5-2009

Teaching Specifications Using An Interactive Reasoning Assistant

Dana Leonard
Clemson University, dleonar@clemson.edu

Teaching Specifications Using An Interactive Reasoning Assistant

A Thesis
Presented to
the Graduate School of
Clemson University

In Partial Fulfillment
of the Requirements for the Degree
Masters of Science
Computer Science

by
Dana Leonard
May 2009

Abstract

The importance of verifiably correct software has grown enormously in recent years as software has become integral to the design of critical systems, including airplanes, automobiles, and medical equipment. Hence, the importance of solid analytical reasoning skills to complement basic programming skills has also increased. If developers cannot reason about the software they design, they cannot ensure the correctness of the resulting systems. And if these systems fail, the economic and human costs can be substantial.

In addition to learning analytical reasoning principles as part of the standard Computer Science curriculum, students must be excited about learning these skills and engaged in their practice. Our approach to achieving these goals at the introductory level is based on the Test Case Reasoning Assistant (TCRA), interactive courseware that allows students to provide test cases that demonstrate their understanding of instructor-supplied interface specifications while receiving immediate feedback as they work. The constituent tools also enable instructors to rapidly generate graphs of student performance data to understand the progress of their classes. We evaluate the courseware using two case-studies. The evaluation centers on understanding the impact of the tool on students' ability to read and interpret specifications.

Table of Contents

Title Page . . . . **i**

Abstract . . . . **ii**

List of Figures . . . . **iv**

1 Introduction **1**
1.1 Motivation 1
1.2 Problem Statement 3
1.3 Solution Approach 3
1.4 Thesis Statement 4
1.5 Summary 4

2 Test Case Reasoning Assistant (TCRA) **5**
2.1 Classroom Reasoning Module 5
2.2 Specification Approach and Syntax 7
2.3 User View 7
2.4 Implementation View 12

3 Related Work **25**
3.1 Teaching Rigorous Foundations 25
3.2 Teaching Tools 26
3.3 Collaborative Learning 27
3.4 Specification-Based Tools 28

4 Evaluation **30**
4.1 First Classroom Experiment 30
4.2 Group Experiment 39
4.3 Second Classroom Experiment 42
4.4 Threats to Validity 43

5 Conclusions and Discussion **51**
5.1 Summary 51
5.2 Future Work 52

6 Appendix **54**

Bibliography **61**

List of Figures

2.1 TCRA Student View 8
2.2 TCRA Instructor View 11
2.3 TCRA UML Diagram 13
2.4 Example Log File Data 18
2.5 TCRA Instructor UML Diagram 18
2.6 XMLLog Structure 19
2.7 State Diagram of LogParser 20

4.1 Q1 Scores (Pre-Tool) 31
4.2 Q2 Scores (Some Using TCRA) 32
4.3 Q3 Scores (All Using TCRA) 33
4.4 Correctness of Test Cases Over Time for Q2. 35
4.5 Correctness of Test Cases Over Time for Q3. 36
4.6 Number of Test Cases Submitted Over Time for Q2. 37
4.7 Number of Test Cases Submitted Over Time for Q3. 38
4.8 Correctness of Test Cases Over Time for Group Exercise. 40
4.9 Number of Test Cases Submitted Over Time for Group Exercise. 41
4.10 Quiz 1 Scores (Pre-Tool) 43
4.11 Quiz 2 Scores (Half Using TCRA) 44
4.12 Quiz 3 Scores (All Using TCRA) 45
4.13 Correctness of Test Cases Over Time for Quiz 2. 46
4.14 Correctness of Test Cases Over Time for Quiz 3. 47
4.15 Number of Test Cases Submitted Over Time for Quiz 2. 48
4.16 Number of Test Cases Submitted Over Time for Quiz 3. 49

Chapter 1

Introduction

Computer science education is a cornerstone of future software development. A student should leave their academic program with the ability to analyze a problem, design a solution, and implement that solution. Some students will find themselves developing software for critical systems like medical equipment, automobiles, aircraft, etc. If the software contains implementation errors, the wrong set of conditions could bring about deadly results [20, 22, 27, 38, 12]. This is why educators have been looking for ways to teach students to develop verifiably correct software. Three important topics for the completion of this goal are: formal verification, computer-aided study tools, and student performance monitoring.

1.1 Motivation

Formal Verification. There are several proposed remedies for error-prone development strategies: formal methods, model checking, design-by-contract, etc. Many of these remedies rely on an individual's mathematical reasoning abilities. This requires that students have a reasonable background in discrete mathematics, Boolean logic, state machines, and other topics.

These subject areas can be difficult for students as many of them are taught for a week or two in one class and then never used again. Likewise, some computer science degree programs require very little discrete math, focusing instead on calculus [18]. Another complication lies in the ability of a student to practice this mathematical reasoning outside the classroom. Most practice comes in the form of pen-and-paper problems given to students as homework. This has several intrinsic problems. To determine if the problem has been completed correctly, the student needs some form of outside checking or feedback. This generally comes in the form of an instructor who checks student work, but instructors are not always available. Pen-and-paper exercises are also time-consuming for both students and instructors. Students must hand write each solution and may be required to present several test cases per exercise which may involve writing the same information more than once. Instructors must grade each solution by reading the test case and comparing it to the specification. This can be very slow for non-trivial specifications, which increases the amount of time until students receive feedback. This creates a need for automated verification software, which is described in the next section.

Computer-aided Study Tools. *Interactive courseware* refers to a collection of programs that enable students to practice the topic of instruction and receive immediate feedback. The feedback can be tailored to provide varying levels of detail, from simple "right/wrong" feedback to explanations of the problem and its solution. This feedback allows students to know immediately if they can proceed to the next exercise or if their solution needs modification. Interactive courseware is also intended to get students more engaged then they are when performing pen-and-paper exercises. Examples of successful courseware are Pargas' MessageGrid [28] and Kumar's online tutor problets [16]. The next paragraph describes how the results from the tools can be used to monitor student understanding.

Student Performance Monitoring. An important property of interactive courseware is its ability to collect and summerize data on student performance. Performance data can be collected while students use a given tool and stored for later analysis. This data may be difficult to interpret, so tools are needed to help instructors make use of the data. Web-CAT [5], an online grading system, uses several code qualities to determine a student's grade for a given submission. Each quality has an automated evaluator and supplies simplified performance for instructor viewing. This relieves the instructor from combing through large amounts of data and student submissions, and preserves the important information pertinent to student performance. Student performance data can also be viewed over time to locate particular problem areas in the class. If the data shows that most students performed below their usual level on a particular exercise, this may indicate that the topic covered by the exercise requires more instruction time to be understood fully.

Monitoring can also help in determining whether students are making random guesses until they reach a solution. If student input values are viewable by the instructor, he can see how students attempt to correct their errors. If a student is clearly not making changes to the incorrect part of the solution, the instructor could choose to not give the student credit for that exercise or make a point to review the exercise in class.

1.2 Problem Statement

Currently, no courseware exists that enables students to practice their formal verification skills, provides immediate feedback, and enables instructors to monitor student performance during practice. Students must rely on traditional pen-and-paper methods of practice and instructor feedback when it is available. Instructors only have tests, homework, and student questions to monitor student performance. This thesis addresses the following problems:

- How can courseware help students study and practice software specifications more effectively?
- How can courseware help instructors better gauge student understanding of specifications?

1.3 Solution Approach

The courseware developed for this thesis is called the Test Case Reasoning Assistant (TCRA). It enables students to read program specifications and create test cases that are validated programmatically. The courseware then provides feedback based on the correctness of the test cast presented. This provides students the ability to immediately identify their mistakes and correct them. By repeatedly correcting mistakes, students are able to better understand the specification content than if they were to submit paper-based homework responses and receive feedback after the instructor has had a chance to grade them.

Student inputs and correctness results are logged by the tool so that the instructor may collect the logs for the purpose of grading and/or to determine how well the class understands the material. The instructor-side of the tool generates several types of graphs from the collected logs to make the data easy to interpret. If the instructor finds a common problem area in the student results, they can revisit and focus on that specific area to help students understand better.

1.4 Thesis Statement

This thesis defends the following statements:

- The use of TCRA improves student understanding of interface specifications with the use of immediate feedback mechanisms.
- TCRA allows students to identify their mistakes, if present.
- TCRA improves student performance on class quizzes.

1.5 Summary

The remainder of this thesis is organized as follows. In Chapter 2, introduces the goals of the course module and the design of the TCRA student and instructor modules is presented. API design and use-cases are discussed in detail. Chapter 3 discusses the in-classroom case studies, and the evaluation of the results. Chapter 4 relates TCRA to related work from the field, followed by a summary of contributions and an overview of future work in Chapter 5.

Chapter 2

Test Case Reasoning Assistant (TCRA)

This chapter will introduce TCRA, a pair of Java [26] applications designed for student and instructor use, respectively. The first section describes the classroom module that the tools target. The next section introduces the specification approach and syntax. Next, the user view of both the student and instructor tools will be described. Finally, the implementation view of both tools is detailed.

2.1 Classroom Reasoning Module

The reasoning module is designed to be used in the first undergraduate course that covers interface contracts. At Clemson, this is CpSc 215, Software Development Foundations. In the module, students learn analytical reasoning principles related to abstract mathematical models, formal interface specifications, and how to choose components based on interface specifications. The reasoning module is introduced after the basics of specifications using pre- and post-conditions have been taught.

The module begins with a short survey of the basic mathematical types used in program specifications. These types include integers, reals, sets, and strings. More advanced types are deferred to later modules or courses. Each type is introduced with examples of values an instance of the type may hold and the operations supported by the type. For example, students are shown String instances such as <1,2,3>, a string containing three integers, as well as <>, the empty string. They are then introduced to the length (||) and concatenation ($\star$) operators. Simple examples are used initially to ensure student understanding. After the mathematical types have been introduced, existential and universal quantifiers are reviewed, and simple quantification expressions are discussed. It is worth noting that this portion of the module is largely review; the concepts are familiar to most students.

Next, students discuss how to map program types to their representative mathematical types. For example, students are introduced to the string model in the context of a list object, which students then extend to objects such as stacks and queues. Once several of these connections have been identified, interface specifications are presented. For example, the RESOLVE specification of the `Push()` operation defined for a stack component is introduced as follows:

```
Operation Push(preserves X: Integer,
               updates S: Stack);
requires |S| < MAX_DEPTH;
ensures S = <X> ⋆ #S;
```

The parameter annotations indicate that `Push()` does not modify the value of `X`, but may modify the value of `S`. Requires and ensures clauses represent the operation's pre- and post-conditions, respectively. The # symbol denotes the pre-conditional value of the corresponding variable. Thus, `Push()` can be called if and only if the stack `S` contains fewer than `MAX_DEPTH` elements, a parameter defined as part of the stack abstract model. Upon termination, `Push()` adds the value of `X` as the leftmost (topmost) entry of `S`. The `Pop()` operation is similarly defined; students may be asked to try and write the specification for the operation.

Once several component specifications have been covered, students are given specifications for mystery operations and asked to determine their behavior. These may be completed individually or in small groups. An example mystery specification can be seen below:

```
Operation Mystery(updates S1, S2: Stack);
requires (|S1| > 0) and |(S2| < MAX_DEPTH - 1)
ensures there exists E: Integer, S: String of Object such that
        #S1 = <E>⋆<S> and
        S2 = <E>⋆<#S2>⋆<E>
```

After the mystery exercises are covered, students are asked to create test cases for the exercises. Students provide pre- and post-conditional values for an operation's variables to test the behavior of the operation, demonstrating their ability to use component implementations based on formal interface specifications. It is at this point that the Test Case Reasoning Assistant is introduced.

2.2 Specification Approach and Syntax

TCRA consists of a pair of Java applications designed for student and instructor use, respectively. The student application is designed to enable users to quickly work through an extensible set of test case creation exercises, receive rapid feedback as they work, and benefit from performance data collected automatically for online and offline analysis. These analysis services are provided by the instructor application, which supports graphical reporting features.

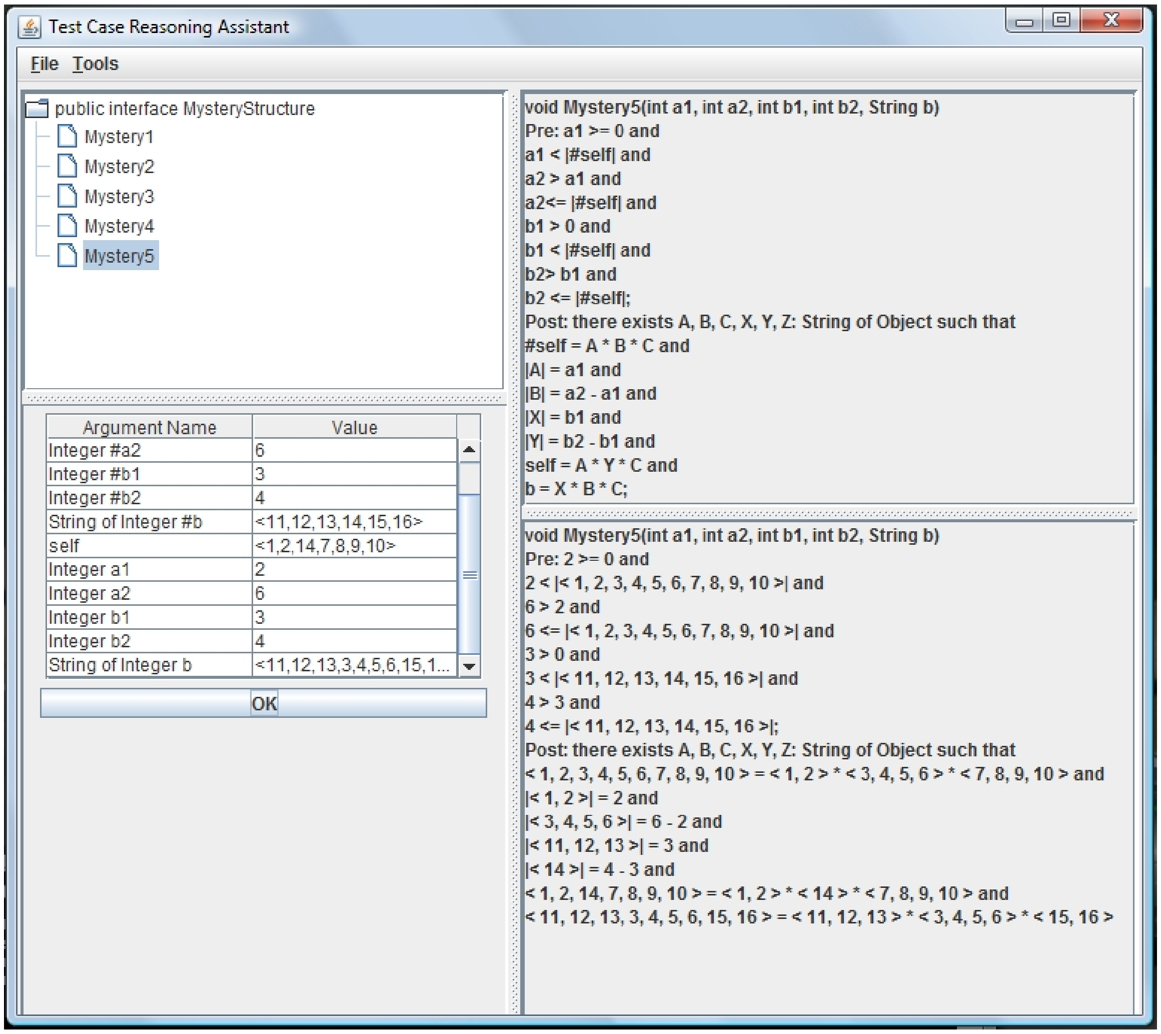
2.3 User View

2.3.1 Student Interface

GUI Description. The student application is shown in Figure 2.1. The screen capture was taken just after the user opened and began the exercise displayed. Note that the user interface design is intentionally simple. The objective is to eliminate student learning time associated with using the tool, focusing instead on the reasoning exercises themselves.

The student tool consists of four sections. The top left portion of the window displays the interface specification under consideration. The front matter preceding the three document icons characterizes the mathematical model. In this case, the Mystery interface is modeled as a String of Integers. Each of the document icons represents an operation included as part of the interface. When one of these icons is selected, the corresponding specification is shown in the top right portion of the exercise window.

The bottom left portion is where users enter test cases, in the form of pre- and post-conditional argument and return values. Basic syntax checking and error reporting are provided. After the test case has been submitted for verification, the *instantiated specification*, derived by substituting the student-provided values into the original specification, is shown in the bottom right


Test Case Reasoning Assistant
File Tools
public interface MysteryStructure
Mystery1
Mystery2
Mystery3
Mystery4
Mystery5
OK

Argument Name	Value
Integer #a2	6
Integer #b1	3
Integer #b2	4
String of Integer #b	<11,12,13,14,15,16>
self	<1,2,14,7,8,9,10>
Integer a1	2
Integer a2	6
Integer b1	3
Integer b2	4
String of Integer b	<11,12,13,3,4,5,6,15,1...

```
void Mystery5(int a1, int a2, int b1, int b2, String b)
Pre: a1 >= 0 and
a1 < |#self| and
a2 > a1 and
a2<= |#self| and
b1 > 0 and
b1 < |#self| and
b2> b1 and
b2 <= |#self|;
Post: there exists A, B, C, X, Y, Z: String of Object such that
#self = A * B * C and
|A| = a1 and
|B| = a2 - a1 and
|X| = b1 and
|Y| = b2 - b1 and
self = A * Y * C and
b = X * B * C;
```

```
void Mystery5(int a1, int a2, int b1, int b2, String b)
Pre: 2 >= 0 and
2 < |< 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 >| and
6 > 2 and
6 <= |< 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 >| and
3 > 0 and
3 < |< 11, 12, 13, 14, 15, 16 >| and
4 > 3 and
4 <= |< 11, 12, 13, 14, 15, 16 >|;
Post: there exists A, B, C, X, Y, Z: String of Object such that
< 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 > = < 1, 2 > * < 3, 4, 5, 6 > * < 7, 8, 9, 10 > and
|< 1, 2 >| = 2 and
|< 3, 4, 5, 6 >| = 6 - 2 and
|< 11, 12, 13 >| = 3 and
|< 14 >| = 4 - 3 and
< 1, 2, 14, 7, 8, 9, 10 > = < 1, 2 > * < 14 > * < 7, 8, 9, 10 > and
< 11, 12, 13, 3, 4, 5, 6, 15, 16 > = < 11, 12, 13 > * < 3, 4, 5, 6 > * < 15, 16 >
```

Figure 2.1: TCRA Student View

portion of the window.

As is obvious in Figure 2.1, the syntax presented in the specifications is different from the syntax used in previous examples. This is a feature that TCRA supports; the instructor can use any specification notation they wish while the validation logic remains the same.

Tool Use. The student's task is to provide, for each operation, test cases that satisfy the corresponding specification. The student begins by reading the specification carefully to understand the method requirements. Once he believes they understand what the method requires, the student enters pre-conditional and post-conditional values for each argument according to its mathematical type. Two mathematical data types are supported in the current implementation: String of Object and Integer. Along with the method arguments and return value if any, the student also enters the pre-condtional and post-conditional values of the target object. When the student is ready to submit the test case for validation, he clicks the OK button. If both the requires and ensures assertions evaluate to true in the instantiated specification, the system notifies the student that a correct test case has been provided. He is then able to continue providing new test cases or move on to another operation or exercise. Otherwise, an error is signaled and the student is prompted to correct the mistake based on his understanding of the instantiated specification. The instantiation process not only provides immediate feedback, but reinforces the process by which students are taught to reason about pre- and post-conditional specifications (i.e., using instantiation).

Log Description. As students work their way through each exercise, the application monitors their activity. This includes recording information about the exercises selected, the test cases provided (including correctness results), and the time at which each activity occurred. Each log is tagged with a unique identifier chosen by the student when the application is first installed, supporting longitudinal analysis without compromising student anonymity. This information can be transmitted to the *log repository* at any time (through the Tools menu). During classroom exercises, students might be prompted to submit their logs at regular intervals. Alternatively, students might submit their logs from home after they've completed a set of assigned exercises. In either case, the logs are collected in the repository for analysis. The address of the repository can be configured in the associated `tcra.config` file by the instructor.

Before considering the instructor interface, it is worth emphasizing that while dozens of exercises are included with the tool, the exercise set is extensible and language neutral. Instructors are not limited to the prescribed exercises, nor the RESOLVE specification and implementation notation. Each exercise is implemented as a single class that conforms to an `Exercise` interface supported by the tool. The design separates the presentation logic used to display the interface signatures and specifications from the checking logic used to test the validity of the inputs provided. These exercises are compiled independently of the TCRA tool, enabling instructors to quickly develop new exercises involving the programming language and specification notation of their choice.

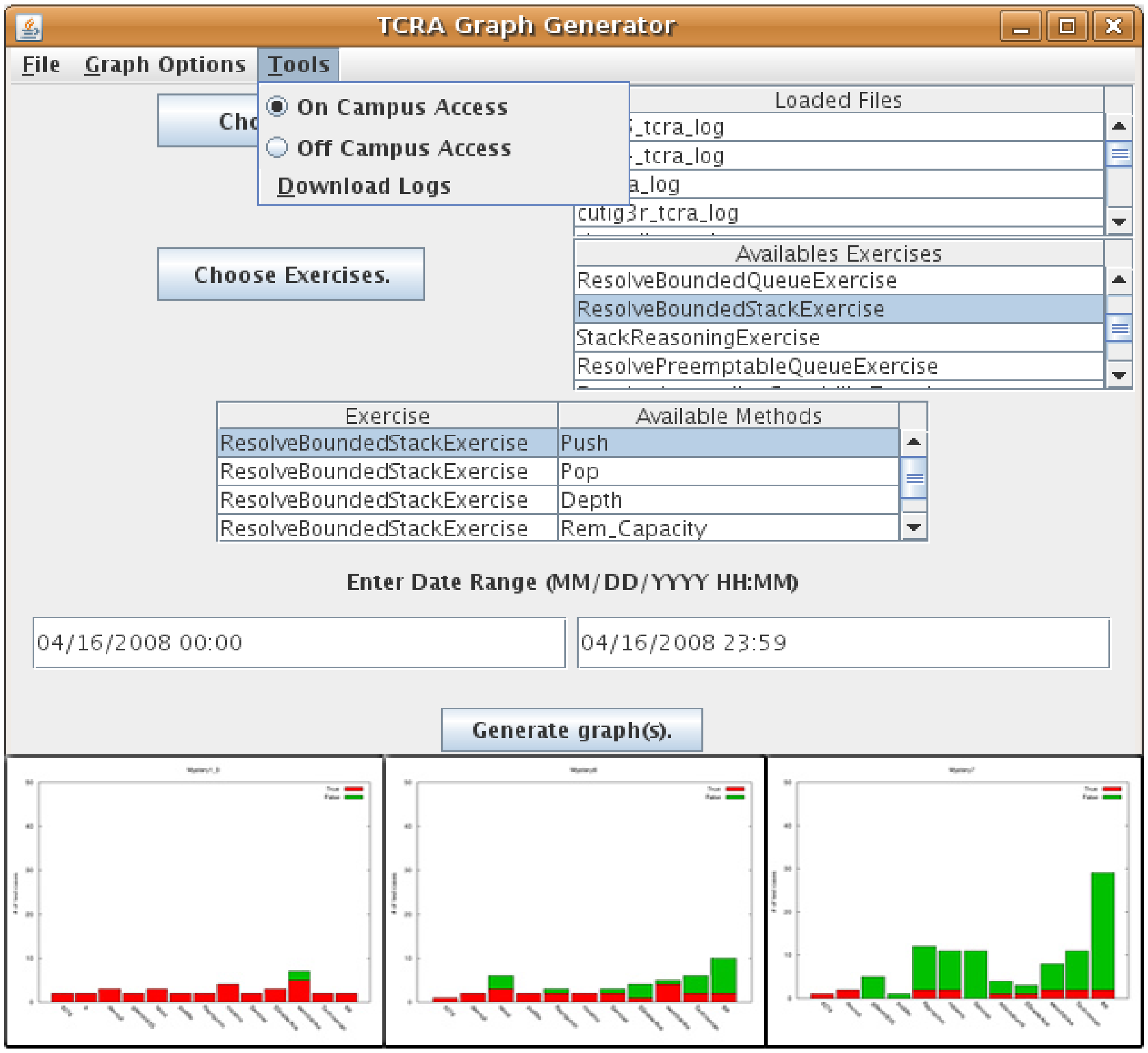
2.3.2 Instructor Interface

Tool Description The instructor interface of TCRA, shown in Figure 2.2 enables instructors to produce graphs from student logs. The generated graphs can be used to grade students' work and to identify weaknesses in understanding. To begin, instructors may download student logs from the repository specified in the `tcra.config` file at any time. The instructor tool includes the same on-campus/off-campus options found in the student interface. The tool then downloads the logs onto the local machine (to the same directory that the tool executes from). Data is stored in separate dated folders so that the instructor can keep a history of student performance.

Next, the instructor chooses the files she wishes to include in a generated graph using the "*Choose Files*" button. This button triggers a file-open dialogue used to navigate to the folder containing the logs of interest. Once the logs are selected, the tool lists all the exercises identified in the logs within the table labeled "*Available Exercises.*" This list is the union of all the exercises found in the logs, so it is possible that some logs do not have test cases for that particular exercise. However, if the tool is used for grading purposes, it is unlikely that a student did not attempt the exercise at all.

Next, the instructor selects the exercises she would like to focus on and then clicks the "*Choose Exercises*" button. In response, the tool populates the "*Available Methods*" table with the method contracts that have been attempted by students. The instructor then chooses the contracts she wishes to include in the graph (recall that an exercise may consist of multiple contracts).

Finally, the instructor enters a date range and time to further further focus on. Once satisfied with her selections, the instructor clicks the "*Generate Graph(s)*" button. When the tool is done generating the graph, an image viewer displays the new graphs automatically.



[● On Campus Access] ○ Off Campus Access

Loaded Files
_tcra_log
_tcra_log
a_log
cutig3r_tcra_log

Availables Exercises
ResolveBoundedQueueExercise
ResolveBoundedStackExercise
StackReasoningExercise
ResolvePreemptableQueueExercise

Exercise	Available Methods
ResolveBoundedStackExercise	Push
ResolveBoundedStackExercise	Pop
ResolveBoundedStackExercise	Depth
ResolveBoundedStackExercise	Rem_Capacity

Enter Date Range (MM/DD/YYYY HH:MM)

04/16/2008 00:00

04/16/2008 23:59

Figure 2.2: TCRA Instructor View

Graph Types Before generating the graphs, the instructor can choose the type of graph she wishes to see. There are three types of graphs available. Some examples can be seen at the bottom of Figure 2.2. The first option, "*aggregate results per method*," creates one graph per method, each capturing the number of right and wrong test cases presented by students. The second graph, "*individual results per student*," creates an graph for each student, showing the number of correct and incorrect test cases for each method selected in the GUI. The final graph, "*aggregate results for all methods*," generates one graph. The graph is similar to that of the "*aggregate results per method*" graph but concatenates the bar graphs, forming one large graph.

For each graph, the y-axis captures the number of test cases while the x-axis captures either the name of the method contract, or the student id. The graph uses stacked format, with false test-cases appearing in green on top of the true test cases in red.

2.4 Implementation View

In this section, the implementation of the two interfaces will be discussed. The description focuses on component interaction and data flow during student and instructor use. A high-level UML diagram shows the object organization of the student view in Figure 2.3.

2.4.1 Student Tool

Exercise Data Structure. A TCRA exercise is comprised of three components: an *interface header*, a collection of method contracts, and information about the mathematical model that represents the interface. The header is displayed as the root of the tree that appears in the top-left pane of the student view, and its child nodes display the names of the methods the exercise contains. The interface may be as simple as "`public interface Stack`," or it may give more detail about the interface, such as length restrictions on the corresponding type. The mathematical model information is used to inform the user of what data type #self and self are. For example, a queue is modeled as a String of Object. An example exercise class can be seen in Listing 2.1.

The `StackReasoningExercise` is derived from the abstract `TCRAExercise` class. It sets the interface header to the typical Java-style interface signature. Then the contracts `Vector` is instantiated and method contracts are added in the order the instructor wishes them to be displayed. The `getMathRep()` function is implemented to return a `MathStringModel`, signifying that #self

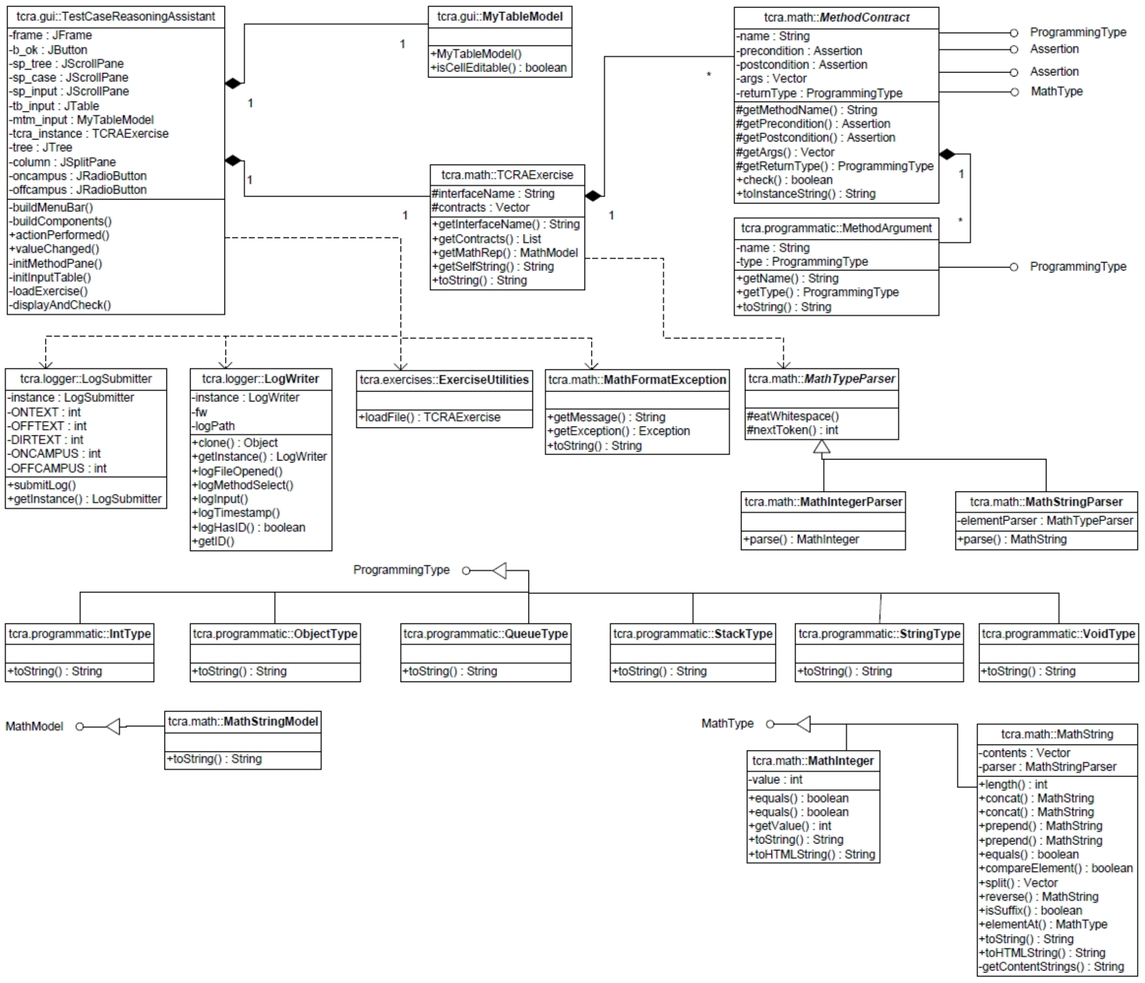


Figure 2.3: TCRA UML Diagram

Listing 2.1: Java-style Stack Reasoning Exercise

```java
public class StackReasoningExercise extends TCRAExercise
{
   public StackReasoningExercise()
   {
      super("public interface Stack");
      contracts = new Vector<MethodContract>();
      contracts.add(new StackPushContract());
      contracts.add(new StackPopContract());
      contracts.add(new StackTopContract());
      contracts.add(new StackLengthContract());
      contracts.add(new StackClearContract());
   } // end ctor

   public MathModel getMathRep() { return new MathStringModel(); }
} // end class
```

is represented by a String of Object.

Method contracts are more detailed. Consider Listing 2.3. `StackPopContract` is derived from the abstract `MethodContract` class. It contains three protected variables that will store the objects that result from parsing student input. They are used to avoid reparsing input when `toInstanceString()` is called. The constructor sets the contract name, associates the pre- and post-conditions with the appropriate `Assertion` objects, instantiates the `Vector<MethodArgurment>`, and sets the `returnType` variable to the appropriate `ProgrammingType` object. It then sets the protected variables to `null` for error-checking later.

The `check()` function parses the input passed down from the GUI using a `MathStringParser` and a `MathIntegerParser`. The objects returned from the `parse()` function of these parsers are stored in the protected variables and then checked to see if any of them remain to be `null`. If so, a `MathFormatException` is thrown signifying that an error was encountered during parsing. If the parsing was successful, the objects are stored in a `Vector<MathType>` and passed to the `Assertions`' `check()` methods. If either `check()` fails, `false` is returned, `true` otherwise.

`toString()` is used to print the specification. It uses HTML tags to allow for better formatting in a single `JLabel` object in the GUI. `toInstanceString()` returns a similar `String`, but uses the instantiated specification instead.

The most important component of a contract is a pair of assertions that define the behavior of the method. The pre-condition assertion defines the conditions that must hold before the method can be invoked, and the post-condition assertion defines the conditions that must be true after the method completes. An example `check()` function can be found on line 5 of Listing 2.2 and will be discussed later. The contract also contains a collection of method arguments that represent the programming types and formal names of the method's arguments. These arguments are used to build the user input table in the lower right pane of the student tool. The contract also contains a field that represents the return type of the method. If a return type exists for the method, an extra row is added to the input table. The GUI accesses the method arguments and return type through accessor methods in `MethodContract`.

The assertions are the core of a method contract. They specify the conditions that must be satisfied by the user's test case through the `check()` method. Consider this example. Listing 2.2 shows how `check()` checks that `#self` is equal to `self` concatenated with `pop`. The assertions also return strings representing their formal representation and the value instantiated specification through `toString()` and `toInstanceString()`, respectively. This finishes decoupling the checking logic from display logic throughout the `TCRAExercise` class structure.

An example `Assertion` is supplied in Listing 2.2. The `check()` method simply casts the objects from the `Vector` argument into temporary variables of the appropriate type, which is safe to do since the exercise developer controls the ordering of `Vector` elements throughout the exercise. The `if` statements below check the values of `lb_self`, `self`, and `pop` for errors. The first `if` statement is used as a shortcut around the more expensive `concat()` and `equals()` functions in the `if` statement below it. If `self` does not contain one fewer elements than `lb_self` then `false` can be returned immediately. Next, `pop` is concatenated onto `self` and then compared to `lb_self`. If the two are equal, `true` is returned. `toString()` returns the formal representation of the assertion in any specification notation the developer wishes to use. `toInstanceString()` returns the value instantiated specification by calling the `toHTMLString()` function of each `MathType`. `toHTMLString()` is necessary since the "<" and ">" characters will not be displayed correctly in an HTML context.

User Input Validation. When a user has entered a test case and clicks the OK button, input validation begins. The input is passed to the method contract object through its `check()` method. Within this method, the input strings are parsed into `MathType` objects using the `MathTypeParser` appropriate for the type of input expected. Since the exercise developer knows the order and type of the arguments in the contract, she can parse the strings accordingly. If the parse fails for any reason, a `MathFormatException` is thrown and an error window is shown to the user. If everything parses correctly, the `MathTypes` are passed to the pre-condition and post-condition objectsfor further validation as seen on line 32 of Listing 2.3.

To understand the validation process more fully, assume the following input to the `StackPopContract` in Listing 2.3:

```
#self = <1,2,3,4>
self  = <1,2,3>
pop() = 4
```

The input is passed from the GUI object, `TestCaseReasoningAssistant`, to the contract via the `check()` method. `StackPopContract` parses the first two Java `Strings` to `MathStrings` using a `MathStringParser` which, in turn, uses a `MathIntegerParser` to parse the elements of the `MathString` to `MathIntegers`. The `pop` value is parsed with a `MathIntegerParser` to a `MathInteger` as well. If there are any parsing errors, such as a missing "<" in a `MathString`, the contract throws a `MathFormatExepction`, which is caught by the GUI, and a simple error box is displayed to the user. If the objects are parsed successfully, they are saved to private class variables so that when the `toInstanceString()` method is called, there is no need to reparse the input. The objects are also put into a Java `Vector` and passed to the pre-condition `Assertion` which checks that #self has a length greater than zero. If that condition passes, the `Vector` is then passed to the post-condition shown in Listing 2.2 by invoking the `Assertion`'s `check()` method. `StackPostPopAssertion`'s `check()` method begins by creating temporary copies of the objects in the `Vector` that it can modify. Next, it makes a quick check of #self and self to ensure that self contains exactly one less item than #self. If that is true, it checks to see if #self is equal to self concatenated with `pop`. If that is also true, the `check()` function returns `true`. `StackPopContract`'s `check()` method then returns `true` to the GUI object. The GUI then calls the contract's `toInstanceString()` method and displays the returned `String`, representing the instantiated specification, in the bottom right panel. Finally, the GUI displays a message to the user informing her of whether the test case satisfies the specification.

Log Submission. As a student uses the tool, TCRA keeps a log of his activity. This log can be submitted at any time through the tool to a specific FTP server, where it is stored for the instructor to download later through the instructor side tool. The configuration is handled through the `tcra.config` file which is read when the student selects "*Submit Log*" through the Tools menu. There are three lines in the configuration file:

1. An on-campus IP address (or fully-qualified computer name)
2. An off-campus version of the previous address
3. The directory in the FTP server used to store logs

The reason for having two addresses for the same repository is because of potential firewall restrictions. The server named by the first address may not be accessible from outside the firewall; likewise the second address may be inaccessible from inside the firewall.

The student simply selects which site location he is in via a set of radio buttons in the tools menu and then clicks "*Submit Log.*" TCRA then attempts to connect to the selected address over an SSH tunnel (using the open-source J2SSH library [13]). The student is then prompted for his username and password. After successfully opening the SSH tunnel, an FTP session is started and the tool transmits the user log to the specified directory on the specified system. The tool prepends the ID the student chose on the first run of the tool to the common `tcra_log` filename to ensure that no file overwrites another. If the student has previously uploaded a log, the old one is overwritten. Since the logs create a running record of all activities since the first execution of the tool, there is no data loss. The tool then notifies the student whether the upload completed successfully.

2.4.2 Instructor Tool

Log Data Format. A portion of an example log is shown in Figure 2.4, every student log begins with "id:" followed by the ID selected by the student. Each of the remaining data blocks inside the log begins with one of four codes used to specify what type of event the block records:

- "000" specifies that the program was executed
- "001" denotes the opening of a file
- "010" represents the selection of a method contract
- "1xy" specifies a test case, where the integer xy specifies the number of inputs to the test case.

```
id: new_version
000: Tue Mar 24 19:13:16 EDT 2009: Program executed
001: Tue Mar 24 19:13:27 EDT 2009: User opened file Q2MysteryInterfaceExercise.class
010: Tue Mar 24 19:13:27 EDT 2009: User selected method Mystery1
102: Tue Mar 24 19:13:43 EDT 2009:
#self = <1,2,3>
self = <1,2,3>
Test case was deemed false
104: Tue Mar 24 19:34:47 EDT 2009:
#self = <1, 3, 4>
String of Integer #b = <6, 8>
self = <1, 3, 4, 6, 8>
String of Integer b = <6, 8>
Test case was deemed true
010: Tue Mar 24 19:34:49 EDT 2009: User selected method Mystery2
106: Tue Mar 24 19:35:48 EDT 2009:
#self = <1, 3, 5, 6>
Integer #index = 2
String of Integer #b = <4>
self = <1, 3, 4, 5, 6>
Integer index = 2
String of Integer b = <4>
Test case was deemed true
```

Figure 2.4: Example Log File Data

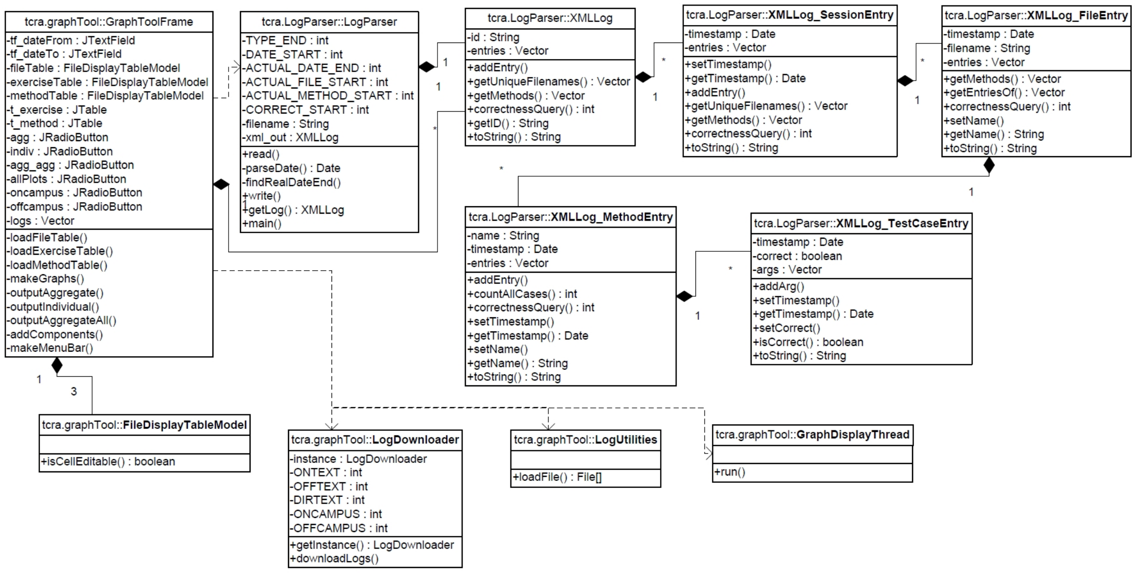


Figure 2.5: TCRA Instructor UML Diagram

```
<TCRA_Log id="jeremiy">
   <Session timestamp="Fri Apr 24 19:32:12 EDT 2009">
      <File name="Q3MysteryInterfaceExercise.class"
            timestamp="Fri Apr 24 19:34:31 EDT 2009">
         <Method name="Mystery1" timestamp="Fri Apr 24 19:34:31 EDT 2009">
            <TestCase timestamp="Fri Apr 24 19:35:48 EDT 2009" correct="false">
               #self = &lt;1,2,3&gt;<br/>
               String of Integer #b = &lt;5&gt;<br/>
               self = &lt;3,5&gt;<br/>
               String of Integer b = &lt;5&gt;<br/>
            </TestCase>
         </Method>
         <Method name="Mystery2" timestamp="Fri Apr 24 19:36:55 EDT 2009">
            <TestCase timestamp="Fri Apr 24 19:38:20 EDT 2009" correct="false">
               #self = &lt;1,2,3&gt;<br/>
               Integer #index = 0<br/>
               String of Integer #b = &lt;9&gt;<br/>
               self = &lt;1,9,2,3&gt;<br/>
               Integer index = 0<br/>
               String of Integer b = &lt;9&gt;<br/>
            </TestCase>
         </Method>
      </File>
   </Session>
</TCRA_Log>
```

Figure 2.6: XMLLog Structure

Each code is followed by a timestamp. 001 blocks include the name of the file that was opened. 010 block include the name of the method selected. 1xy blocks are followed by xy+1 blocks, xy of them being the inputs to the test case, and the last line indicating whether the test case was correct or not.

LogParser and the XMLLog Data Structure. The data structure used to make the raw log data more accessible programatically is built in the same fashion as an XML file. This data structure makes it much easier to query the log and produce the graphs that instructors want to see.

The `XMLLog` object (whose UML diagram is part of Figure 2.5) contains the ID chosen by the student and a `Vector` of `XMLLog_SessionEntrys`. Each session entry contains the Java `Date` of when the session began and a `Vector` of `XMLLog_FileEntrys`. Each file entry contains the `Date` of when the file was opened, the name of the file, and a `Vector` of `XMLLog_MethodEntrys`. The method entry contains the `Date` of when the contract was selected, the name of the method contract, and a `Vector` of `XMLLog_TestCaseEntrys`. The test case entry contains the `Date` of

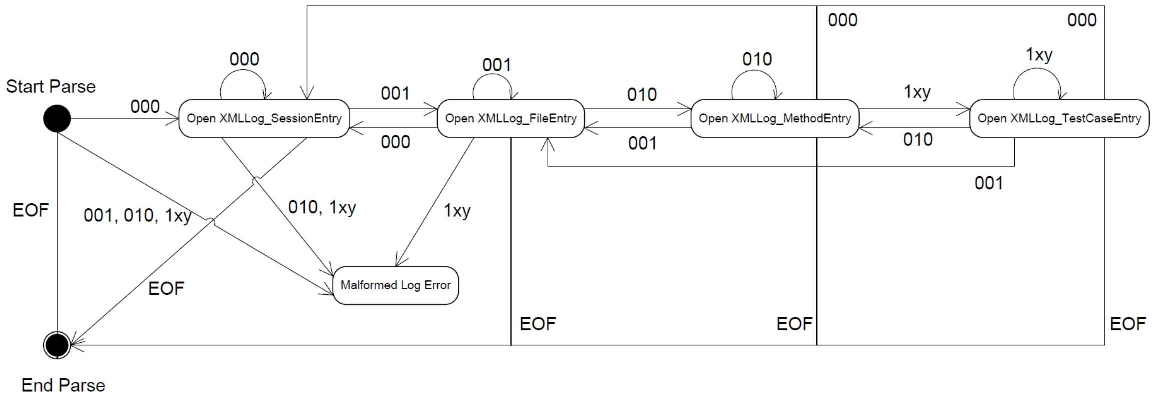


Figure 2.7: State Diagram of LogParser

when the test case verification was attempted, whether or not it was true, and a `Vector` of strings containing the test case inputs. Figure 2.6 shows the output of an `XMLLog`'s `toString()` method and helps visualize the data structure.

One `XMLLog` is generated for each log file opened by the instructor, and each one is used during queries. This process is visualized by a state diagram in Figure 2.7, where the transitions correspond to the block codes read, and the nodes represent the entry type being constructed. The log parsing begins by creating the `XMLLog` object and storing the user id. Next, the parser expects to read a "000" code; if it does read one, an `XMLLog_Session` object is created. The session then stores the timestamp in a `Date` object. If some other code is found, the `LogParser` throws an error, notifying the instructor of a malformed log, which may signify log editing performed by the student.

Next, the parser can read either a "001" code, which leads to the creation of an `XMLLog_FileEntry`, or it may find another "000", signifying that the current session is empty and can now be stored in the `XMLLog`. If the file entry is created, it stores the timestamp in a `Date` and the filename in a `String`. If an unexpected code is found in place of "001," the malformed log error is thrown.

Following a file entry, the parser can accept three different codes. "000" will add the file entry to the session entry, and the session entry will be added to the `XMLLog`. "001" will add the file entry to the session entry, but the session entry will remain current. "010" will create a new `XMLLog_MethodEntry`. The method entry stores the timestamp in a `Date` and the method name in a `String`. If a "1xy" code is encountered, the malformed log error is thrown.

Finally, the parser can accept any code. "000" will add the method entry to the file entry, the file entry to the session entry, and the session entry to the XMLLog. “001” will add the method entry to the file entry, and the file entry to to the session entry. “010” will add the method entry to the file entry. “1xy” will create an XMLLog_TestCaseEntry that adds the timestamp as a Date and reads xy+1 lines, adding each one to the Vector that holds the input values.

Once the XMLLogs are built, getUniqueFilenames() is invoked on each XMLLog and the returned Strings are put in the exercise table found below the list of opened files. Then, after the instructor selects the exercise files of interest and clicks the *Choose Exercises* button, getMethods() is invoked to get all the unique method names from each exercise and place those in the method table. The instructor then selects the methods he or she wishes to graph, enters a date range in the boxes below the table to further refine the query, and clicks the *Generate Graph(s)* button. The tool then invokes the XMLLog’s correctnessQuery() method to get the number of correct and incorrect test cases for the selected method contracts. Several parameters are passed to the correctnessQuery() method; these include: exercise name, method name, correctness (true or false), and the date range. These are passed down the tree, some being stripped off when they are no longer relevant, such as the method entry not receiving the exercise name.

The XMLLog and the contained items also use the toString() method to generate a valid XML file. This is included for instructors who wish to use the XML format for their own unique purposes (e.g. extended analysis). It can be run from the a command line using `java LogParser tcra_log`.

This data collection provides the foundation for our evaluations studies in Chapter 4.

Graph Generation. The instructor tool generates graphs through the use of gnuplot version 4.2 [7]. An input data file is created by querying each XMLLog for the number of correct and incorrect test cases for the selected methods within the designated time frame. These counts are then processed by the graph-specific method in the tool. The receiving method then writes out the input file to gnuplot. The “*aggregate results per method*” option creates one input file per method each containing the number of right and wrong test cases presented by each user. This kind of graph can be used to gauge understanding across the whole class. The “*individual results per student*” option creates an input file for each student, each with the number of true and false test cases per method selected in the GUI. This graph is useful for grading students based on completion of the exercise. The “*aggregate results for all methods*” option generates one input file that contains all the data from the "*aggregate results per method*" plot option. This would allow the instructor viewing the graph to see which exercise was most difficult due to a higher amount of false test cases than average. Gnuplot is then run once for every file generated, using the script found in Figure 6.6 in the Appendix. Finally, in a separate thread, an image viewer is opened to view the newly produced images.

Listing 2.2: Example Assertion - Pop Post-condition

```
public class StackPostPopAssertion implements Assertion
{
   public StackPostPopAssertion() {}

   public boolean check(List<MathType> l)
   {
      MathString<MathType> lb_self = (MathString<MathType>)l.get(0);
      MathString<MathType> self    = (MathString<MathType>)l.get(1);
      MathType             pop     = (MathType)l.get(2);

      if(self.length() != lb_self.length()-1) return false;
      if(!lb_self.equals(self.concat(pop))) return false;

      return true;
   } // end check(List)

   public String toString()
   {
      return("there exists A: String of Object, b:Object<p>" +
             "(#self = A * &lt b &gt )<br>" +
             "(self = A)<br>" +
             "(pop( ) = b)");
   } // end toString()

   public String toInstanceString(List<MathType> l)
   {
      MathString<MathType> lb_self = (MathString<MathType>)l.get(0);
      MathString<MathType> self    = (MathString<MathType>)l.get(1);
      MathType             pop     = (MathType)l.get(2);

      return("there exists A: String of Object, b:Object<br>" +
             "("+lb_self.toHTMLString()+"= A * &lt b &gt )<br>" +
             "("+self.toHTMLString()+" = A)<br>" +
             "("+pop+" = b)");
   } // end toString()
} // end class
```

Listing 2.3: Example MethodContract - Stack Pop Operation

```java
public class StackPopContract extends MethodContract
{
   protected MathString<MathType> lb_self;
   protected MathString<MathType> self;
   protected MathInteger          pop;

   public StackPopContract()
   {
      super("pop");
      precondition  = new StringLengthGTZAssertion();
      postcondition = new StackPostPopAssertion();
      args          = new Vector<MethodArgument>();
      returnType    = new IntegerType();
      lb_self = self = null;
      pop     = null;
   } // end ctor
   public boolean check(String[] l) throws MathFormatException
   {
      MathIntegerParser mip = new MathIntegerParser();
      MathStringParser  msp = new MathStringParser(mip);
      lb_self = msp.parse(new StringBuilder(l[0]));
      self    = msp.parse(new StringBuilder(l[1]));
      pop     = mip.parse(new StringBuilder(l[2]));

      if(lb_self == null || self == null || pop == null)
         throw new MathFormatException(
                    "One or more unparsable arguments.");
      Vector<MathType> args = new Vector<MathType>();
      args.add(lb_self);
      args.add(self);
      args.add(pop);
      return (precondition.check(args) && postcondition.check(args));
   } // end check(List)
   public String toString()
   {
      return("<html>" + returnType + " " + name + "( )<br>" +
             "requires:<br>"+ precondition + "<br>ensures:<br>" + postcondition);
   } // end toString()
   public String toInstanceString()
   {
      Vector<MathType> argVals = new Vector<MathType>();
      argVals.add(lb_self);
      argVals.add(self);
      argVals.add(pop);
      return("<html>" + returnType + " " + name + "( )<br>" + "requires:<br>"+
             precondition.toInstanceString(argVals) + "<br>ensures:<br>" +
             postcondition.toInstanceString(argVals));
   } // end toInstanceString()
} // end class
```

Chapter 3

Related Work

This chapter describes work related to this thesis. The first section describes prior work in teaching rigorous mathematical foundations. The second section surveys existing courseware for computer science education. The third section covers works in collaborative learning. Finally, the fourth section compares tools for teaching specifications with the TCRA tool.

3.1 Teaching Rigorous Foundations

The importance of teaching mathematical techniques to complement traditional programming skills is well-documented by the community [8, 2, 18, 25, 33, 31, 32, 24, 29]. One of the most well-known efforts is that of Henderson [10, 11], whose approach to teaching introductory computer science involves weaving topics in discrete mathematics, problem solving, and algorithm design to emphasize their interconnections. He advocates for an introductory class that models those of other engineering disciplines, where the foundations are taught prior to the main discipline concepts. The instructor would cover general problem solving, discrete math, algorithmic problem solving, basic computer science concepts, and establish links between these areas to show how they support one another. Our approach shares his emphasis on assertion-based program reasoning, including the use of formal pre-conditions and post-conditions.

LeBlanc and Leibowitz [18] advocate a need for more discrete math content for computer science students. They argue that one-semester of exposure to discrete mathematics is insufficient for students to truly understand formal systems, proofs, and discrete structures in computing. They propose a two courses of discrete math. The first course would introduce students to several topics, such as sets, relations, graphs, and logic, that will be repeated in the second course. These topics are also related to their uses in computing. The second course teaches students about proofs and algorithms that are associated with the topics taught in the first class. The authors believe that this repeated exposure will give students a deeper understanding of discrete math topics and by relating them to computing, a better chance of using these topics effectively in computer science classes. This also relieves computer science instructors of having to cover the topics in detail.

The notion of teaching discrete math and programming together is supported by McMaster et al. [25]. They also argue for a two semesters of discrete math and divide each course into *modules*. Each module consists of a topic, such as "Sets, functions, and properties of integers," and a programming project that exercises these topics. Course evaluations suggest that students enjoy and benefit from from this kind of course structure.

3.2 Teaching Tools

The pervasive availability of network-enabled desktops and laptops creates new opportunities for enhancing computer science education with interactive courseware [30, 16, 28, 9, 21, 3]. One of the most successful efforts is led by Kumar [16, 17]. His *problet* approach involves computer-based tutors that guide students through short program reasoning exercises that reinforce their understanding of programming language semantics. Kumar has developed several such problets, covering topics such as: expression evaluation, predict loop output, and identification of syntax errors. Each problet provides feedback to the user that explains the answer to the problet and how to arrive at the answer. While our focus on formal reasoning distinguishes our work from that of Kumar, the tools are similar in design. TCRA provides the student with a formal contract and accepts test cases that are used to instantiate the contract. There is no pre-defined answer to the problem, so TCRA's feedback is limited to stating correctness of the test case. It was designed not to identify the root cause of an error in the instantiated contract in order to encourage students to continue testing until they succeed. The system for collecting student response data for instructor analysis is also similar. Both approaches benefit from the rapid response feedback common to *clicker-based* instructional techniques [28], but provide more detailed guidance in the event of reasoning errors. Kumar's problets have three levels of feedback: *none*, where the tutor tells the student to continue to the next problem; *minimal*, here correctness result is provided by the tutor; and *detailed*, where the tutor explains the answer in a graphical or textual fashion. As stated above, TCRA's feedback is limited to providing a boolean correctness result and an instantiation of the contract under test.

Rodger et al. [35] have piloted *JFLAP* [34, 30] to make automata theory more interactive. JFLAP was designed to enable students to practice problems that would be difficult with traditional pencil-and-paper methods, which is a goal shared by our tool. The authors note that automated test case verification relieves the instructor from the burden of grading hand-written assignments. JFLAP can be used to build and simulate several types of machines such as finite automata, push-down automata, and Turing machines; it can also convert from one machine to another. Students are able to enter input strings and observe the behavior of the automata. Students can also test their ability to convert from one machine to another by performing the conversion by hand and then comparing against the output of the tool.

TCRA provides similar functions for students practicing formal reasoning. The tool verifies the student's input and rapidly provides feedback to the student. The feedback comes in two forms: the student is told whether or not their test case is correct and the specification is instantiated with the student's input. The specification instantiation enables the student to determine where their error lie without explicitly highlighting the error for the student. This was designed to encourage the student to rethink the specification rather than simply continuing on with to the next exercise.

3.3 Collaborative Learning

The *Peer Instruction* approach pioneered by Mazur [23] is based on the concept of learning through one's peers and has proven effective through over a decade of experimentation. Though the approach was originally developed in a physics context, the approach has more recently been applied successfully in computer science classrooms by Chase and Okie [4], among others. Their experiment compared two sections of an introductory computer science course, one taught traditionally and the other using collaborative learning techniques. The experimental course had peer instructors that helped students during labs and the students were assigned to groups during normal class meetings. Group members were responsible for helping each other and for motivating each other to remain in the class. The authors noted that the WDF rate (percentage of students withdrawing, or receiving a D or F in the class) dropped from 56% to 32% in one experiment and 33% in the other. Many of these students received instead a C. Female students seemed to respond better to this kind of class, as the female WDF rate dropped from 53% to 18% in one section and 12% in another.

Beck and Chizhik [1] found similar results in their classroom experiment. The control section of the course used the traditional lecture-based format while the experimental section used shorter lectures combined with mini-lectures, administrative topics, and general class discussion. Students in the experimental section also also worked together to simulate program execution. This was done whenever new concepts were introduced. The authors found that all students in the experimental section benefited from the cooperative learning, but female and minority students displayed greater improvement over their peers in the control section than white male students did.

3.4 Specification-Based Tools

Few tools have been developed that enable instructors to teach formal specifications in an interactive manner. Feldman [6] surveyed several tools in hopes of using them in an educational setting. Many of them were similar in function, so only *iContract* [15] is summarized below. The other tools were not discussed in detail as they were found to be inappropriate for classroom use.

iContract is a tool that instruments Java source code with assertions that check pre- and post-conditions specified by comments in the source files. A developer adds annotations to their source code to specify pre-conditions, post-conditions, and invariants. The iContract preprocessor creates executable assertions within the annotated functions, which are in turn tested during program execution. If a contract is violated, a `RuntimeException` is thrown describing the violation.

While iContract seems to be useful for runtime checking, it seems limited in its usefulness in education. To develop test cases for exercises written in iContract, a student would need to build a test driver for the class in the exercise. This could be time-consuming and would focus more on the students' programming ability and less on their understanding of specifications. Also, if a student needs to read source code to develop test drivers, they may be able to trace code to develop a correct test case instead of understanding the specification's details. Students also lose sight of the abstract mathematical models that form specifications if they are able to read and infer behavior directly from code.

The most serious problem with iContract is that the annotations do not accurately represent the formal mathematical models. Consider an example:

```
    @post has(k) == ($ret != null)
```

As is obvious in the example, the annotations make reference directly to functions and other programmatic elements, instead of the underlying models. This distracts from the formal mathematical contracts being taught and may lead students to believe that this is proper syntax in formal mathematics. Additionally, according to Feldman, students who used iContract found it difficult to install, causing loss of valuable class time [1].

TCRA has been developed to overcome many of the problems associated with simple code instrumentation. It enables instructors to create exercises that focus on abstract mathematical models and supports arbitrary specification languages. Exercises have been developed to use Java-style specifications as well as RESOLVE-style specifications. Student test case input has been designed for fast creation, testing, and modification. The syntax used in the test cases comes directly from the mathematical models it represents. For example, if a student needs to enter the state of a string of integers, they might enter "`<1,2,3>`." The students simply enter each test case and click a button to verify their entry. There is no need to develop test drivers, which saves time, enabling rapid exercise completion.

TCRA's installation procedure is straightforward; it requires the extraction of an archive file to a directory of the user's choice. Scripts that execute the tool either on Windows- or Linux-based systems are included. The scripts set the `CLASSPATH` at the time of execution, further simplifying the install process. Since TCRA is written in Java, it is available on any platform that has an associated Java Virtual Machine. It has already been tested on Windows XP and Vista, several versions of Linux, as well as Mac OS X.

The benefits that come with TCRA do, however, come at the price of increased effort on the part of the exercise developer. Instead of simply instrumenting source code with annotations, the developer must implement an exercise class using the TCRA exercise API.

[1]iContract appears to no longer be available.

Chapter 4

Evaluation

In this chapter we describe experiments performed to evaluate the tool's benefit to students in classroom situations. Specifically, we focus on evaluating: student performance on quizzes with and without TCRA and the students' ability to identify their mistakes in test cases while using TCRA.

4.1 First Classroom Experiment

We first focused on determining whether student performance improved when using TCRA to complete class quizzes.

Experimental Design. TCRA was used in an undergraduate software engineering class at Clemson (CPSC 372) where the RESOLVE specification approach [37] was taught. The experiment was conducted after students had bee introduced to RESOLVE notation simple specifications. Each day, students received a quiz to test their knowledge of the material covered in the preceding class. Each quiz required that students present two test cases for each specification. The experiment began with a quiz (Q1) before students used TCRA. Q1 was used to create a baseline of student understanding of specifications. After the quiz, students were split into two groups selected non-deterministically, one of which was given the tool to use for the next quiz (Q2), while the other was not. In the next class, the students with the tool were allowed to use it to complete Q2, while the rest of the students used the traditional pencil-and-paper method. Q2 allowed us to directly compare student

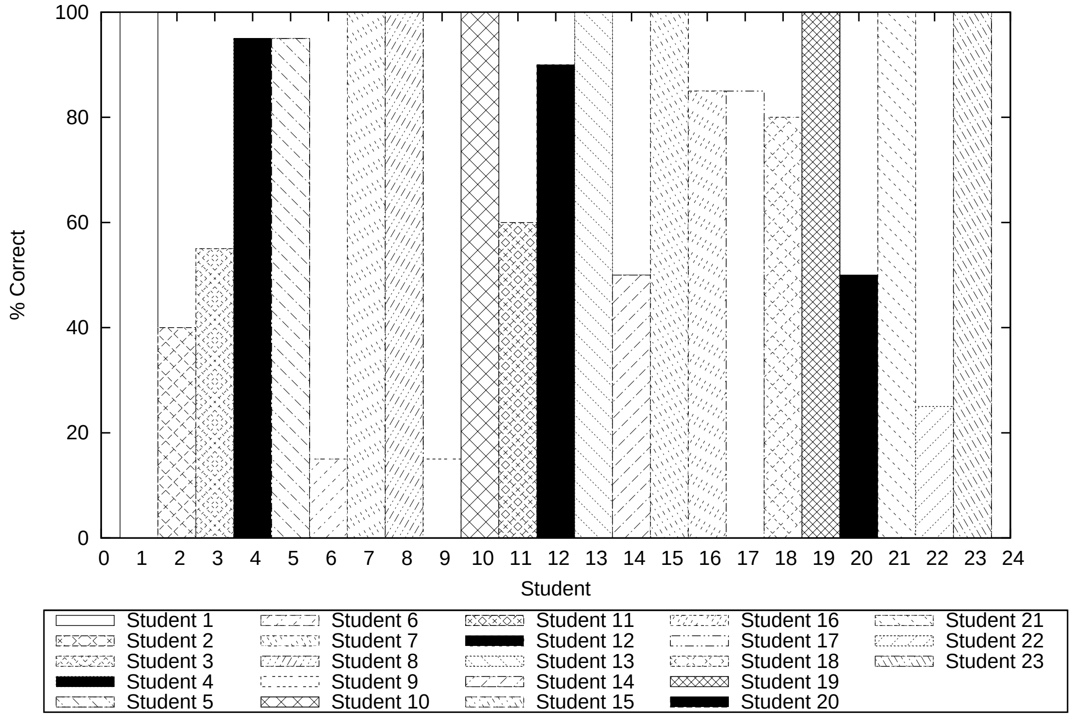


Figure 4.1: Q1 Scores (Pre-Tool)

performance with pencil-and-paper exercises and performance with TCRA. After the quiz, the rest of the students were given the tool. For the last quiz (Q3), all students used the tool to complete the quiz [1].

Results. First we consider quiz scores. These measurements allows us to determine whether the use of TCRA improves student quiz scores. Q1, the quiz administered before the tool was distributed to the class, had an average score of 75%. Q2, for which half the students used the tool, had an average of 84%. The students who used the tool averaged 94%, while those without the tool averaged 78%. For Q3, the average was 90%. In Figures 4.1, 4.2, and 4.3, student scores have been individualized for better visualization of performance.

Next, we consider the correctness of student provided test cases over time. This allows

[1]The specifications included in Q2 and Q3 can be found in Listing 6.1, in the appendix of this document.

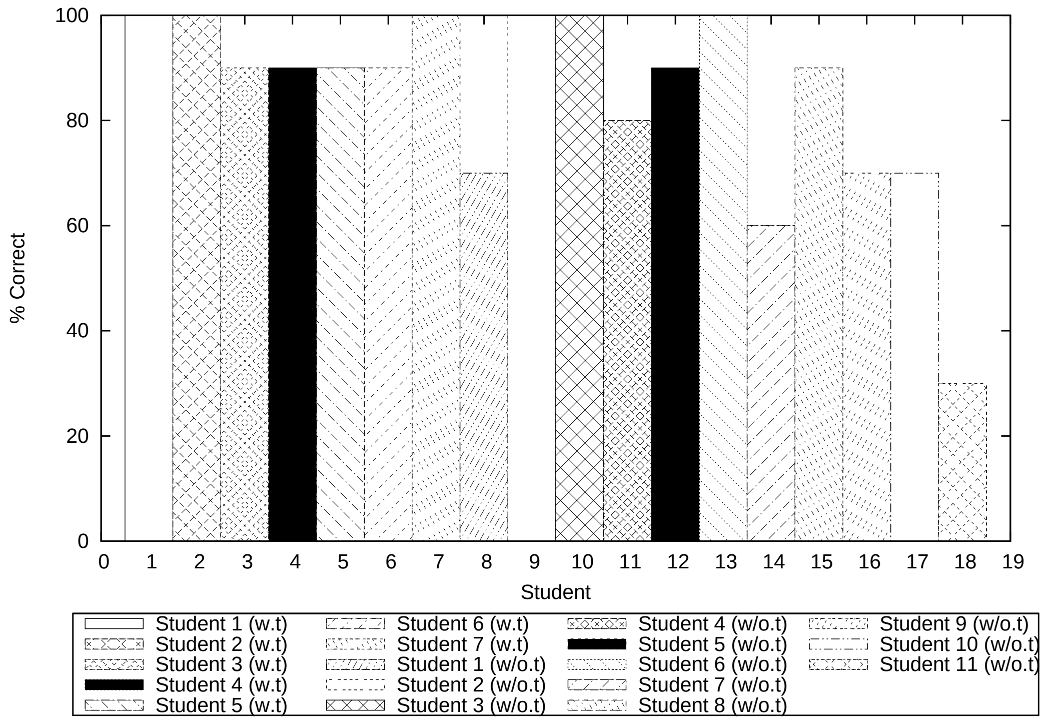


Figure 4.2: Q2 Scores (Some Using TCRA)

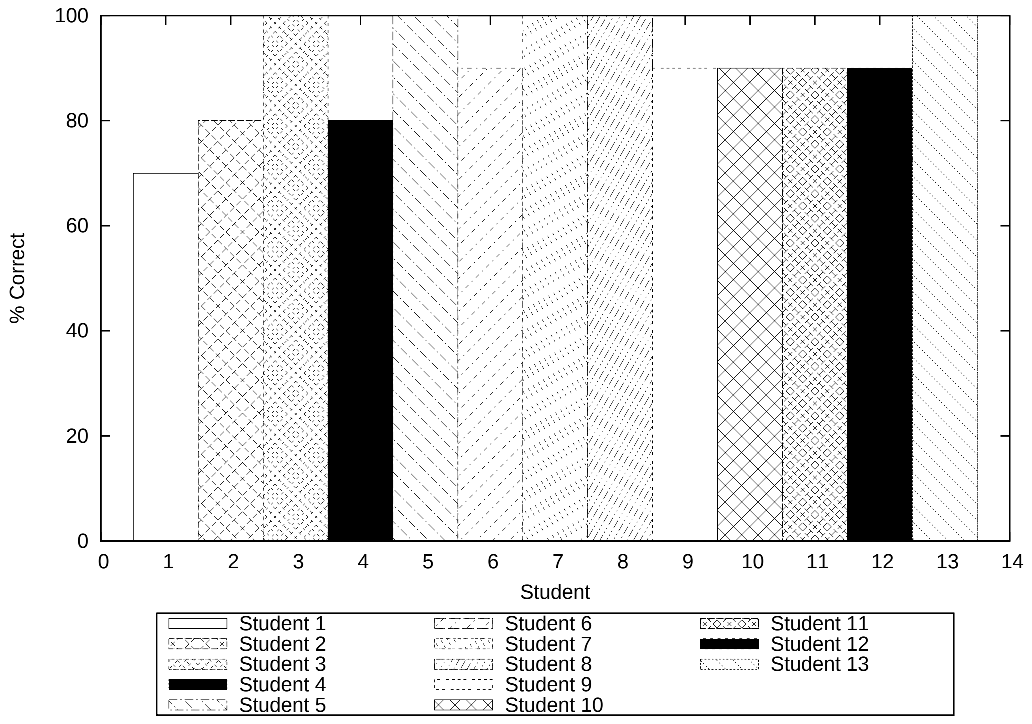


Figure 4.3: Q3 Scores (All Using TCRA)

us to determine whether fewer incorrect test cases were presented as students used the tool. This information is not yet available through the instructor view of TCRA, so the student logs had to be manually examined and the data analyzed in a spreadsheet. For each test case, one point was awarded for each true test case and zero for each false test case. The average of this score was calculated and plotted after each test case, as summarized in Figures 4.4 and 4.5. The vertical axis represents this average of test cases that are correct. The horizontal axis shows the number of attempts made over the course of the quiz. The legend shows the correspondence between line type and student. This kind of data could not be created for Q1 as there is no history data available from pencil-and-paper exercises. Both figures show similar results; the line plots begin high as students were able to reach a correct test case for the first specification on their first attempt. From there, the lines drop as students proceed onto the next specification and present several incorrect test cases. This is followed by a small spike upwards when a student finally creates a correct test case. As students progress through the quiz, this long down slope and short up slope is repeated. This shows that students continually require several attempts before reaching a correct test case, and once students achieved a correct test case, they moved on to the next specification.

The final metric we looked at was the average inter-arrival rate of test cases. This was measured as the number of seconds between test cases entered by the students. This allows us to determine whether students presented test cases more rapidly as they used TCRA more. Again, this type of data is not yet available in the instructor view of TCRA, so the data analysis had to be performed via spreadsheet. The number of seconds between each test case was recorded and an average was taken after each test case. The plot of these averages shows if a student uses more or less time to enter a test case as they progress through the quiz. These plots can be seen in Figures 4.6 and 4.7. On these graphs, the vertical axis represents the average number of seconds between test case entries. The horizontal axis measures the number of test cases entered. The legend shows the correspondence between line type and student. As the figures show, most students began with a very high average which quickly fell as they progressed. This suggests that students require "warm-up" time to get used to the tool and read the specifications, but then begin to rapidly enter test cases as they become more comfortable with the tool.

Other Observations. The logs received from students showed that even after presenting a successful test case for a given specification, all of the students continued to test the specification with

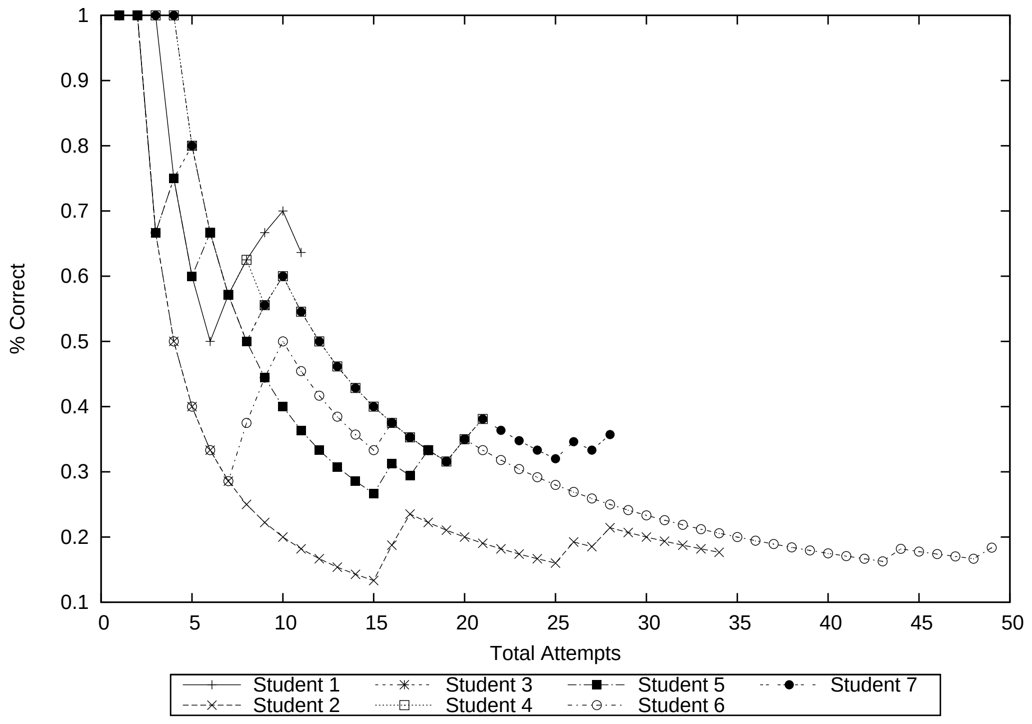


Figure 4.4: Correctness of Test Cases Over Time for Q2.

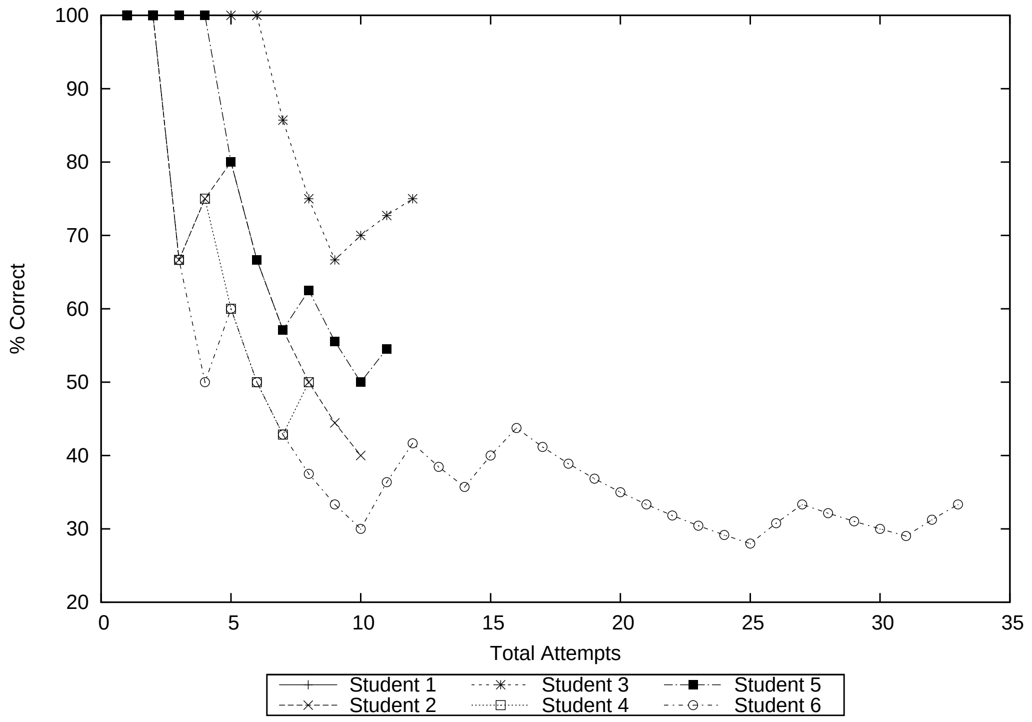


Figure 4.5: Correctness of Test Cases Over Time for Q3.

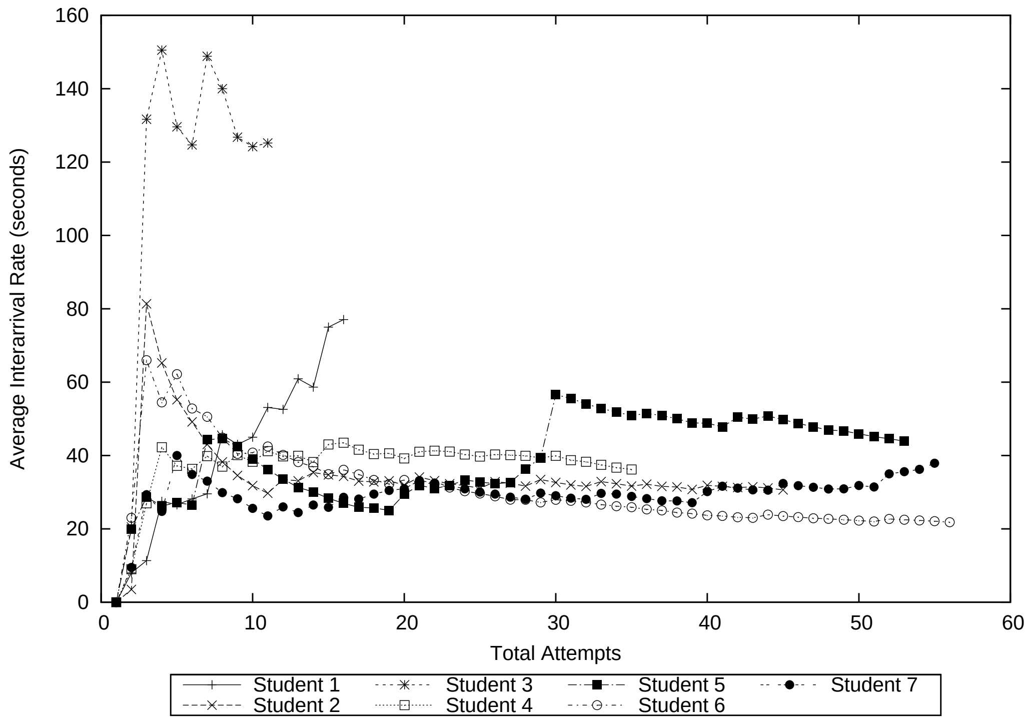


Figure 4.6: Number of Test Cases Submitted Over Time for Q2.

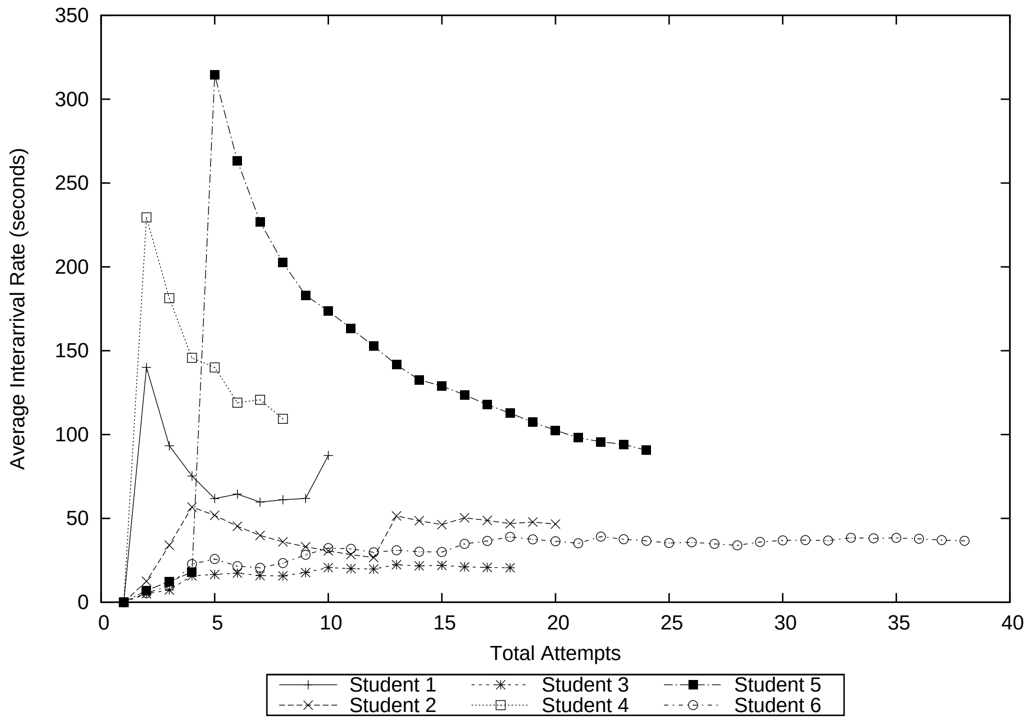


Figure 4.7: Number of Test Cases Submitted Over Time for Q3.

at least one additional test case. Many students presented both true and false test cases after the first success. We postulate that the students did this to gain a better understanding that would be unavailable to them without an interactive tool.

One of the questions in Q3 had no possible solution. The specification contained a small contradiction that was difficult to recognize. The tool logs show that this question was tested by more than half the students with more than ten test cases. Two students presented almost 40 test cases each. This would not have been possible via the pen-and-paper testing method without a considerable time investment.

4.2 Group Experiment

Overview. Next we focus on determining how student performance changes if exercises are completed in a group.

Experimental Design. The students from CPSC 372 were split into eight groups to test the *One Way List* specification (Listing 6.2 in the appendix) using the tool. The exercise included ten method contracts, as opposed to the three or four found on the quizzes. To increase the difficulty, the One Way List data structure was not taught prior to the project; this way no prior knowledge of the structure could be used to ignore the specifications' details.

Results. The same metrics measured in the individual quiz logs were also measured for the group project logs: score, correctness over time, and average inter-arrival rate over time. In terms of score, every group made received 100% of the total points. In contrast to the quiz results, the majority of the groups' correctness average improved with time, as seen in Figure 4.8. The inter-arrival rate trends are also opposite those found in the quizzes, as seen n Figure 4.9. All but one of the groups had an increasing inter-arrival rate. We believe that the discussion that takes place in groups explains the inversion of the two trends. Discussion takes place before the test case is presented to the tool and modifications are made, causing the test case to take longer to present, but more likely to be correct.

Other Observations. One group log appeared to have been a mixture of several individual logs. This lead us to believe that the group did not work together as instructed, but rather divided up

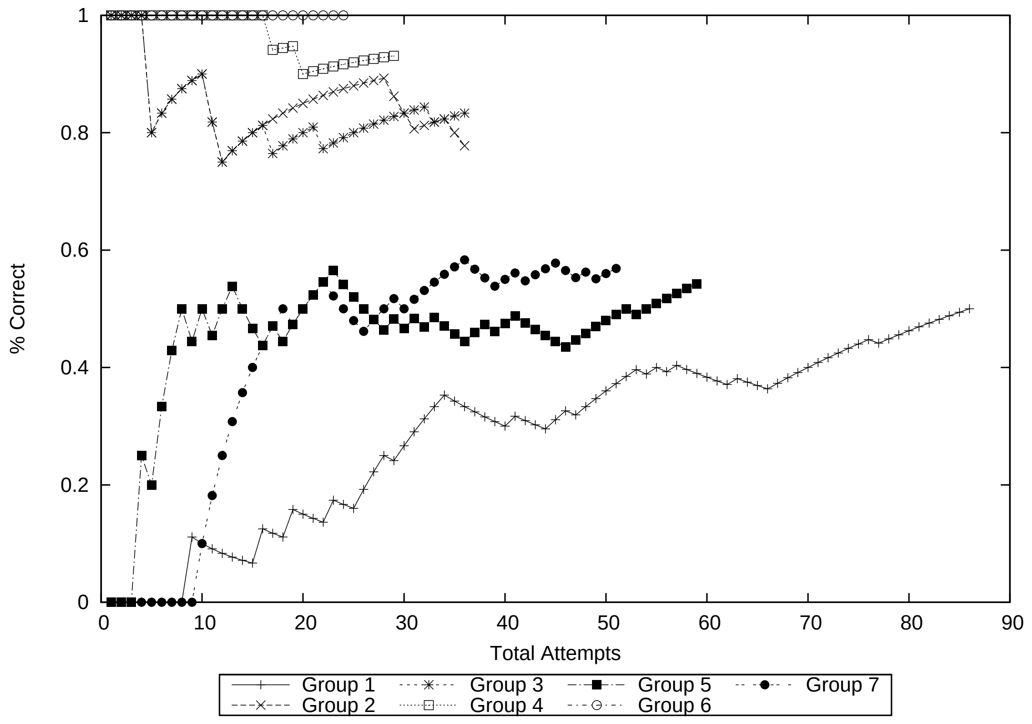


Figure 4.8: Correctness of Test Cases Over Time for Group Exercise.

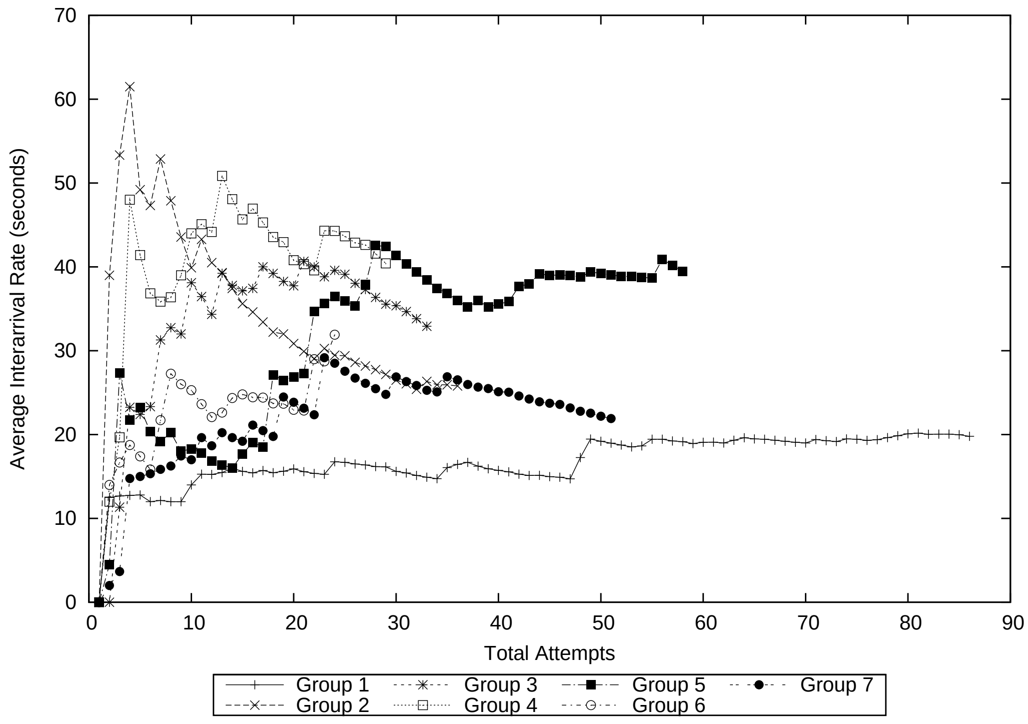


Figure 4.9: Number of Test Cases Submitted Over Time for Group Exercise.

the specifications and then combined the log data into one log. This log was not included in the inter-arrival and correctness over time analysis.

4.3 Second Classroom Experiment

Overview. Finally, we focused on validating our findings from the first experiment by repeating the three-quiz evaluation.

Experimental Design. TCRA was used as an exercise during both sections of the CPSC 215 closed lab. The experiment was conducted in the second half of the course after the basics of specifications and JUnit [14] had been introduced. The exercise used in the experiment consisted of a set of three quizzes similar to those used in the first classroom experiment. All the quizzes presented five "`Mystery()`" method contracts so that students could not infer behavior from the method name[2]. The specification difficulty increased over the course of the five questions. First, a paper-based pre-quiz (Quiz 1) was given to every student to gauge non-tool performance. Next, the students were split into two groups. One group took the second quiz (Quiz 2) on paper, while the other used TCRA to enter test cases. The third quiz (Quiz 3) was completed by all students using TCRA.

Results. The average score across both lab sections for the first quiz was 37%. The average score on the second quiz for those who took the paper quiz was 66% while those who used TCRA averaged 98%. The average for the third quiz was 98%. The individual student scores for Quiz 1, Quiz 2 and Quiz 3 are shown in Figures 4.10, 4.11, and 4.12, respectively.

The same two metrics analyzed in the first experiment, test case inter-arrival rate and percent correct over time, were also analyzed in this experiment. Figures 4.13 and 4.14 summarize the "percent test cases correct over time" data collected from student logs. This data was collected and analyzed in the same manner as described in Section 4.1. As the figures show, a similar pattern of long downward slopes followed by short upward slopes is present. This shows that, just like the students in the first experiment, each specification required several incorrect attempts before an correct test case was achieved. Also, as the difficulty increased, the number of incorrect test cases

[2]The specifications included in the three quizzes can be found in Listings 6.3, 6.4, and 6.5, in the appendix of this document.

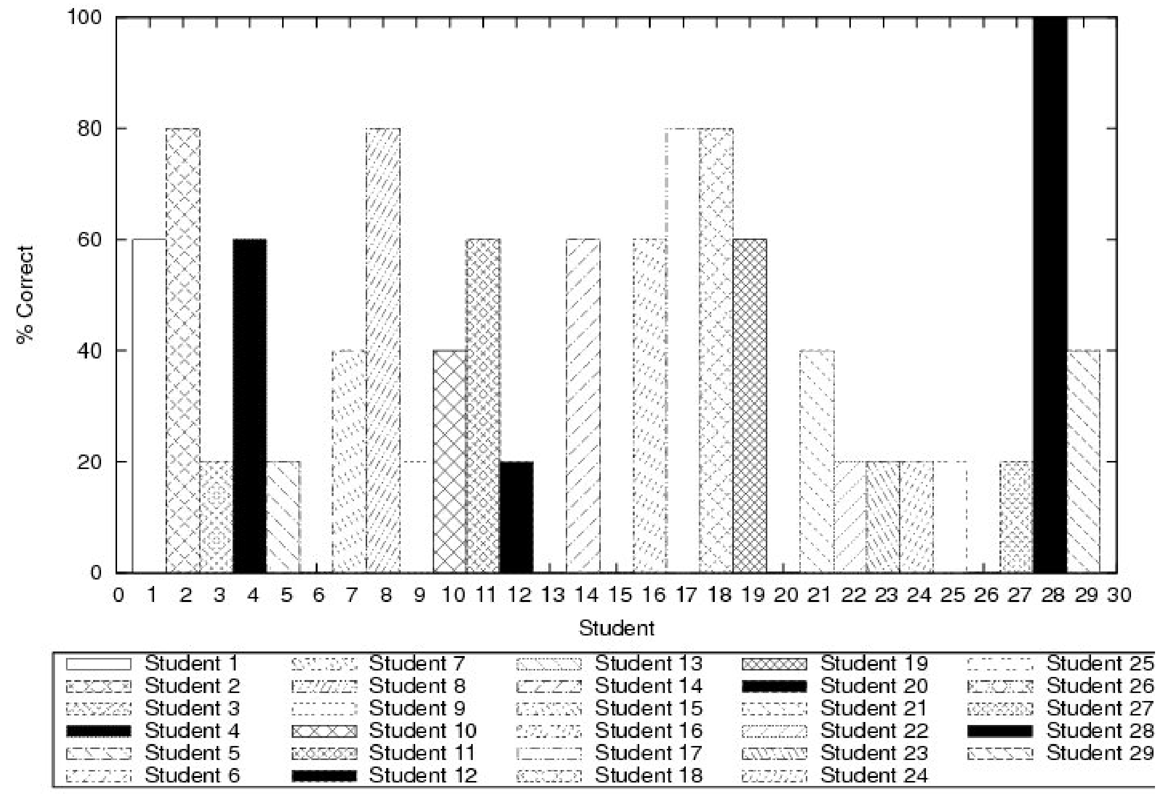


Figure 4.10: Quiz 1 Scores (Pre-Tool)

entered before a correct one also increased.

Average inter-arrival rates of test cases were also analyzed. These too follow the trend set by the first experimental group. The rates begin high and fall quickly as students progress through the quizzes, as shown in Figures 4.15 and 4.16.

Other Observations. It is worth noting that a few logs included rapid entry of the same, incorrect test case several times. During the lab, students could be seen repeatedly clicking the "OK" button, vocally insisting that their test case was correct. These occurrences were rare; we believe that they did not alter our results.

4.4 Threats to Validity

Course Structure. The two experimental groups had several variations that could threaten the validity of this study. The two courses (CPSC 372 and CPSC 215) were taught by different instruc-

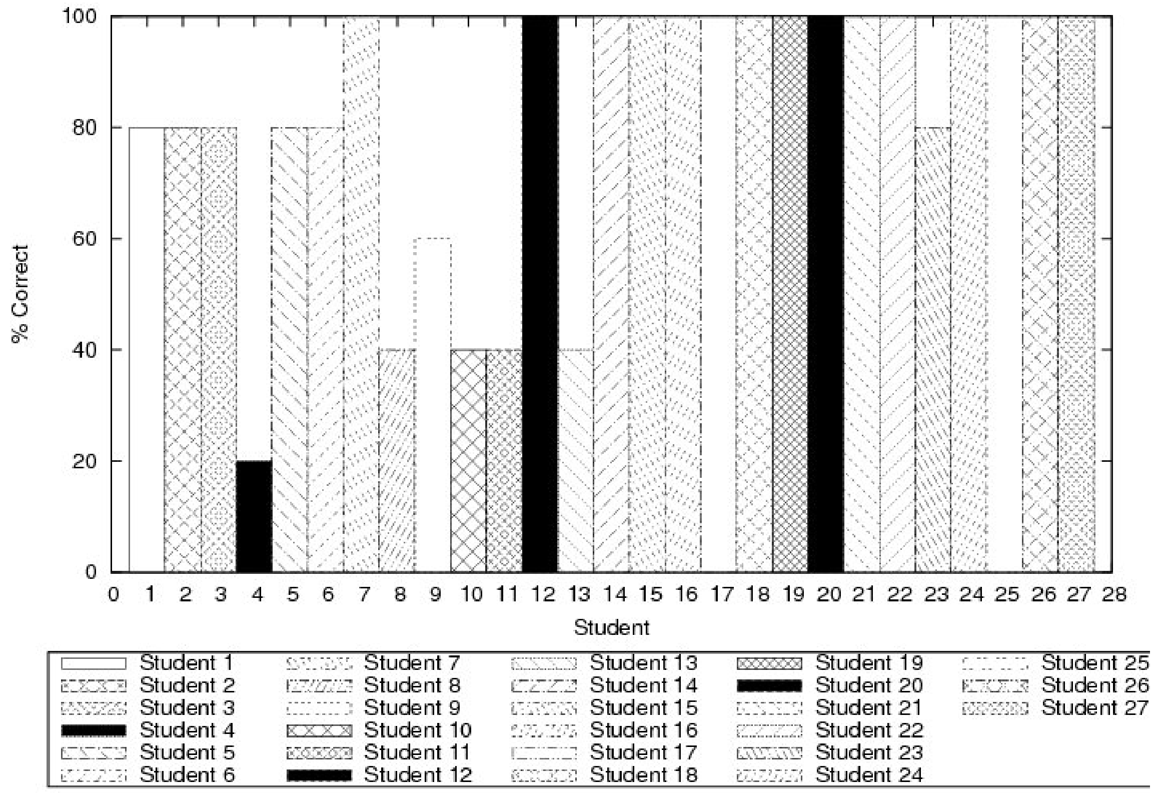


Figure 4.11: Quiz 2 Scores (Half Using TCRA)

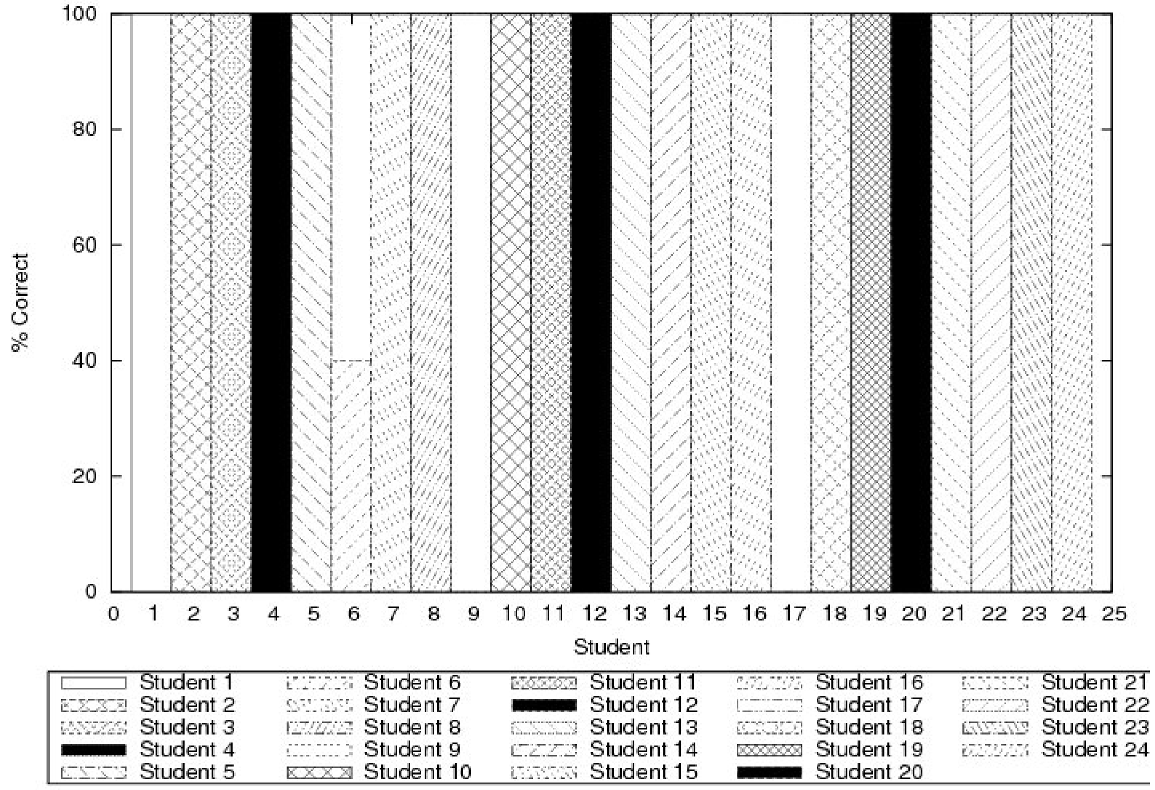


Figure 4.12: Quiz 3 Scores (All Using TCRA)

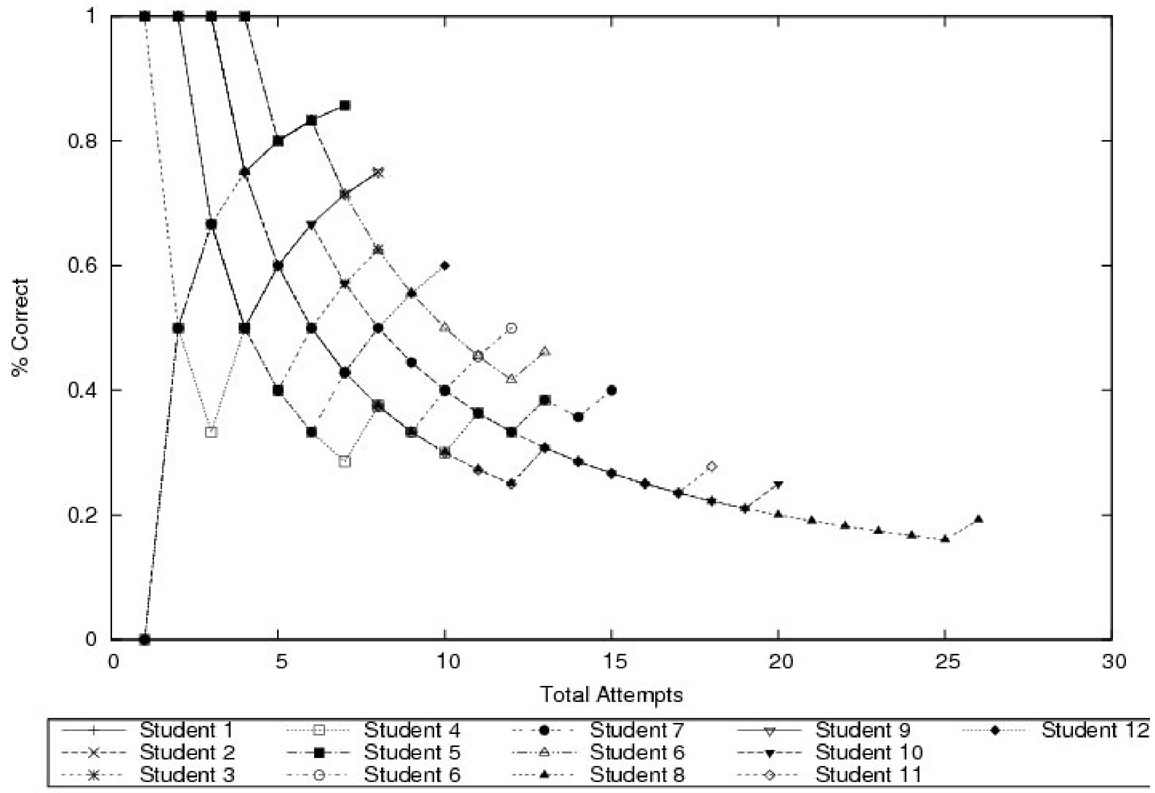


Figure 4.13: Correctness of Test Cases Over Time for Quiz 2.

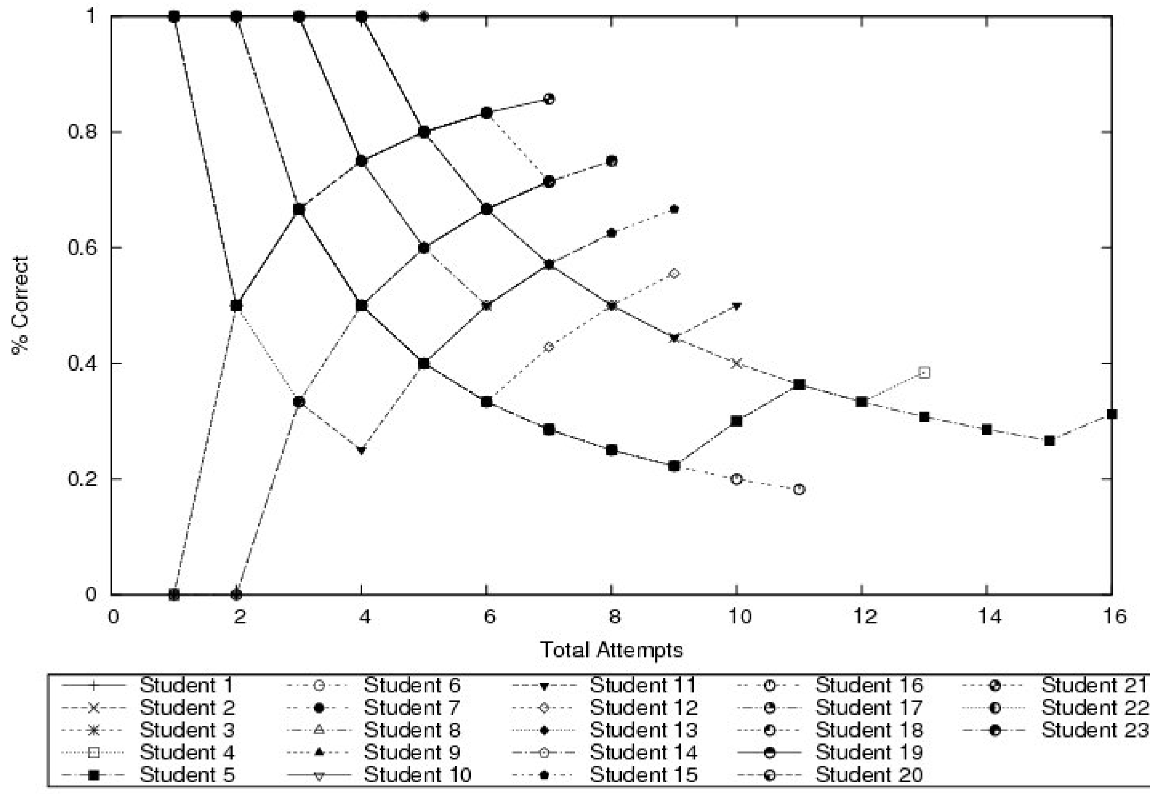


Figure 4.14: Correctness of Test Cases Over Time for Quiz 3.

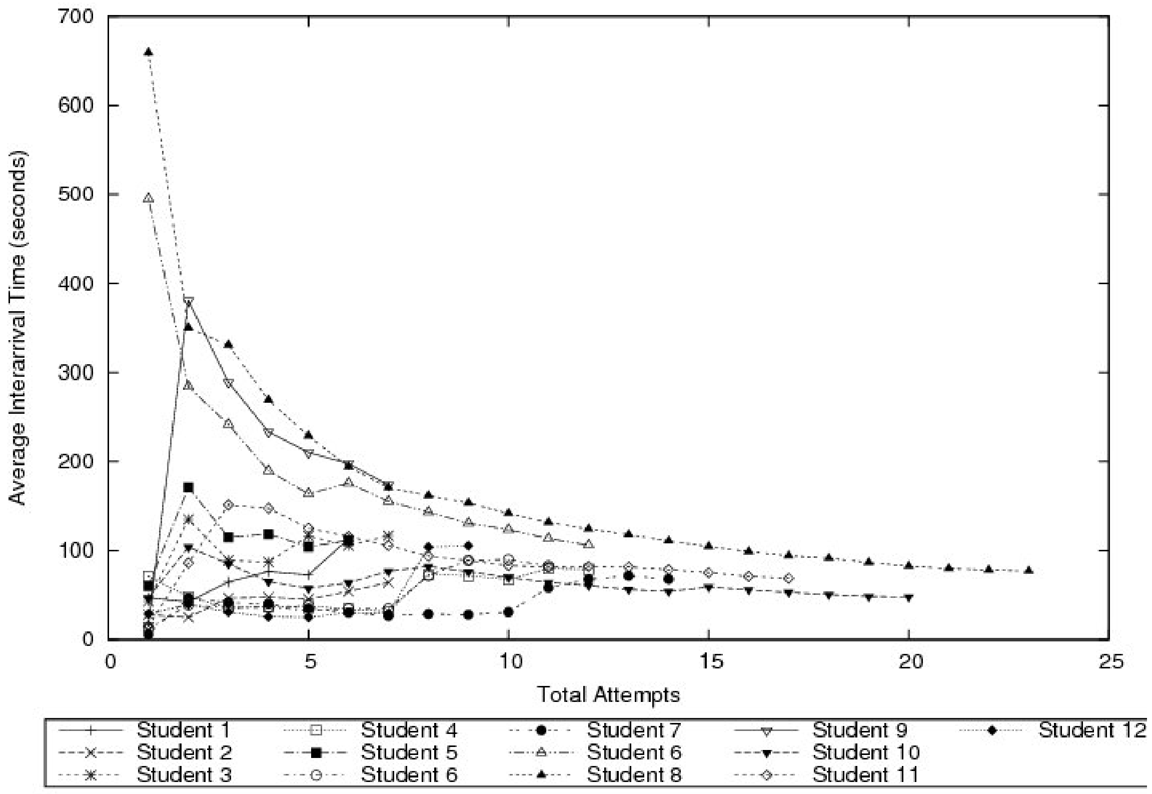


Figure 4.15: Number of Test Cases Submitted Over Time for Quiz 2.

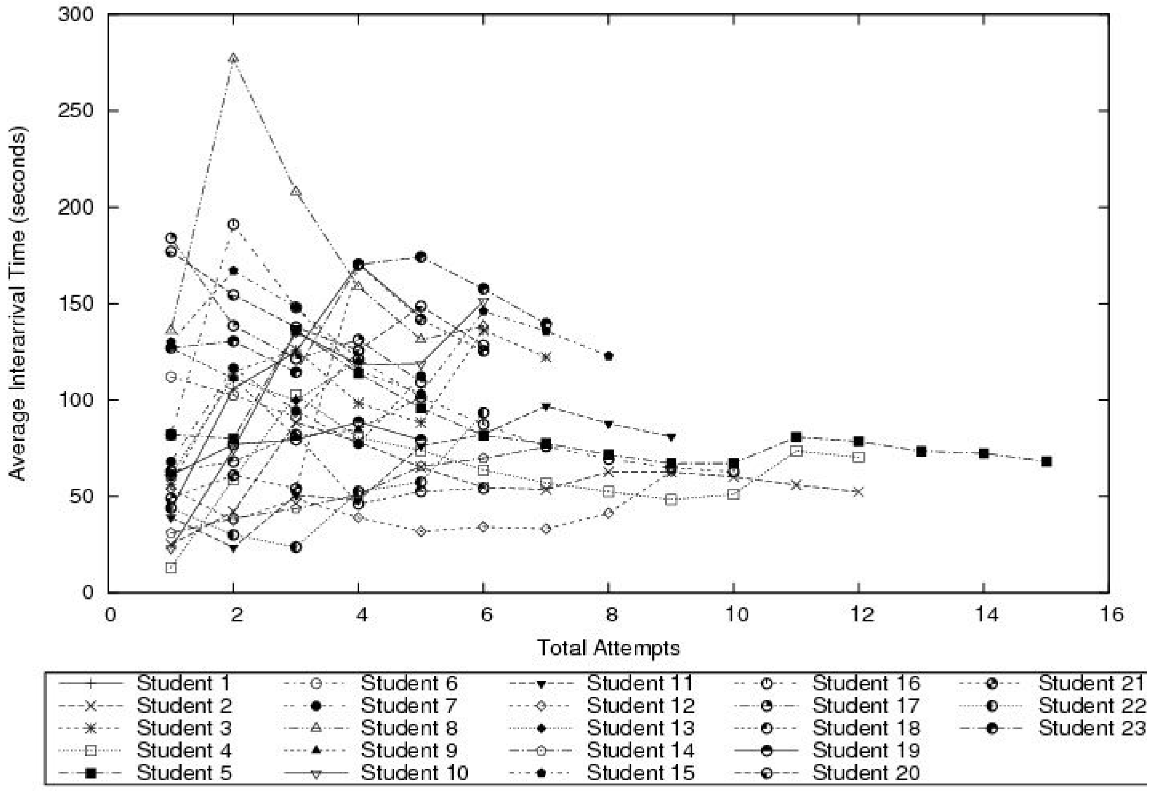


Figure 4.16: Number of Test Cases Submitted Over Time for Quiz 3.

tors, who's teaching styles could effect how well the students understood specifications before using the TCRA tool. Also, these two courses have students of differing skill levels. CPSC 215 is mostly first semester sophomores, just coming from the basic programming classes in the curriculum while the students of CPSC 372 are mostly juniors and seniors who may have taken theory courses in prior semesters, possibly giving them advanced knowledge of specifications. Also, many of the students in CPSC 372 may have already taken the discrete math course required for graduation, while most sophomores have not.

Sample Size. The Computer Science 372 class during the spring semester had only 22 students enrolled. This class size is not large enough to make statistically significant claims regarding the data collected. To worsen this, not all students attended class every time one of our quizzes was taken and some students did not submit their logs, though the reason is unknown.

The Computer Science 215 lab used in the second experiment had only 30 students across two lab sections. One log received from this group seemed to not have attempted any of the tool-based questions and another did not attempt quiz 3 at all.

Log Data Editing. One log file submitted by a group showed evidence that the file was constructed from several individual logs. This was identified by the instructor tool as a malformed log and was ignored during analysis.

Chapter 5

Conclusions and Discussion

This chapter summarizes the problem addressed by this thesis and the corresponding contributions. Key elements of future work are also discussed.

5.1 Summary

Prior to the work in this thesis, there were inadequate tools to enable students to practice their formal verification skills and receive immediate feedback, and enable instructors to monitor student performance during practice. Instead students traditionally rely on pen-and-paper methods of practice and instructor feedback when it becomes available. Likewise, typically instructors have only tests, homework, and student exercises to monitor student performance. This thesis addresses the following questions:

- How can courseware help students study and practice software specifications more effectively?
- How can courseware help instructors better gauge student understanding of specifications?

TCRA addresses the first problem through the student interface. It enables students to enter test cases that are checked against instructor-provided specifications. TCRA then provides immediate feedback to students based on the correctness of those test case. These test cases are based on instantiations of mathematical models, such as <1,2,3,4>, an instantiation of the String of Object model. The specification implementations separate their checking logic from their display logic, enabling instructors to use whatever specification notation they wish with no dependency on how the test case input is checked.

The second problem is addressed through the instructor interface of TCRA. The instructor tool uses student logs recorded by the student interface to create graphs that summarize student performance. These logs contain information that is not obtainable through paper exercises. Instructors can use this extra information to better understand student performance and understanding.

The evaluation data suggests that the reasoning module, and the TCRA tool in particular, had a positive impact on student performance in the current pilots. This appears to be evidenced both by quantitative student quiz data and qualitative analysis of TCRA logs. The benefits seem to be enhanced when students work collaboratively. We emphasize that while the evaluation results are promising, they are only suggestive.

5.2 Future Work

While TCRA is an operational tool, there are areas open to improvement. The API needs to be extended to support more advanced mathematical models and data types. The instructor interface needs more graph types included to extend its log analysis capabilities.

An API extension will allow instructors to create new exercises without having to develop their own mathematical models. This will reduce the development time for new exercises and encourage adoption in other courses. The new data types in this extension will allow for a richer set of test cases to be entered. Students will be able to use not only integers, but characters, text strings, and others.

Extending the graphing capabilities of the instructor interface will greatly enhance the functionality of the tool. More complicated analyses, such as the ones used in this thesis, will allow instructors to summarize time-based data, such as the inter-arrival rate of test cases. These extensions would also include new gnuplot scripts to perform the actual data plotting.

Operational Reasoning Assistant A second tool in development is the Operational Reasoning Assistant (ORA), which targets introductory programming courses and extends interactive courseware further through the curriculum. This tool displays a section of code in a programming language of the instructor's choice. As the code is executed, students are asked to provide the value of all the variables in scope as well as to identify which line will be executed next. If the values provided by the student are correct, execution continues to the next line. If not, student must identify their error and correct it. When execution continues, the previous values of variables are displayed in their own pane. This removes the need for students to track variable values on paper. This courseware allows students to practice code tracing while receiving the same feedback found in TCRA. TCRA's logging system will also be included to track student performance.

The work presented in this thesis is based on results previously reported in [19, 36].

Chapter 6

Appendix

Listing 6.1: Specifications for Experiment 1 Quiz

```
Operation Mystery1(updates S: Stack);
   ensures S /= empty_string;

Operation Mystery3(replaces E: Entry; updates S: Stack);
   requires |S| > 0;
   ensures <E> o S = #S;

Operation Mystery5(updates E: Entry; updates S: Stack);
   requires $|S| >$ 0;
   ensures There exists F: Entry, Rst: Str(Entry) such that
           $<F> o Rst = $\#S and $<E> o Rst = S$;

Operation Mystery6(updates S, T: Stack);
   requires |S| > 0 and |T| > 0;
   ensures There exists E: Entry such that <E> o S = #S and
           <E> o T = #T;

Operation Mystery7(updates S, T: Stack);
   requires |S| > 0 and |T| > 0;
   ensures There exists E: Entry such that
           Rev(S) o T = Rev(#S) o #T and <E> o S = #S;
```

Listing 6.2: Specification of One Way List

```
Concept One_Way_List_Template(type Integer, eval Inital_Capacity: Integer);
uses Std_Integer_Fac, String_Theory;
Family List_Position is modeled by Cart_Prod Prec, Rem: Str(Entry);
end;
exemplar P;
initialization ensures P.Prec = empty string and P.Rem = empty string;

Operation Advance(updates P: List_Position);
requires: P.Rem /= empty_string;
ensures: P.Prec o P.Rem = #P.Prec o #P.Rem and |P.Prec| = |#P.Prec| + 1;

Operation Reset(reassigns P: List_Position);
ensures: P.Prec = empty_string and P.Rem = #P.Pre o #P.Rem;

Operation Length_Of_Rem(restores P: List_Position): Integer;
ensures: Length_Of_Rem = |P.Rem|;

Operation Insert(clears New_Entry: Integer, reassigns P: List_Position);
ensures: P.Prec = #P.Prec and P.Rem = <#New_Entry> o #P.Rem;

Operation Remove(replaces Entry_Removed: Integer, updates P: List_Position);
requires: P.Rem /= empty_string;
ensures: P.Prec = #P.Prec and #P.Rem = <Entry_Removed> o P.Rem;

Operation Advance_To_End(updates P: List_Position);
ensures: P.Rem = empty_string;

Operation Swap_Remainders(updates P, Q: List_Position);
ensures: P.Prec = #P.Prec and Q.Prec = #Q.Prec and
         P.Rem = #Q.Rem and Q.Rem = #P.Rem;

Operation Swap_Previous_Entry_w(updates E: Entry; updates P: List_Position);
ensures: P.Rem = #P.Rem and (there exists A: Str(Entry), #P.Prec = A o <E> and
         P.Prec = A o <#E>);

Operation Length_Of_Prec(restores P: List_Position): Integer;
ensures: Length_Of_Prec = |P.Prec|;

Operation Clear List(clears P: List Position);
```

Listing 6.3: Specifications for Experiment 2 - Quiz 1

```
void mystery1() {
   Pre: |#self| > 0
   Post: there exists A: String of Object; B, C: Object such that
      #self = <B> * A * <C> and
      self = A
}

void Mystery2(int index) {
   Pre: index > 0 and index < |#self|
   Post: there exists A, B: String of Object; C: Object such that
      #self = A * <C> * B and
      |A| = index and
      self = A * B
}

void mystery3(int begin, int end) {
   Pre: being > 0 and begin < |#self| and end > begin and end <= |#self|
   Post: there exists A, B, C: String of Object such that
      #self = A * B * C and
      |A| = begin and
      |B| = end - begin and
      self = A * C
}

Object mystery4(int index, Object sub) {
   Pre: index > 0, index < |#self|
   Post: there exists A, B: String of Object; C: Object such that
      #self = A * <C> * B and
      |A| = index and
      self = A * <sub> * B and
      mystery4 = C
}

String mystery5(int begin, int end) {
   Pre: begin > 0 and begin < |#self| and end > begin and end <= |#self|
   Post: there exists A, B, C: String of Object such that
      #self = A * B * C and
      |A| = begin and
      |B| = end - begin and
      self = A * sub * C and
      mystery5 = B
}
```

Listing 6.4: Specifications for Experiment 2 - Quiz 2

```
void mystery1() {
   Pre: |#self| > 0
   Post: there exists A, B: String of Object such that
      #self = A * B and
      |A| = |#self| / 2 and
      self = B * A
}

void mystery2() {
   Pre: true
   Post: there exists int N = |#self|; A1, A2, ..., AN: Object such that
      #self = <A1> * <A2> * ... * <AN> and
      self = <AN> * <AN-1> * <AN-2> * ... * <A2> * <A1>
}

void mystery3(int index, int count) {
   Pre: index >= 0 and index < |#self| and index - count >= 0
   Post: there exists A, B, C: String of Object such that
      #self = A * B * C and
      |A| = begin and
      |B| = end - begin and
      self = A * C
}
```

Listing 6.5: Specifications for Experiment 2 - Quiz 3

```
void mystery1(String b) {
   Pre: true
   Post: self = #self * b
}

void mystery2(int index, String b) {
   Pre: index >= 0 and index < |#self|
   Post: there exists A, B: String of Object such that
      #self = A * B and
      |A| = index and
      self = A * b * B and
}

void mystery3(int index, String b) {
   Pre: index >= 0 and index < |#self| and index < |#b|
   Post: there exists A, B, X, Y: String of Object such that
      #self = A * B and
      #b = X * Y and
      |A| = index and
      |X| = index and
      self = A * Y and
      b = X * B
}

void mystery4(String b) {
   Pre: |#self| == |#b|
   Post: there exists int N = |#self|;
         A1, A2, ..., AN, B1, B2, ..., BN: Object such that
      #self = <A1> * <A2> * ... * <AN> and
      #b = <B1> * <B2> * ... * <BN> and
      self = <A1> * <B1> * <A2> * <B2> * ... * <AN> * <BN>
}

void mystery5(int a1, int a2, int b1, int b2, String b) {
   Pre: a1 >= 0 and a1 < |#self| and a2 > a1 and a2 <= |#self| and
        b1 >= 0 and b1 < |#b| and b2 > b1 and b2 <= |#b|
   Post: there exists A, B, C, X, Y, Z: String of Object such that
      #self = A * B * C and
      #b = X * Y * Z and
      |A| = a1 and
      |B| = a2 - a1 and
      |X| = b1 and
      |Y| = b2 - b1 and
      self = A * Y * C and
      b = X * B * Z
}
```

Listing 6.6: BASH Script Used to Generate Graphs

```
#!/bin/bash
gnuplot << EOF
set terminal png transparent nocrop \
    font "/usr/share/fonts/freefont/FreeSans.ttf" 10 size 800, 600
set output "$1.png"
set boxwidth 0.9 absolute
set style fill solid 1.00 border -1
set style histogram rows gap 1 title  offset character 0, 0, 0
set datafile missing -
set style data histograms
set xtics border in scale 1,0.5 nomirror rotate by -45 offset character 0, 0, 0
set title "$2"
set yrange [ 0 : 50 ] noreverse nowriteback
set ylabel "# of test cases"
plot "$1" using 2:xtic(1) ti col,  u 3 ti col
```

Bibliography

[1] L.L. Beck and A.W. Chizhik. An experimental study of cooperative learning in cs1. *SIGCSE Bull.*, 40(1):205–209, 2008.

[2] K.B. Bruce, R.L. Scot Drysdale, C. Kelemen, and A. Tucker. Why math? *Communications of the ACM*, 46(9):40–44, 2003.

[3] A.B. Campbell and R.P. Pargas. Laptops in the classroom. In *SIGCSE '03: Proceedings of the 34th SIGCSE technical symposium on Computer science education*, pages 98–102, New York, NY, USA, 2003. ACM.

[4] J.D. Chase and E.G. Okie. Combining cooperative learning and peer instruction in introductory computer science. *SIGCSE Bulletin*, 32(1):372–376, 2000.

[5] S.H. Edwards. Teaching software testing: automatic grading meets test-first coding. In *OOPSLA '03: Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 318–319, New York, NY, USA, 2003. ACM.

[6] Y.A. Feldman. Teaching quality object-oriented programming. *J. Educ. Resour. Comput.*, 5(1):1, 2005.

[7] gnuplot. Home page. http://www.gnuplot.info/, April 2009.

[8] D. Gries, B. Marion, P. Henderson, and D. Schwartz. How mathematical thinking enhances computer science problem solving. *SIGCSE Bulletin*, 33(1):390–391, 2001.

[9] M.T. Grinder. A preliminary empirical evaluation of the effectiveness of a finite state automaton animator. In *SIGCSE '03: Proceedings of the 34th SIGCSE technical symposium on Computer science education*, pages 157–161, New York, NY, USA, 2003. ACM.

[10] P.B. Henderson. Modern introductory computer science. In *The 18th SIGCSE Technical Symposium on Computer Science Education*, pages 183–190, New York, NY, USA, February 1987. ACM.

[11] P.B. Henderson. Mathematical reasoning in software engineering education. *Communications of the ACM*, 46(9):45–50, 2003.

[12] Intel Corporation. Statistical analysis of floating point flaw (white paper). http://support.intel.com/support/ processors/pentium/fdiv/ wp/, November 1994.

[13] J2SSH. Home page. http://j2ssh.sourceforge.net/, April 2009.

[14] JUnit. Home page. http://www.junit.org/, April 2009.

[15] R. Kramer. icontract - the java(tm) design by contract(tm) tool. In *TOOLS '98: Proceedings of the Technology of Object-Oriented Languages and Systems*, page 295, Washington, DC, USA, 1998. IEEE Computer Society.

[16] A.N. Kumar. Using online tutors for learning - what do students think? In *The 34th Annual ASEE/IEEE Frontiers in Education Conference*, pages (*cd–rom*), Piscataway, NJ, USA, October 2004. IEEE.

[17] A.N. Kumar. Results from the evaluation of the effectiveness of an online tutor on expression evaluation. In *The 36th SIGCSE Technical Symposium on Computer Science Education*, pages 216–220, New York, NY, USA, February 2005. ACM.

[18] M.D. LeBlanc and R Leibowitz. Discrete partnership: a case for a full year of discrete math. *SIGCSE Bull.*, 38(1):313–317, 2006.

[19] D. Leonard, J. Hallstrom, and M. Sitaraman. Injecting rapid feedback and collaborative reasoning in teaching specifications. In *SIGCSE '09: Proceedings of the 40th ACM technical symposium on Computer science education*, pages 524–528, New York, NY, USA, 2009. ACM.

[20] N. Leveson and C.S. Turner. An investigation of the Therac-25 accidents. *IEEE Computer*, 26(7):18–41, 1993.

[21] M.A.M. Vieira L.F.M. Vieira and N.J. Vieira. Language emulator, a helpful toolkit in the learning process of computer theory. In *SIGCSE '04: Proceedings of the 35th SIGCSE technical symposium on Computer science education*, pages 135–139, New York, NY, USA, 2004. ACM.

[22] J.L. Lions. Ariane 5: Flight 501 failure, report by the Inquiry Board, European Space Agency. http://esamultimedia.esa.int/docs/ esa-x-1819eng.pdf, April 1996.

[23] E. Mazur. *Peer Instruction: A User's Manual.* Series in Educational Innovation. Prentice Hall, Upper Saddle River, NJ, USA, 1997.

[24] J.W. McGuffee. The discrete mathematics enhancement project. *J. Comput. Small Coll.*, 17(5):162–166, 2002.

[25] K. McMaster, N. Anderson, and B Rague. Discrete math with programming: better together. *SIGCSE Bull.*, 39(1):100–104, 2007.

[26] Sun Microsystems. Java home page. http://www.java.com/en/, 2009.

[27] National Aeronautics and Space Administration. Mars climate orbiter mishap investigation board phase i report. ftp://ftp.hq.nasa.gov/pub/pao/ reports/1999/MCO_report.pdf, November 1999.

[28] R.P. Pargas and D.M. Shah. Things are clicking in computer science courses. In *The 37th SIGCSE Technical Symposium on Computer Science Education*, pages 474–478, New York, NY, USA, March 2006. ACM.

[29] S. Epp P.B. Henderson, W. Barker and W. Marion. Math educators, computer science educators: working together. *SIGCSE Bull.*, 35(1):236–237, 2003.

[30] T. Finley R. Cavalcante and S.H. Rodger. A visual and interactive automata theory course with jflap 4.0. In *SIGCSE '04: Proceedings of the 35th SIGCSE technical symposium on Computer science education*, pages 140–144, New York, NY, USA, 2004. ACM.

[31] A. Ralston. Do we need any mathematics in computer science curricula? *SIGCSE Bull.*, 37(2):6–9, 2005.

[32] A. Ralston and M. Shaw. Curriculum '78—is computer science really that unmathematical? *Commun. ACM*, 23(2):67–70, 1980.

[33] E. Roberts. Computer science final report. *Computinig Curricula 2001*, April 2002.

[34] S.H. Rodger. Jflap home page. http://www.jflap.org/, 2005.

[35] S.H. Rodger, B. Bressler, T. Finley, and S. Reading. Turning automata theory into a hands-on course. *SIGCSE Bull.*, 38(1):379–383, 2006.

[36] M. Sitaraman, J.O. Hallstrom, J. White, S. Drachova-Strang, H.K. Harton, D. Leonard, J. Krone, and R. Pak. Engaging students in specification and reasoning: hands-on experimentation and evaluation. *Technical ReportRSRG-08-05*, pages 1–6, 2008.

[37] M. Sitaraman, T.J. Long, B.W. Weide, E.J. Harner, and L. Wang. A formal approach to component-based software engineering: Education and evaluation. In *The 23rd International Conference on Software Engineering*, pages 601–609, Los Alamitos, CA, USA, May 2001. IEEE Computer Society.

[38] U.S. Food and Drug Administration. FDA seeks injunction against multidata systems intl. http://www.fda.gov/bbs/ topics/NEWS/2003/ NEW00903.html, May 2003.