Integrating Components, Contracts, and Reasoning in CS Curricula with RESOLVE: Experiences at Multiple Institutions

Wayne Heym, Paolo Sivilotti, Joseph E. Hollingsworth, Joan Krone, Murali Sitaraman, and Nigamanth Sridhar

Technical Report RSRG-16-01
School of Computing
100 McAdams
Clemson University
Clemson, SC 29634-0974 USA

May 2016

Integrating Components, Contracts, and Reasoning in CS Curricula with RESOLVE: Experiences at Multiple Institutions

Wayne Heym
Paolo Sivilotti
Ohio State University
Comp. Science and Engineering
Columbus, OH 43210
1-614-292-5813
{heym,paolo}@cse.ohio-state.edu

Joseph E. Hollingsworth
Indiana University Southeast
Computer Science
New Albany, IN 47150
1-812-941-2425
jholly@ius.edu

Joan Krone
Denison University
Math and Computer Science
Granville, OH 29634
1-864-656-3444
krone@denison.edu

Murali Sitaraman
Clemson University
School of Computing
Clemson, SC 29634
1-864-656-3444
murali@clemson.edu

Nigamanth Sridhar
Cleveland State University
Electrical Engineering and Computer Science
Cleveland OH 44115
1-216-687-5341
n.sridhar1@csuohio.edu

SUMMARY

Analytical reasoning is central to code correctness, and every computer science curriculum aims to teach students how to achieve this objective in one form or another. With the acceptance of object-based computing and component-based software engineering, the need for analytical reasoning that is based on formal contracts to establish correctness of software across module boundaries has become ever more obvious. Yet there are few institutions that have integrated modular, analytical reasoning principles in undergraduate curriculum. The reasons are many for this shortcoming, starting with the effort it takes overloaded faculty to integrate new ideas of any kind in their courses, to institutionalizing the ideas within the specific context, and constraints of a particular college. This paper presents our experiences over nearly two decades at three different institutions with the hope that they will serve as useful curriculum examples for like-minded educators at other institutions.

At Ohio State, the ideas are introduced in the introductory CS course sequence where Java is the programming language of choice. At IU Southeast, the principles are taught in a second year data structures and algorithms course, using C++ as the language of instruction. At Clemson, a required junior-level software engineering course presents the ideas in RESOLVE, a specification and implementation language combination that is supported by a web-integrated environment to facilitate reasoning. At Cleveland State, the ideas are used throughout the software lifecycle process in a required software engineering course. At Denison, some of the principles underlying verification are presented in a software engineering course. In all cases, analytical reasoning is one of the topics covered, but not the only one. At all institutions, the ideas have gone through extensive assessment before they were institutionalized, though details of these assessment efforts are not the topic of the present paper.

Categories and Subject Descriptors

D.2.4 [**Software/Program Verification**]: Including but not limited to the following – *programming by contract, class invariants, correctness proofs, formal methods.*

F.3.1 [**Specifying and Verifying and Reasoning about Programs**]: Including but not limited to – *assertions, invariants, pre- and post-conditions, specification techniques.*

F.4.1 [**Mathematical Logic**]: Including but not limited to – *proof theory, set theory, temporal logic*.

General Terms

Algorithms, Design, Reliability, Theory, Verification.

Keywords

Components, correctness proofs, programming by contract, specification, and verification.

1. OVERVIEW

The courses discussed in this paper share two common objectives that we believe are shared by many CS educators: Students must learn to reason about the correctness of code (within components) and students must learn to build and reason about component-

based software using contracts. The first one is a typical objective of earlier courses and the second one is a common objective of later courses. Typically, the reasoning process that is taught and the contracts that are used to describe components are presented informally. We believe the reason for absence of formality in code reasoning and component contracts is multi-fold:

- Commonly used programming languages in CS education, such as C++ and Java, do not have formal syntactic slots for expressing contracts or reasoning.
- Instructors do not have the time to develop formal contract specifications and integrate them seamlessly into their existing courses.
- Fellow faculty at institutions are still skeptical that undergraduate students can actually learn to understand and use formal contracts in reasoning, and that there are benefits for such reasoning in developing high quality software.

We have conducted numerous assessments to show that students can successfully learn formal reasoning principles. A summary of one such assessment spanning a 5-year period in two courses may be found in [1, 2]. More importantly, collectively we have successfully taught nearly 10,000 students at five institutions (four of which are public) over the last 15 years, and have witnessed firsthand the ability of typical undergraduates to learn formal reasoning.

The course structures of the four institutions are different and the places in the curriculum where we introduce formal reasoning are different. Taken together, our hope is that educators will see that formal reasoning ideas can be introduced in a variety of ways, in a variety of courses at different depths and using different example sequences. Materials and software used in these courses are either available publically at our web sites or available freely upon request.

What is common about the offerings of these institutions is the use of RESOLVE specification notation—conceived and refined over a period of 30 years especially with undergraduate computing education in mind [3, 4]. While there are several specification approaches in the literature, RESOLVE notation has been employed and assessed for use in undergraduate audiences for decades. The notations are meant to make a variety of core reasoning concepts taught in undergraduate computing easily accessible. At the same time, RESOLVE is sufficiently developed as a research vehicle that it is possible to specify and verify non-trivial software components automatically using a push-button verifying compiler that we have developed.

It is important to note that the ideas of formal contracts and reasoning are not the only content of the courses discussed in this paper. They are merely integrated with other topics normally taught in the courses. A part of the challenge is how to incorporate reasoning topics without necessarily displacing other topics that need to be covered. For example, in the software engineering course at Clemson, only five weeks are devoted to the topics covered in this paper. The rest of the course is concerned with other software lifecycle topics, such as requirements analysis.

The rest of the paper is organized into the following sections. Section 2 is a summary of the introductory course sequence at Ohio State, a novel aspect of which is a simultaneous introduction to specification, programming (Java), and reasoning. Section 3 is a summary of IU Southeast offering of data structures and algorithms in C++, a novelty of which is its online component. Section 4 is a summary of the Clemson two-course sequence with focus on the novel usage of a web-integrated reasoning environment for developing verified components in a software engineering course. Section 5 contains a summary of the Cleveland State and Denison software engineering courses. In the discussion of our experience at each institution, we give an overview of the courses and outline the reasoning topics covered in the course. We also summarize assignments, labs, or projects that are used to reinforce the ideas. In each case, we present illustrative examples to highlight key points. Section 6 contains a summary.

2. SUMMARY OF THE INTRO SEQUENCE AT OHIO STATE

Introductory computer science at Ohio State is a two-course (one academic year) sequence: Software I and Software II. About 1000 students, mostly CS or EE majors, take the sequence each year. Each course meets four times per week: twice in a lecture hall and twice in a computer lab. Pair programming is encouraged during the lab activities. Students complete frequent small homework assignments, in-lab programming activities, and larger multi-week projects [5].

This sequence is not an introduction to programming. Before taking Software I, students must demonstrate programming proficiency in an imperative language of their choice. AP Computer Science A serves as a possible pre-requisite, as do various one-semester courses offered at Ohio State such as “Intro to programming in Java” and “Intro to programming C++”.

Java is used as the programming language and Eclipse as the development environment for the entire sequence. Eclipse is configured with some plugins (e.g., Checkstyle and FindBugs) as well as specific preference settings (e.g., format-on-save) to encourage adherence to specific coding conventions. Other standard tools, such as JUnit, Javadoc, and the Eclipse debugger, are also used throughout the sequence.

The key characteristic of this sequence is the order of presentation: Students are first taught how to *use* components, then how to *implement* them. We call this approach “client-view first” [6]. This deliberate decomposition cleanly partitions the two courses: Software I is entirely client-view, while Software II is from the implementer’s perspective. In the first course, students learn to read component specifications, including mathematical models of state and behavioral contracts for methods. They write procedural code using these components and reason about the behavior of their code based on the component specifications. In the second course, students peel back the cover and look inside the components they used in Software I. They learn the relationship between a component’s internal concrete realization and its external abstract state. They implement method bodies that satisfy given contracts and they learn how to create layered, modular implementations.

The client-view first approach has many benefits. Firstly, students quickly get to use a rich collection of components to do engaging projects. For example, in the first few weeks of Software I, students use an XMLTree component to scrape interesting information from real-time RSS feeds. Secondly, students gain an appreciation for clean, well-documented interfaces. For example, tracing the step-by-step execution of a program using a Set component is possible only if students fully understand the client-side documentation of that component. Finally, students learn best practices for building modular systems. These best practices emerge as natural consequences of the strict decomposition of

client view and implementer's view, rather than simple ad hoc coding idioms.

We have found that formal methods, including specifications and mathematical models, are invaluable in supporting the client-view first structure of this course sequence. Furthermore, we have found that formal methods can be tightly integrated throughout the introductory sequence and can support learning of traditional topics such as iteration, recursion, argument passing, reference types, etc.

2.1 Abstraction: Mathematical Models for the Client

Information hiding is an important topic in any course that teaches modular programming. Equally important, but often omitted, is the other side of the coin: Abstraction. Since information hiding restricts what a client knows about the internals of a component, there is a complementary need for a client-visible explanation of a component's behavior. Such an explanation should not expose internal implementation details, so it is written in terms of a component's *abstract* state.

Abstraction is pervasive in computer science. It is used so often, in so many ways, and in so many different courses, that it hardly draws any particular attention. In our client-view first introduction to computer science, however, we highlight abstraction as a first-class citizen of software design. For each programming type, a corresponding mathematical model is introduced. In Software I, the description of a component is always done in terms of its mathematical model.

There are many opportunities for confusion between an implementation and its abstraction. There is overlap in vocabulary, for example "type" could refer to a program type (`int`) or a mathematical type (`integer`). There is also overlap in notation, for example "`=`" could be assignment in Java or a statement about equality in mathematics. In some cases, there is even overlap in semantics, when the mapping between one and the other is trivial, for example the mathematical model for an `UnboundedInteger` being an integer.

So it is important that the distinction between implementation and abstraction be made clearly and consistently. Java provides a convenient syntax for supporting this distinction: We use interfaces to define mathematical models and method contracts written in terms of those models, while classes contain implementations. By decomposing each component into an interface and a class, the discipline of writing an implementation-neutral cover story follows per force. The best practice of "coding to the interface" also follows as a natural consequence of a client-view first approach and is supported with interfaces. More details about this use of Java interfaces are in [6].

While interfaces and abstract classes are used in the standard Java libraries, their use is not consistent and they are never fully abstract. Therefore, we provide students with a library of components with clear abstractions defined in distinct interfaces [7]. Some of these components parallel parts of the Java collections framework (e.g., `Queue`, `Stack`, `List`), some are illustrative of basic concepts (e.g., `AMPMClock`), some are utilitarian (e.g., `XMLTree`, `SimpleReader`, `SimpleWriter`), and some are exemplars of best practices (e.g., `NaturalNumber`).

The building blocks for mathematical models for all of these components are simple mathematical types familiar to all freshmen, such as integers, reals, booleans, sets, and strings (i.e., lists of elements). Each type has a few operators, such as `^` for exponentiation and `|` `|` for the length of a string. The difference between math syntax and Java syntax reinforces for students the difference between a client-side cover story (written in the former) and the artifact itself (written in the latter).

Example: Strings for Queues and Stacks

Mathematical strings—finite sequences of entries—are used extensively to model container classes. For example, both Queues and Stacks are modeled by strings.

```
type QueueKernel is modeled by string of T
type StackKernel is modeled by string of T
```

This documentation is part of the Javadoc for each component's core interface which also contains formal contracts for each method. The contracts are written in terms of pre and post-conditions, with `#` in post-conditions to indicate the initial values of formal parameters and the name of the method to indicate the returned value. Parameter modes (such as "clears") are also used as shorthand for common conditions. Custom Javadoc tags are used to structure these contracts, integrating them with standard Java documentation and including them in generated documentation.

For example, below are the contracts for Queue's *append* and *dequeue* methods. These are read by the students and used to reason abstractly about their behavior.

```
/**
 * @updates this
 * @clears q
 * @ensures this = #this * #q
 */
 void append(Queue<T> q)

/**
 * @updates this
 * @requires this /= <>
 * @ensures #this = <dequeue> * this
 */
 T dequeue()
```

Example: Unbounded Natural Numbers

Other components require slightly refined models. For example, the NaturalNumber component stores an unbounded non-negative integer and involves an invariant on the abstract state:

```
type NaturalNumberKernel is modeled by NATURAL

NATURAL is integer
 exemplar n
 constraint n >= 0
```

In a method contract, e.g., *divideBy10* (below), the (abstract) value of *this* is a NATURAL:

```
/**
 * @updates this
 * @ensures #this = 10 * this + divideBy10 and
 *           0 <= divideBy10 < 10
 */
 int divideBy10()
```

2.2 Tracing Tables

Tracing tables employ strictly a client-view when "hand tracing" over any method call. Such tracing is in service of reasoning about the correctness of a method body under development. At this stage, each method to be implemented is done so in "layered" fashion. That is, it is not cooperating with other methods to represent a data type; instead, it provides an enhanced capability for an existing data type or is otherwise a "stand-alone" method. Such a

method's contract presents its client-view; the implementer reasons that the proposed method body always satisfies its contract under the assumption that all methods called satisfy their contracts. The tracing table is similar in spirit to the reasoning table discussed in [8] for a Java-based course at Clemson.

Implementing a method body recursively becomes, then, a problem of the same kind as the foregoing, with just a couple of additional constraints on the reasoning.

2.3 Implementing a Data Type

The task of arranging instance variables and bodies of constructors and methods to cooperate in representing a data type is significantly complex. For example, just the task of reasoning about the correctness of one method body involved in such a representation is significantly more complex than about that of one layered method body. Fortunately, techniques exist to organize and make explicit these additional demands in order to manage and simplify this complexity. Several of these techniques are involved in the art of designing the client-view of a data type; some more are involved in the art of designing a plan of implementation of such a data type. Our pedagogical approach supposes that students are well-served when they are asked to examine and work in the presence of many examples of good design long before they are asked to perform overall design tasks themselves. Therefore, Software II focuses on the practice of implementing a representation of a component and has students practice implementing the component's constructors and methods within the constraints of an already-complete design.

The primary artifact of such a completed design is, of course, the client-view of the component. In the currently-used programming language, this view is presented in Java interfaces, which, typically, have one or more generic type parameters and can be seen in [5] Specification reuse is achieved via multiple inheritance. A salient feature of this design arises from the recognition that layered implementations of methods are significantly easier than implementations involving representation. Hence, the latter are kept to a small number by finding a minimal kernel set of methods that capture the primary features of the component and make its values observable and controllable via these kernel methods. The contracts for these methods are gathered together in the interface that also has the key job of describing the client-view of the values of objects of this component type. All other useful methods for this component have their contracts presented together in an "enhanced" interface, which extends the kernel interface. The enhanced interface is implemented by an abstract class, which doesn't implement any of the kernel methods, but which does implement the enhanced methods in a layered fashion based, ultimately, on calls to the kernel methods, whose implementations are deferred to "kernel" classes that extend this "secondary" abstract class.

It is valuable for a given component to have multiple kernel implementations because these can differ in their time- and space-performance characteristics. As an example of engineering tradeoffs, they also vary in ease of implementation and/or understanding. A kernel class that is easy to implement can be built before one that has better performance characteristics and be used as a reference implementation in unit testing of the latter implementation. We have our students implement kernel classes in an in-class laboratory setting and as outside-of-class pair programming projects. We present our design decisions for each kernel class in three parts. A primary part is the choice of the types and names of the instance variables used in the representation. The next two parts, together, constitute the *internal contract* among the kernel methods. Often, not all (combinations of) possible values of these variables are sensibly used in the representation: a stated *representation invariant* (presented in a custom Javadoc tag @convention) captures this design decision. The intended interpretation of the instance variables' values cooperatively representing a client-view value is stated in the *abstraction relation* (presented in a custom Javadoc tag @correspondence). The abstraction relation need only be sensible for instance variables' values that satisfy the representation invariant. If each constructor and method body respects these three aspects of the design with respect to the corresponding contract in the interface, then the kernel class implementation will be correct. Hence, each operation's body can be developed independently of each other body. Additionally, we may wish our students to satisfy certain performance criteria, so we state those as well and may also require that certain algorithms be used.

An illustrative example comes from the third week of Software II. One in-class laboratory is dedicated to students implementing a kernel for the Set interface using one Queue value as the representation. The class they complete has the name Set2 in our catalog [5]. The one instance variable is named *elements*. The client-view for Queue (presented in QueueKernel) informs us that any value of *elements* is a *string of T*. It is to be used to represent a *set of T*, the client-view found in SetKernel. By this point in the two-course sequence our students are well versed at taking the client view and they reason about a component, e.g., Queue, using its math model rather than a concrete representation such as a linked-list or an array. The obvious interpretation of a string as a set is stated in the abstraction relation using a mathematical function from *string of T* to *set of T* named *entries*. This function "gathers together" every element present in the string to be a member of the set, disregarding any duplicates, of course. Because we designers recognized that the algorithms for methods *remove* and *removeAny* would be made more complicated if duplicates were permitted in the representation string, the representation invariant disallows duplicates (by insisting that the length of the string equals the size of the set obtained from that string via the mathematical function *entries*):

```
|$this.elements| = |entries($this.elements)|
```

We use the dollar sign ('$') as a prefix to "this" to indicate the representing class so that we can use "this" by itself to indicate the client-view value. Hence the abstraction relation here is expressed as:

```
this = entries($this.elements)
```

Consider, then, reasoning about the correctness of an implementation of the SetKernel method *removeAny*. Its contract has a precondition that the set is non-empty. It says that the return value should be some element of the incoming value of the set, and that the outgoing value of the set should contain all values that were in the incoming value, except the value that is returned. The implementer can reason by the abstraction relation that every element in the queue is a member of the represented set, so that dequeueing the front element of the queue will provide an element of the incoming set, and this value can then be returned. It is only by assuming, however, that the queue has no duplicates, that one can reason that dequeuing an element has caused the queue to represent a set that no longer has in it the element that came out of the queue. It is the representation invariant that justifies this assumption. The implementer, then, also has the obligation to satisfy this

same representation invariant upon return. Obeying this obligation, of course, is easily done, for example, by not gratuitously duplicating another element within the queue, even though doing so would still, through the abstraction function, represent the contractually expected outgoing value of the set. The additional involvement of the abstraction relation and the representation invariant is the reason why implementing a kernel method is more difficult than implementing a method without using direct access to a representation. For this in-class laboratory, we asked students to implement one private static helper method (a typical layered implementation) named "moveToFront" as preparatory homework. We provided the implementation of the one constructor specified in comments in the SetKernel interface, but asked students to implement the five SetKernel [5] methods during the laboratory session.

The topic of this subsection revisited in Section 4.2 with a detailed code example in the RESOLVE notation.

3. SUMMARY OF THE DATA STRUCTURE COUSE AT IU SOUTHEAST

IU Southeast's B.S. in computer science includes CS1 and CS2 as the first two courses for majors. Data structures and algorithms are introduced in CS2 as is common, however, a more in-depth coverage is permitted by our third course CSCI C343 which is called Data Structures. The introduction of data structures and algorithms in CS2 frees up time in C343 to integrate principles of engineering software that can lead to higher quality software development. One of the principles that we hit hard is design-by-contract [9] which for us includes contracts consisting of requires and ensures clauses and the assignment of obligations and benefits between the client (calling operation) and service supplier (called operation). This one-semester course typically includes software development labs that include client-view first programming, container component implementation by directly using C++'s built-in types, e.g., pointer types, and also by layering a new component on other existing components. Typical component abstractions implemented are: lists, sequences, maps, and sorting machines. For example, the students often implement Map three different ways using the technique of layering – layered on list using linear search, layered on an array of lists using hashing, and layered on a binary tree component using binary search. All of these C++ components [10] have contracts embedded as comments where the contracts are written in the RESOLVE specification language. Students use the components' contracts to reason abstractly about their behavior – this is taught early in the semester.

3.1 Teaching Students to Evaluate Software Design

One of the primary learning outcomes is: students will be able to evaluate an operation's design (implementation) for adhering to the design-by-contract assignment of obligations and benefits. Instructional materials have been developed to support this learning outcome and time is spent in class examining various operations that adhere and do not adhere to the design-by-contract engineering technique.

Below are two examples of operations that might be given to students for evaluation. During grading of lab assignments we collect examples of student submitted operations that violate design-by-contract and use these to drive discussion or test questions – many end up being more complex and nuanced than what follows.

The Queue template that we supply to the students is an unbounded queue and has specifications written in RESOLVE that are embedded as C++ comments. Specifications of Queue operations are similar to those discussed in the previous section, except that they are presented in a C++ context.

Now examine the operation *addToAll* (below) which has a requires clause of true, i.e., the client of *addToAll* is permitted to call with a queue of any length. In English, *addToAll* ensures that it will add the value found in parameter *x* to all integers found in the parameter *q*. *addToAll* is an example of an operation to be evaluated by the students in order to determine if it is a "good" client of Queue, i.e., that *addToAll* never violates (under any circumstances) any of the requires clauses of the operations it calls. Examining *addToAll's* code quickly reveals that it is a defective client in that it cannot guarantee that it will satisfy dequeue's requires clause under all circumstances. To computing instructors this is an obvious "rookie" mistake, but to the students it is not so obvious. If this example is used as a teaching tool, then once the class agrees that *addToAll* is a defective client, we can then turn our attention to correcting the defect, e.g., by using a 1-trip loop such as *for* or *while*. Training the student's eye to catch these types of defects is paramount for us.

```
typedef Queue<Integer> IntegerQueue;

void addToAll(IntegerQueue& q, Integer x)
// requires:true
// ensures:  for all k: integer (0 <= k < |q|
//           implies q[k,k+1) = #q[k,k+1) + x) and
//           x = #x
{
   IntegerQueue qTemp;

   do {
      Integer y;
      q.dequeue(y);
      y = y + x;
      qTemp.enqueue(y);
   } while (q.length() > 0);
   q.transferFrom(qTemp);
}  // addToAll
```

Another design exercise asks the students to evaluate an implementation of an operation with respect to taking advantage of the operation's requires clause. In C343, students work with both bounded and unbounded versions of our components. The *dequeue* operation found below comes from a bounded version where the BoundedQueue's internal representation is an array named *contents*. This is an example of an implementation that students are asked to evaluate for taking advantage of the requires clause. Under design-by-contract, the obligation for meeting the requires clause is on the client – in this case the caller of *dequeue* is required to call with a non-empty queue. The corresponding benefit is that the implementer of *dequeue* can assume the queue to be non-empty at the time of the call, therefore there is no need to have extra code to check for the queue being empty. The implementer of *dequeue* below failed to take advantage of this benefit as is seen by the all encompassing *if* statement that first verifies that currentLength > 0. What is illustrated below is often referred to as defensive programming, which in general we agree with but only in specific locations of a software system. Defensive programming should be used when the software system has no control of the incoming data, e.g., data incoming from across the

Internet. However, once that data has passed the system's "outer checking wall" and is now moving from one internal operation to another, a switch to design-by-contract must be made which brings with it the ability leverage design-by-contract's benefits.

```
template <class T, int maxLength>
void BoundedQueue1<T, maxLength>::dequeue(T& x)
{
   if(currentLength > 0) {
      x = contents[0];
      int z = (currentLength - 1)
      for (int k = 0; k < z; k++) {
         contents[k] = contents[k + 1];
      } // end for
      currentLength--;
   } // end if
}  // dequeuer
```

3.2 Online Course Offering Considerations

Learning can be improved with increased student engagement; this is true for face-to-face classes as well as with online courses. Effective teaching in the online environment [11] must foster student-to-faculty interaction and also engagement among students. We have used the examples above in discussion forums as a way to foster and increase online student engagement in our face-to-face, hybrid, and our totally online classes. Through our learning management system, we post one or more examples where the students are asked to evaluate an operation's implementation with respect to adhering to design-by-contract principles. We are careful to set the discussion forum's options so that a student can only see the posts of other students after he or she has first made a post. These posts can be as simple as determining if design-by-contract has been followed and why not if the answer is "no". Or, going further, the instructions might require the student to post a better-engineered version of the operation under evaluation.

4. SUMMARY OF THE SOFTWARE ENGINEERING SEQUENCE AT CLEMSON

At Clemson, principles of specification and reasoning are institutionalized in two required courses for CS majors, one a second year introduction to software development foundations course where the basic concepts are presented in the context of Java [refs] and another, a third year software engineering course. Students use the RESOLVE specification and implementation language in this course and develop verified components using a web-integrated environment [12, 13].

4.1 Software Development Foundations

At Clemson, a prerequisite to the software engineering course (discussed next) is a required course that introduces students to object-based development using Java, and instills basic principles of analytical reasoning using specifications. Contents and organization of this course as an exemplar ACM 2013 curriculum course are detailed in [8].

4.2 Software Engineering

The software engineering course at Clemson uses the RESOLVE specification and implementation language, and students develop code according to given contract specifications. Moreover, they use a web-integrated and automated RESOLVE verification system, available through a web IDE to verify that their code satisfies formal contracts [14].

One of the simplest exercises students do in this course is to develop code for the secondary Queue Rotate operation. In the ensures clause, Prt_Btwn is a mathematical string notation and it is the substring between the given positions. The verification system checks that the code (procedure) satisfies the contracts for the operations it calls (requires clauses of Dequeue and Enqueue) and that the code ensures the contract of the operation it implements (ensures clause of Rotate). The verification, of course, assumes the requires clause of Rotate, because it is the responsibility of Rotate's caller. Any errors are reported and students iterate the process until the code is verified.

```
Operation Rotate(updates Q: Queue);
        requires 1 <= |Q|;
        ensures Q = Prt_Btwn(1, |#Q|, #Q) o
                    Prt_Btwn(0, 1, #Q);

Procedure
        Var E: Integer;
        Dequeue(E, Q);
        Enqueue(E, Q);
end Rotate;
```

A more complex activity involves implementing one component using another. For example, implementation of a Set component was discussed in Section 2. Given below is part of an implementation of Queue_Template using List_Template in RESOLVE. The key idea in implementing data abstraction is the use of internal contracts using a conventions assertion (to capture the representation invariant) and a correspondence assertion (that captures the abstraction function or relation between the abstraction in the specification and the representation in the implementation). Students are given the interface specifications of components and the internal contracts that must be satisfied.

In the List_Template, a list is modeled mathematically as an ordered pair of strings of entries: a *preceding* string of entries, and a *remaining* string of entries. This conceptualization allows one to think of a list as having a cursor position in between the two strings. When an element is inserted into the list, it is added to the front of the *remaining* string. The remove operation has the opposite effect, so that an insertion followed by an immediate removal leaves the list unaffected. There is also an operation Advance that allows the cursor position to be moved so that insertions can take place at different places in the list. The Reset operation has the effect of moving the cursor position to the front and Advance_to_End has the effect of moving it to the end.

For the assignment given below, the representation invariant is that the preceding string is always empty. The correspondence is that the abstract queue is what is in the remaining string. This is why the code for Dequeue can simply call the list Remove operation. But the Enqueue code needs to move the cursor position to the end before insertion. To satisfy the representation invariant that code needs to Reset the cursor, so that the preceding string is empty.

```
Realization List_Based_Realiz for Queue_Template;
    -- declaration of an instance of list
    -- of entries, named LF
    Type Queue = Record
        Contents: LF.List;
        …
    end;
        convention Q.Contents.Prec = Empty_String;
        correspondence Conc.Q = Q.Contents.Rem;
```

```
    Procedure Enqueue(alters E: Entry;
                      updates Q: P_Queue);
        Advance_to_End(Q.Contents);
        Insert(E,Q.Contents);
        Reset(Q.Contents);
    end Enqueue;

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

    -- code for other Queue operations
    …
end List_Based_Realiz;
```

5. SUMMARY OF THE SOFTWARE ENGINEERING COURSE AT CLEVELAND STATE AND DENISON

At both Cleveland State and Denison universities, the RESOLVE notation is used in an undergraduate Software Engineering course, which is a 4 credit hour, overview course that covers all aspects of the software engineering life-cycle.

At Cleveland State, the course has specific emphasis on the correct construction of software systems and it is required for all Computer Science and Computer Engineering students. Students spend five weeks (20 hours of total instructional time) working with the RESOLVE notation. The first use of the notation is in the context of requirements specification. Students are introduced to the idea of specifying requirements in a formal, unambiguous manner. At the same time, students are also made conscious that rigorous formal specifications do not necessitate heavy, detailed, proof systems. Rather, the students are exposed to the modular specification method that is characteristic of RESOLVE—the proof obligations are embedded in the lower-level components, and *clients* of these lower-level components can leverage these proof guarantees in a light-weight manner. Later in the course, when working on design and implementation phases in the software engineering lifecycle, students rely on RESOLVE-style specifications to reason about correctness of implementations.

An example of an exercise that students work on this in course is the detailed specification of software for portion of an air-traffic controller (ATC) system. In particular, students describe how the ATC system can manage a number of aircraft in a given sector of airspace at a particular time, with the following constraints: (a) the airspace may include a number of aircraft, each with a unique identifier, (b) all aircraft must be separated by at least 300 meters in height, (c) a controller may create a new sector, (d) aircraft may enter, leave, and move in the sector, and (e) a controller may lookup a particular aircraft, and check if a particular region is occupied. The students go through an iterative design process, writing specifications for the abstract state of the sector, and each of the operations. Over the course of the iterations, the students use specifications for modeling the `Set`, `PartialMap`, and eventually, `BijectivePartialMap` components. The students then work through a reasoning exercise to construct a correctness argument for the Sector component.

The catalog description for the elective software engineering course at Denison emphasizes the importance of making connections between theory and practice, and it states that students will apply their theoretic background, together with current research ideas to solve real problems and that they will draw from their entire computer science curriculum, noting how theoretic results apply to real problems.

This four credit hour course covers all aspects of software engineering even as students seek to apply the principles to a real world problem. For example, in one offering, students completed a webpage that allowed local taxpayers to look at the tax reports of the school system and make predictions according to what level of taxation the village had.

Students learn to design software that is based on the principles that the RESOLVE system promotes – carefully specified, possible to reason about, and put together using components that are cohesive. They have access to the RESOLVE web interface provided by Clemson University and they see examples of software specified, implemented, and verified using that system. Before their introduction to RESOLVE, the students examine specifications of library components in C++ to critique those specifications and figure out why they are not adequate for making choices about using those library components. Then they examine RESOLVE specifications for stacks and queues and are introduced to the concept of formal, mathematical specifications. They write one or more enhancements of stacks or queues to see the importance of, and ease of programming components using other components that are formally specified and that have been verified.

Students at Denison have a strong mathematical foundation and so we look at a few proof rules for RESOLVE constructs in order that students get a glimpse of what it means to use automated verification. Then they use the RESOLVE web IDE to verify the enhancements they have written. Although students are not RESOLVE experts upon completing CS349, they have gained a sense of how software might be done in the future and they have not only some practical experience in doing their project, but they have developed and mastered many principles that they can carry over into their graduate programs or jobs.

6. RELATED WORK AND SUMMARY

Educators have explored a variety of ways to teach formal methods in undergraduate courses [15]. In principle, any formal specification language, such as those summarized in [16] could be used to describe contracts. Some specification languages are tailored to particular programming languages (e.g., JML) and some others (e.g., Z) are more general. Some other specification languages have been designed to be used in mechanical reasoning about code correctness [17]. However, few specification paradigms have been conceived and developed to teach component-based software engineering and reasoning for undergraduate computing students. Almost none has received multiple decades of undergraduate educational experimentation and tuning as RESOLVE.

This paper illustrates how RESOLVE can serve as an ideal vehicle in a range of computing courses, starting from introductory courses to advanced software engineering courses with or without using any popular programming language in conjunction. It also shows that the language is conducive to presenting software engineering and reasoning concepts in a few lectures to a few weeks at a variety of educational institutions with differing curricular constraints. The plethora of materials, exercises, activities, and assignments available make it easy for educators to adapt and customize.

ACKNOWLEDGMENTS

The members of our research groups contributed significantly to the ideas contained in this proposal. We also acknowledge NSF grants CCF-0811748, CCF-1161916, CCF-1162331, CNS-0745846, DUE-0942542, DUE-1022191, and DUE-1022941. Any opinions, findings, conclusions, or recommendations expressed here are those of the authors and do not necessarily reflect the views of the National Science Foundation.

REFERENCES

[1] Svetlana Drachova. 2013. Teaching and Assessment of Mathematical Principles for Software Correctness Using a Reasoning Concept Inventory. Ph.D. Dissertation. Clemson University.

[2] Svetlana V. Drachova, Jason O. Hallstrom, Joseph E. Hollingsworth, Joan Krone, Rich Pak, and Murali Sitaraman. 2015. Teaching Mathematical Reasoning Principles for Software Correctness and Its Assessment. Trans. Comput. Educ. 15, 3, Article 15 (August 2015), 22 pages. DOI=10.1145/2716316 http://doi.acm.org/10.1145/2716316

[3] Murali Sitaraman and Bruce W. Weide. 1994. Special Feature on RESOLVE, ACM SIGSOFT Software Engineering Notes, October 1994, 21-68.

[4] Murali Sitaraman, et al.. 2010. Building a Push-Button RESOLVE Verifier: Progress and Challenges. Formal Aspects of Computing 23, 2011, 607-626.

[5] "OSU CSE Components – API Specification." (31 January 2016) Retrieved from http://web.cse.ohio-state.edu/software/common/doc/

[6] Paolo A.G. Sivilotti and Matthew Lang. 2010. Interfaces first (and foremost) with Java. In Proceedings of the 41st ACM technical symposium on Computer science education (SIGCSE '10). ACM, New York, NY, USA, 515-519. DOI=http://dx.doi.org/10.1145/1734263.1734436

[7] Charles T. Cook, Svetlana Drachova, Jason O. Hallstrom, Joseph E. Hollingsworth, David P. Jacobs, Joan Krone, and Murali Sitaraman. 2012. A systematic approach to teaching abstraction and mathematical modeling. In Proceedings of the 17th ACM annual conference on Innovation and technology in computer science education (ITiCSE '12). ACM, New York, NY, USA, 357-362. DOI=http://dx.doi.org/10.1145/2325296.2325378

[8] Jason O. Hallstrom, Cathy Hochrine, Jacob Sorber, and Murali Sitaraman. 2014. An ACM 2013 exemplar course integrating fundamentals, languages, and software engineering. In Proceedings of the 45th ACM technical symposium on Computer science education (SIGCSE '14). ACM, New York, NY, USA, 211-216. DOI=http://dx.doi.org/10.1145/2538862.2538969

[9] Bertrand Meyer. Applying Design by Contract. Computer, 25(10):40–51, October 1992.

[10] "C343 Standard C++ Components." (31 January 2016) Retrieved from http://pages.iu.edu/~jholly/C343/Notes/ComponentSpecs/StandardComponents.html

[11] Marcia D. Dixson. 2010. Creating effective student engagement in online courses: What do students find engaging? Journal of the Scholarship of Teaching and Learning, 10(2):1-13, June 2010.

[12] Charles T. Cook, Heather Harton, Hampton Smith, and Murali Sitaraman. 2012. Specification Engineering and Modular Verification Using a Web-Integrated Verifying Compiler. Proc. 34th International Conference on Software Engineering, IEEE/ACM, 2012, 1379-1382.

[13] Charles T. Cook, Svetlana Drachova, Yu-Shan Sun, Murali Sitaraman, Jeff Carver, and Joseph E. Hollingsworth. 2013. Specification and Reasoning in SE Projects Using a Web-IDE. Proc. 26th Conference on Software Engineering Education and Training, IEEE, 2013.

[14] http://www.cs.clemson.edu/group/resolve (5 Sept 2014).

[15] Doug Baldwin, "Math-thinking-l – Mathematical reasoning in CS curricula", at: http://mail.geneseo.edu/mailman/listinfo/math-thinking-l/ (5 Sept 2014).

[16] John Hatcliff, Gary T. Leavens, K. Rustan M. Leino, Peter Müller, and Matthew Parkinson. 2012. Behavioral interface specification languages. *ACM Comput. Surv.* 44, 3, Article 16 (June 2012), 58 pages. DOI=http://dx.doi.org/10.1145/2187671.2187678

[17] Vladimir Klebanov, et al., The 1st Verified Software Competition: Experience Report. In FM 2011: Formal Methods, Proceedings 17th International Symposium on Formal Methods, Springer LNCS 6664, June 2011, 154-168.