the digital logic concept inventory. *Proceedings of the 41st ACM Technical Symposium on Computer Science Education*.

59. Hestenes, D., Wells, M., Swackhamer, G. (1992). Force concept inventory. *The Physics Teacher*, *30*(3), 141-151.

60. Hilburn, T. (1996). Inspections of formal specifications. *Proceedings of the 27th SIGCSE Technical Symposium on Computer Science Education*.

61. Hinchey, M. G., Rouff, C. A., Rash, J. L., Truszkowski, W. F. (2005). Requirements of an integrated formal method for intelligent swarms. *Proceedings of the 10th international workshop on Formal Methods for Industrial Critical Systems*.

62. Hinchey, M., Jackson, M., Cousot, P., Cook, B., Bowen, J. P., Margaria, T. (2009). Software engineering and formal methods**.** *Communications of the ACM, 51*(9).

63. Hoare, T., Misra, J. (2008). *Verified software: Theories, Tools, and Experiments: Vision of a Grand Challenge project.* First IFIP TC2/EG2.3 Conference, Lecture Notes in Computer Science, vol. 4171, Springer-Verlag, Berlin, Heidelberg, Germany, 1-18.

64. Hollingsworth, J., Blankenship, L., Weide, B. (2000). Experience Report: Using RESOLVE/C++ for commercial software. *Proceedings of the 8th Eighth International Symposium on the Foundations of Software Engineering: Twenty-Forst Century Applications.*

65. Holloway, C.M. (1997). Why engineers should consider formal methods. *Proceedings of the AIAA/IEEE Digital Avionics Systems Conference.*

66. Holzmann, G. J. (2005). The Spin Model Checker. Addison-Wesley. http://spinroot.com/spin/whatispin.html

67. Howe, E., Thornton, M., Weide, B. (2004). Components-first approaches to CS1/CS2: principles and practice. *Proceedings of the 35th SIGCSE Technical Symposium on Computer Science Education*.

68. Hung, P. N., Aoki, T., Katayama, T. (2009). An effective framework for assume-guarantee verification of evolving component-based software. *Proceedings of the joint international and annual ERCIM workshops on Principles of software evolution (IWPSE) and software evolution (Evol) workshops.*

69. IEEE/ACM 2001 Computing Curriculum: Computer Science Volume. (2001). http://www.acm.org/education/curricula/ComputerScience2008.pdf

70. Jacobi, A., Marti, J., Mitchel, J., Newell, T. (2003). A concept inventory for heat transfer. *Frontiers in Education*.

71. Jennigs, T.J. (2009). SPARK: the libre language and toolset for high-assurance software programs. *Proceesings of the ACM SIGAda annual international conference on Ada and related technologies.*

72. Kirschenbaum, J., Adcock, B., Bronish, D., Bucci, P., Weide, B. W. (2008). Adapting Isabelle theories to help verify code that uses abstract data types. *Proceedings 7th International Workshop on Specification and Verification of Component-Based Software.*

73. Klebanov, V., Muller, P., Shankar, N., Leavens, G. T., Wustholz, V., Alkassar, E., ...Weiss, B. (2011). The 1st verified software competition: Experience report. *Proceedings of the 17th International Symposium on Forman Methods, Springer LNCC 6664*, 154-168.

74. Krause, S., Birk, J., Bauer, R., Jenkins, B., Pavelich, M. J. (2004). Development, testing, and applications of a chemistry concept inventory. *Frontiers in Education.*

75. Krone, J., Baldwin, D., Carver, J., Hollingsworth, J., Kumar, A. *Teaching Mathematical Reasoning Across the Curriculum*. Panel Session. Special Interest Group in Computer Science Education, 2012.

76. Krone, J., Sitaraman, M., and Hallstrom, J. O. (2011). Mathematics throughout the CS curriculum. *Journal of Computing Sciences in Colleges, 27*(1).

77. Krone, J., Ogden, W. F. (2006). Software verification is not dead, but it needs a new way to express mathematics. *Proceedings of the RESOLVE Workshop*.

78. Krone, J., Hollingsworth, J. E., Sitaraman, M., Hallstrom, J. (2010). A Reasoning Concept Inventory for Computer Science. Technical Report RSRG-09-01.

79. Krone, J., Sitaraman, M., Hallstrom, J. *Making Mathematical Reasoning Fun: with Tools and Collaborative Learning Methods. Birds-Of-a-Feather Session,* 41st ACM Technical Symposium, Special Interest Group in Computer Science Education, 2010.

80. Kulczycki, G., Sitaraman, M., Ogden, W. F., and Weide, B. W. (2005). Clean semantics for calls with repeated argument. *Technical Report RSRG-05-01, Clemson University School of Computing.*

81. Kulczycki, G., (2004). Direct Reasoning. *Ph. D. Dissertation, Clemson University School of Computing.*

82. Kulczycki, G., Sitaraman, M., Yasmin, N., Roche, K. (2009). Formal Specification. In: *Wah B (ed.), Wiley Encyclopedia of Computer Science and Engineering, John Wiley & Sons*.

83. Kulczycki, G., Sitaraman, M., Weide, B., Rountev, A. (2005). A specification-based approach to reasoning about pointers. *Proceedings of the 2005 Conference on Specification and Verification of Component-Based Systems*.

84. Kumar, A. (2000). Dynamically generating problems on static scope. *Proceedings of the 5th Annual Conference on Innovation and Technology in Computer Science Education*.

85. Kumar, A. (2005). Generation of problems, answers, grade, and feedback - case study of a fully automated tutor. *Journal on Educational Resources in Computing, 5*(3).

86. Kuncak, V., Rinard, M. (2006). An overview of the Jahob analysis system - Project goals and current status. *NSF Next Generation Software Workshop*.

87. Langari, Z., Pidduck, A. B. (2005). Quality, cleanroom and formal methods. *ACM SIGSOFT Software Engineering Notes*, *30*(4).

88. Larsen, P. G., Fitzgerald, J., Brooks, T. (1996). Applying formal specification in industry, *IEEE Software*, *13*(3), 48-56.

89. Leavens, G. T., Baker, A. L., Ruby, C. (1999). JML: A notation for detailed design. In: *Kilov, H., Rumpe, B., Simmonds, I. (editors). Behavioral Specifications of Businesses and Systems*, *12*, 175-188.

90. Leonard, D. P., Hallstrom, J. O., Sitaraman, M. (2009). Injecting rapid feedback and collaborative reasoning in teaching specifications. *Proceedings of the 40th ACM Technical Symposium on Computer Science Education*.

91. Lethbridge, T., Diaz-Herrera, J., LeBlanc, R., Thompson, J. (2007). Improving software practice through education: Challenges and future trends. *Future of Software Engineering*.

92. Lethbridge, T. (2000). What knowledge is importasnt to a software professional? *Computer, 33*(5).

93. Liu, S., Takahashi, K., Hayashi, T., Nakayama, T. (2009). Teaching formal methods in the context of software engineering. *SIGCSE Bulletin*, *41*(2),17-23.

94. Maibaum, T. (2009). Formal methods versus engineering. *SIGCSE Bulletin*, *41*(2).

95. McLean, J. (2007) Formal methods in security engineering: Where we've been, where we are, where we need to go. *Proceedings of the 2007 ACM workshop on Formal Methods in Security Engineering*.

96. McLoughlin, H., Hely, K. (1996). Teaching formal programming to first year computer science students. *SIGCSE Bulletin, 28*(1).

97. Meyer, B. (1992). Applying design by contract. *Computer, 25*(10), 40-51.

98. Meyer, B. (1986). Design by contract. *Technical Report TR-EI-12/CO. Interactive Software Engineering Inc*.

99. Meyer, B. (1992). *Eiffel: The Language*. Prentice Hall.

100. Midkiff, K. C., Litzinger, T. A., Evans, D.L. (2001). Development of engineering thermodynamics concept inventory instruments. *Proceedings of the31st ASEE/IEEE Frontiers in Education Conference.*

101. Mitzner, T., Rogers, W. (2005). Create II: Manual for conducting focus groups. *Technical Report HFA-TR-05-05, Atlanta, Ga.: Georgia Institute of Technology*.

102. Mottola, L., Voigt, T., Osterland, F., Eriksson, J., Baresi, L., Ghezzi, C. (2010). Anquirro: Enabling efficient static verification of sensor network software. *Proceedings of the 2010 ICSE Workshop on Software Engineering for Sensor Network Applications.*

103. Nakkrasae, S., Sophatsathit, P. (2002). A formal approach for specification and classification of software components. *Proceedings of the 14th International Conference on Software Engineering and Knowledge Engineering.*

104. Ohlsson, S. (1987). Some principles of intelligent tutoring. *In: Lawler & Yazdani (eds.), Artificial Intelligence and Education, 1,* 203-238.

105. Ogden, W. F., Sitaraman, M., Weide, B. W, Zweben, S. H. (1994). Part I: The RESOLVE framework and discipline: a research synopsis. *Software Engineering Notes*, *19*(4), 23-38.

106. Ogden, W. F., Hollingsworth, J. E., Krone, J., Sitaraman, M., Weide, B. W. (2007). The RESOLVE software verification vision. *Proceedings of the RESOLVE Workshop.*

107. Padue, W. (2010). Measuring complexity, effectiveness, and efficiency in software course projects. *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering, 1,* 545-554.

108. Parham, J. Chinn, D., Stephenson, D. E. (2009). Using Bloom's taxonomy to code verbal protocols of students solving a data structure problem. *Proceedings of the 47th Annual Southeast Regional Conference.*

109. Paige, R. F., Ostroff, J. S. (2004). Specification-driven design with Eiffel and agents for teaching lightweight formal methods. *Teaching Formal Methods*, 107-123.

110. Parnas, D. L. (1990). Education for Software Professionals. *Computer, 23*(1).

111. Periyasamy, K. (2008). A GUI-based editor and type checker for object-Z. *Journal of Computing Sciences in Colleges*, *24*(1).

112. Reed, J. N., Sinclair, J. E. (2004). Motivating study of formal methods in the classroom. *Teaching Formal Methods*, 32-46.

113. RESOLVE Workshop 2006, Virginia Tech, March 22-23, 2006, Blacksburg, VA.

114. RESOLVE Workshop 2007, Clemson University, June 11-13, 2007, Clemson, SC.

115. RESOLVE Workshop 2009, 11the International Conference on Software Reuse, September 27-30, 2009, Falls Church, VA.

116. RESOLVE Workshop 2010, Denison University, Junue 8th, 2010, Granitville, OH.

117. Riener, H., Fey, G. (2012). FAuST: a framework for formal verification, automated debugging, and software test generation. *Proceedings of the 19th international conference on Model Checking Software.*

118. Roedel, R. J., El-Ghazaly, S., Rhoads, T. R., El-Sharawy, E. (1998). The wave concepts inventory an assessment tool for courses in electromagnetic engineering. *Frontiers in Education*.

119. Rhoads, T. R., Roedel, R. (1999). The wave concept inventory - A cognitive instrument based on Bloom's taxonomy. *Frontiers in Education*.

120. Rybalchenko, A. (2011). Towards automatic synthesis of software verification tools. *Proceedings of the 13th international ACM SIGPLAN symposium on Principles and practices of declarative programming.*

121. Sitaraman, M., Hallstrom, J., Whire, J., Drachova-Strang, S., Harton, H. K., Leonard, D., Krone, J., Pak, R. (2008). Engaging students in specification and reasoning: "hands-on" experimentation and Evaluation. *Proceedings of the 14th annual ACM SIGCSE Conference on Innovation and Technology in Computer Science Education.*

122. Sitaraman, M., Weide, B. W. (1994). Component-based software using RESOLVE. *Software Engineering Notes, 19*(4), 21-22.

123. Sitaraman, M., Long, T. J., Weide, B.W., Harner, E. J., Wang, L. A formal approach to component-based software engineering: Education and evaluation. *Proceedings of the 23rd International Conference on Software Engineering*.

124. Sitaraman M., Adcock B., Avigad J., Bronish D., Bucci P., Frazier D., Friedman H., ..., Weide B.W. (2011). Building a push-button RESOLVE verifier: Progress and challenges. *Formal Aspects of Computing, 23*(5), 607-626.

125. Skevoulis, S., Falidas, M. (2002). Integrating formal methods tools into undergraduate computer science curriculum**.** *Proceedings of the 7th annual Conference on Innovation and Technology in Computer Science Education*.

126. Stone, A., Allen, K., Rhoads, R. R., Murphy, T. J., Shehab, R. L., Saha, C. (2003). The statistics concept inventory: A pilot study. *Proceedings of the 33rd ASEE/IEEE Frontiers in Education Conference*.

127. Sykes, E. R., Franek, F. (2003). An intelligent tutoring system prototype for learning to program Java. *Proceedings of the 3rd IEEE International Conference on Advanced Learning Technologies*.

128. Tagore, A., Zaccai, D., Weide, B. W. (2011). To expand or not to expand: Automatically verifying software specified with complex mathematical definitions. *Technical Report OSU-CISRC-5/11-TR18, Department of Computer Science and Engineering, The Ohio State University, May 2011 (revised September 2011).*

129. Thang, N. T., Katayama, T. (2005). Specification and Verification of inter-component constraints in CTL. *Proceedings of the 2005 Conference on Specification and Verification of Component-Based Systems*.

130. Tomer, T. S., Baldwin, D., Fox, C. J., (1998). Integration of mathematical topics in CS1 and CS2. *Proceedings of the 29th SIGCSE Technical Symposium on Computer Science Education*.

131. VCC: A verifier for Current C. http://research.microsoft.com/en-us/projects/vcc/

132. Weide, B. W., Sitaraman, M., Harton, H. K., Adcock, B., Bucci, P., Bronish, D., ... , D. Frazier. (2008). Incremental benchmarks for software verification tools and techniques. *Proceedings of 2008 Conference on Verified Software: Theories, Tools, and Experiments*.

133. Wing, J. (1987). Writing Larch interface language specification. *Transactions on Programming Languages and Systems, 9*(1).

134. Woodcock, J., Larsen, P. G., Bicarregui, J., Fitzgerald, J. (2009). Formal methods: Practice and experience. *Computing Surveys*, *41*(4).

135. Zic, J. J. (1987). Extensions to communicating sequential processes to allow p rotocol performance specification. *Proceedings of the ACM workshop on Frontiers in Computer Communications Technology.*

136. Zingaro, D. (2008). Another approach for resisting student resistance to formal methods. *SIGCSE Bulletin, 40*(4).