A Reasoning Concept Inventory for Computer Science

Joan Krone, Joseph E. Hollingsworth, Murali Sitaraman, and Jason O. Hallstrom

Technical Report RSRG-09-01
School of Computing
100 McAdams
Clemson University
Clemson, SC 29634-0974 USA

September 2010

A Reasoning Concept Inventory for Computer Science

Joan Krone
Denison University
Mathematics and Computer Science
Granville, OH 43023
(740) 587-6484
krone@denison.edu

Joseph E. Hollingsworth
University of Indiana, SE
Computer Science
New Albany, IN 47150
(812) 941-2425
jholly@ius.edu

Murali Sitaraman
Jason O. Hallstrom
Clemson University
School of Computing
Clemson, SC 29634-0974
(864) 656-3444
{murali, jasonoh} @cs.clemson.edu

ABSTRACT

We identify a set of basic reasoning principles for computer science students that are essential to the development of high quality software. These principles can be integrated in various courses throughout the CS curriculum so that students understand not just how to write correct software, but to reason about why their software is correct. This paper summarizes evaluation results from our attempts to teach some of these principles. It also presents results from a survey of faculty interested in teaching reasoning concepts. The results show the importance of specific principles and their applicability to a range of courses.

Categories and Subject Descriptors

K.3.2. [**Computers and Education**] Computer and Information Science Education - *computer science education*; Also: D.3.1 [**Software Engineering**]: Specifying and Verifying and Reasoning about Programs – *assertions, invariants, mechanical verification, pre and post conditions, specification techniques.*

General Terms

Design, Documentation, Human Factors, Languages, Verification.

Keywords

Software engineering, components, mathematical thinking, tools.

1. INTRODUCTION

In 1992, Hestenes, Wells, and Swackhamer, along with a number of their peers, noted that typical physics students were entering their classes with preconceived, incorrect notions about fundamental physics concepts. Even when presented with material to correct misunderstood ideas, these students tended to revert to their original thinking in subsequent courses. To address this problem, the educators developed an inventory of physics concepts they believed to be a necessary part of every physics student's education [Hestenes 92].

Since then, many in the physics community have responded to the inventory, noting ways in which various courses and content modules address the need for introducing and deepening students' understanding of these basic concepts [Huffman 95].

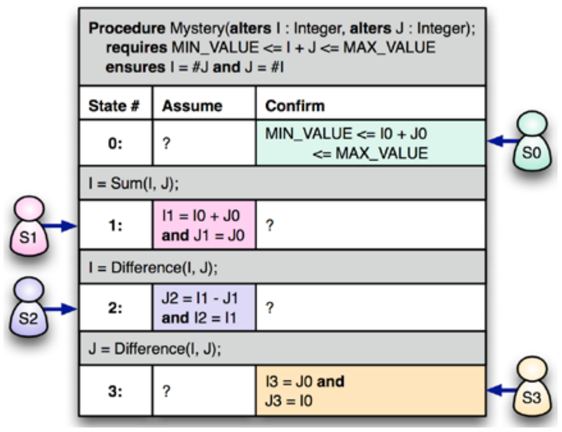
Using this effort in physics as our motivation, we have identified a set of basic reasoning principles essential to the development of high quality software. At the same time, the process we have used to develop the inventory is necessarily different, because unlike physics, where students have plenty of opportunities to develop misconceptions every day, few CS students have any conceptions about formal reasoning before they enter college [Tew 2010].

We have introduced the reasoning principles in some of our courses using a collaborative learning approach and supporting software tools. Results from attitudinal surveys described in [Sitaraman 09] show that students have both benefitted from and enjoyed learning these principles. Some of the tool-related benefits to support reasoning are summarized in [Leonard 09].

How important is formal reasoning for CS students? In the software verification competition held recently at the Verified Software Conference [VSTTE 2010], representatives for ten different verification systems offered solutions to competition problems. Each of the systems assumes that programmers can read and write formal specifications and assertions, such as class and loop invariants. So as verified software becomes more critical in the future, it becomes ever more important that our students know these principles.

2. BASIC REASONING PRINCIPLES

The goal of developing a *reasoning concept inventory* for computer science is to identify the principles that students must be taught to develop high quality software and to reason rigorously about the software they write and maintain. In the process, the hope is that they will also understand and appreciate intricate and important connections between discrete mathematics and computer science, the topic of discourse in [Baldwin 09]. Integrating reasoning as a connecting thread among courses also



Procedure Mystery(**alters** I : Integer, **alters** J : Integer);
requires MIN_VALUE <= I + J <= MAX_VALUE
ensures I = #J **and** J = #I

State #	Assume	Confirm
0:	?	MIN_VALUE <= I0 + J0 <= MAX_VALUE
I = Sum(I, J);		
1:	I1 = I0 + J0 and J1 = J0	?
I = Difference(I, J);		
2:	J2 = I1 - J1 and I2 = I1	?
J = Difference(I, J);		
3:	?	I3 = J0 and J3 = I0

Figure 1. Blackboard Exercise for Teaching Principle 5

helps students develop a cohesive view of the discipline as they first learn to develop software with introductory examples and objects, and then move on into data structures and algorithms, programming languages, software engineering, and other courses.

The important idea of teaching mathematical reasoning principles in undergraduate computing has had several pioneers [Baldwin 01, Bruce 03, Bucci 03, Devlin 03, Edwards 03, Gries 01, Henderson 01, Kumar 05, Long 98, Tomer 98]. A careful study of the work of these experts, as well as our own experiences over the past decades, has led us to the essential reasoning principles and the concept inventory presented in this section.

To reason about their programs using Boolean logic, students must be confident using and interpreting Boolean expressions, including expressions involving standard logical connectives (e.g, conjunction, disjunction, negation, implication, quantification). Similarly, they must be able to apply fundamental rules of logic, such as modus ponens and the law of the excluded middle. This principle can be broadened even further: Students need the ability to read and understand mathematical notations.

Mastering material in discrete mathematics is important, but not sufficient for specifying and reasoning about software. Students must understand the distinction between mathematical structures and their computer counterparts. For example, they must recognize that the integers provided by computer hardware are not the same as mathematical integers. We have ways to describe computer integers (with their associated bounds) so students can clearly distinguish between that formal description and the typical description of mathematical integers found in discrete math textbooks. Of course, it is important for students to see how the two integer sets are related.

With regard to specifications, a key lesson is that any given abstract data type (ADT) can be described mathematically so that it is possible to reason formally about the operations it provides. An ADT is specified using a mathematical model that describes its state space. Each operation can then be specified using appropriate pre- and post-condition assertions that capture the operation's impact on that state space. Reasoning about any program that uses the ADT can be based on those assertions without regard for how the ADT is implemented.

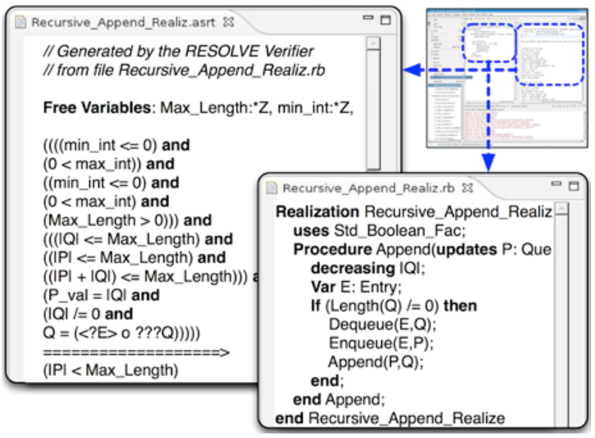
The ADT specification serves as an external *"client view"*, so that users of the ADT will know exactly how the component will behave without needing to examine its implementation. A developer of the ADT will use the specification as a guide for structuring the internal *"implementer's view"* hidden from the client. This clear-cut separation between client and implementer benefits students by enabling instructors to emphasize that there may be multiple implementations for any given specification, each with its own performance tradeoffs.

The benefits of this separation principle extend even further. Separating abstraction from implementation promotes the construction of large programs from component parts. The programmer of a large component can choose the parts needed for the implementation based only on the specifications of those parts. In fact, one can construct such a component even before any implementations of the smaller parts are ready. (Of course, those parts must eventually be implemented, either in parallel or at some later time for the software system to work.) Once a component has been proven correct, any client can incorporate it into a larger component without any need to re-verify its correctness. This principle of modular reasoning is critical for large systems.

An analogous case can be made regarding the benefits of internal component specifications. Given the representation *invariant* and abstraction *correspondence* for a given component implementation, team members can take turns writing code for the constituent operations independently. The operation bodies will only work together if everyone properly understands the internal contract assertions.

Implementations also need to be documented with invariants for loops, and terminating progress metrics for loops and recursive code. Given an annotated implementation for some specification, and specifications for the components used in that implementation, it becomes possible to generate verification conditions (mathematical clauses) that must be proven to establish correctness. Students can use their skills in applying logic to both develop and prove those assertions.

We have taught various combinations of these principles in our courses using collaborative blackboard exercises and reasoning tools. For a detailed discussion, we refer readers to [Leonard 09, Sitaraman 09]. Figure 1 corresponds to principle #5; it illustrates a blackboard exercise that we use to reinforce the connection between computing and mathematics. Students develop verification conditions for a given fragment of code using a *reasoning table*. They identify the *assumptions* and *obligations* for each state of the program. At the beginning of the code for an operation, the pre-condition may be assumed; at the end, the post-condition must be proven. If the code entails calling another operation, it is necessary to prove the called operation's pre-condition in the state before it is called. Then, in the next state (upon termination), its post–condition may be assumed. In the example shown, it is easy enough to view `Integer` additions and subtractions as calling `Integer` operations to add and subtract. Students can complete this table on the blackboard, either sequentially or in parallel. One exciting outcome is that students tend to note technical improvements to the assertions provided by their peers. The code is correct only if all obligations are proved using only assumptions in the preceding states.



```
// Generated by the RESOLVE Verifier
// from file Recursive_Append_Realiz.rb

Free Variables: Max_Length:*Z, min_int:*Z,

((((min_int <= 0) and
(0 < max_int)) and
((min_int <= 0) and
(0 < max_int) and
(Max_Length > 0))) and
(((|Q| <= Max_Length) and
((|P| <= Max_Length) and
((|P| + |Q|) <= Max_Length))) a
(P_val = |Q| and
(|Q| /= 0 and
Q = (<?E> o ???Q)))))
===================>
(|P| < Max_Length)
```

```
Realization Recursive_Append_Realiz
  uses Std_Boolean_Fac;
  Procedure Append(updates P: Que
    decreasing |Q|;
    Var E: Entry;
    If (Length(Q) /= 0) then
      Dequeue(E,Q);
      Enqueue(E,P);
      Append(P,Q);
    end;
  end Append;
end Recursive_Append_Realize
```

Figure 2. Tool-Assisted Reasoning for Teaching Principle 6

Figure 2 shows an example of tool-generated verification conditions for a recursive implementation of a queue reversal operation. Students can prove the correctness of these conditions (and therefore the code) by appealing to logic and their knowledge of appropriate discrete structures (strings, in this example).

This discussion leads us to the following principles for a reasoning concept inventory for CS students:

1. The use of Boolean logic to reason about programs is critical for establishing correctness.
2. In order to model software components, one needs familiarity with basic discrete math structures, such as sets, strings, integers and other number systems, relations, and functions.
3. Precise (mathematical) specifications for software components are critical in order to reason about component-based software and establish its correctness.
4. Modular reasoning must allow for individual components to be certified as correct without the need to re-verify those components when they are used in a larger program.
5. From mathematical specifications for a given component, it is possible to generate mathematical clauses (verification conditions) that are equivalent to the correctness of the component.
6. Students can use proof techniques from Boolean logic to prove the verification conditions (VC's) generated from the specifications.

A reasoning concept inventory that corresponds to these principles, along with key concept terms, is given in Figure 3.

3. INVENTORY SURVEY AND ANALYSIS

To study how well the terms and concepts match the expectations of other reasoning experts and educators, we conducted a survey of professors at various stages in their careers at a number of institutions, most of which are 4-year liberal arts colleges. There were 11 participants. The survey allowed for anonymity, if desired. It consisted of 21 questions about particular terms and more general concepts. For each question, respondents used a 6-point scale to express their agreement or disagreement with a given statement. The scale was labeled using these terms: *strongly agree*, *agree*, *moderately agree*, *moderately disagree*, *disagree*, and *strongly disagree.*

A few sample questions, responses, and their implications for teaching mathematical reasoning are discussed in this section.

Major Reasoning Topic	**Subtopic Summary**	**Concept Term Highlights**
Boolean Logic	Standard logic symbols, standard proof techniques	Connectives including implication; quantifiers; supposition-deduction
Discrete Math Structures	Sets, strings, numbers, relations, and other mathematical theories as needed	Example set notations: element, union, intersection, and subsets; string notations: concatenation and length
Precise Specifications	Mathematical descriptions of software; Interfaces for clients and implementers; Math models for data structures; Specifications of operations	Mathematical modeling; constraints; specification parameter modes; pre- and post-conditions; notations for input and output values
Modular Reasoning	Internal contracts and assertions in implementations; Construction of large programs from certified components; Understanding the role of specifications in reasoning; Each module needs to be proven correct only once	Representation invariant; abstraction correspondence; loop invariants; progress metrics; Reasoning with multiple specifications; tabular reasoning; goal-directed reasoning
Correctness Proofs	Construction of verification Conditions (VCs); VCs as mathematical assertions equivalent to program correctness; Application of proof techniques to VCs	States and abstract values of objects; Assumptions; Obligations; VCs; application of proof techniques on VCs

Figure 3. Summary of Reasoning Concept Inventory

Observation #1:

The statements ranged from the most basic, such as:

> **Q#1:** CS students need to understand how to use Boolean logic not only for understanding how computers work, but for establishing correctness of programs.

To the more advanced, such as:

> **Q#10:** CS students need to understand formal descriptions of internal code assertions, such as class representation invariants and loop invariants.

In the first case, 10 out of 11 respondents agreed or strongly agreed with the statement. In the latter case, there was still agreement from 10 out of 11 respondents. There was, however, a shift away from strong agreement; only 5 of them agreed or strongly agreed, whereas 6 agreed only moderately.

Clearly the time necessary to cover the latter topics is greater than the first. (We discuss loop invariants separately later in this section.) Teaching the types of representation invariants necessary to prove the correctness of data abstraction implementations is certainly non-trivial. Yet if we are to benefit fully from the reuse of objects, the topic is important. In a software engineering course, we were able to introduce students to formal assertions of representation invariants and abstraction functions, but we didn't teach them how to reason about the correctness of an implementation using those assertions. In other words, *"understanding"* of a representation invariant may have strong agreement, whereas *"application"* of the principle may have less agreement.

Observation #2:

Another set of questions concern the verification of code involving loops and recursion. Representative examples include:

> **Q#16:** CS students need to understand how to reason about termination formally.
>
> **Q#17:** CS students need to understand how to reason about correctness of code involving loops (using invariants) formally.
>
> **Q#18:** CS students need to understand how to reason about recursive code formally.

Two out of 11 respondents disagreed with all three questions, perhaps because the statements (by design) insisted on formality. The others agreed with the statements. The level of agreement was highest for the termination question. Also, there was slightly more agreement about teaching reasoning in relationship to loops, relative to reasoning about recursion. This is possibly because educators feel that students encounter more iterative code in practice than recursive code.

Observation #3:

The concept of modular reasoning and the use of interfaces in reasoning were explored in two questions:

> **Q#6:** CS students need to understand the distinct roles of clients (users) and implementers of components, and the use of interfaces.
>
> **Q#13:** CS students need to understand the concept of modular reasoning, which allows for individual components to be certified as correct without a need to re-verify when those components are placed in a larger program.

The level of agreement for both questions was uniformly high; about half the respondents strongly agreed with both statements.

Observation #4:

Convinced of the importance of formal reasoning (observation #2) and modular reasoning (observation #3), both dependent on component specifications, respondents were also in agreement on the importance of teaching specifications. There was a slight shift in the level of agreement when "formality" was introduced, but agreement was still high. Specifically, we consider responses to the following two questions, which differ only in their formality requirement.

> **Q#7:** CS students need to understand pre- and post-conditions of operations.
>
> **Q#8:** CS students need to understand formal descriptions of pre- and post-conditions of operations.

For both questions, all respondents were in agreement. For the first, 5 respondents agreed strongly with the statement, and 5 agreed. For the second, 3 respondents agreed strongly, 4 agreed, and 4 agreed moderately. The responses were similar when asked about loop and representation invariants.

Observation #5:

Some questions were designed to measure the perceived importance of directly connecting mathematics and computer science. One question focused on making this connection through specifications, whereas another focused on making this connection by applying logical reasoning principles in dispatching the verification conditions that arise in proofs of code correctness:

> **Q#2:** CS students need to understand the connections between software specifications and basic discrete math structures, such as sets, strings, integers and other number systems, relations, and functions.
>
> **Q#20:** CS students need to be able to apply their proof techniques from Boolean logic, such as induction, modus ponens, etc., to the challenge of proving the verification conditions (VCs) generated from the specifications.

All respondents agreed with the first statement; 9 of 11 agreed or strongly agreed. Two disagreed with the second statement. The agreement level among other respondents shifted towards moderate agreement. We attribute these responses to the difficulty of teaching the associated reasoning exercises without suitable reasoning tools. More precisely, in the absence of widely available tools accessible to undergraduate CS students, the task of generating and proving verification conditions can be difficult.

Observation #6:

The survey also asked educators to provide feedback on where the six principles might be taught in the computing curriculum. They were given eight course subjects and asked to identify the principles that should be included as part of each course. The results are summarized in Figure 4.

The most important observation is that respondents generally agreed that the reasoning principles are relevant to a broad range of computer science courses. Indeed, approximately half of the respondents indicated that principles 1 and 2 should be included in at least half of the courses. Further, with the exception of CS1 and AI, each course was associated with at least 4 of the reasoning principles. Data structures and algorithms and software

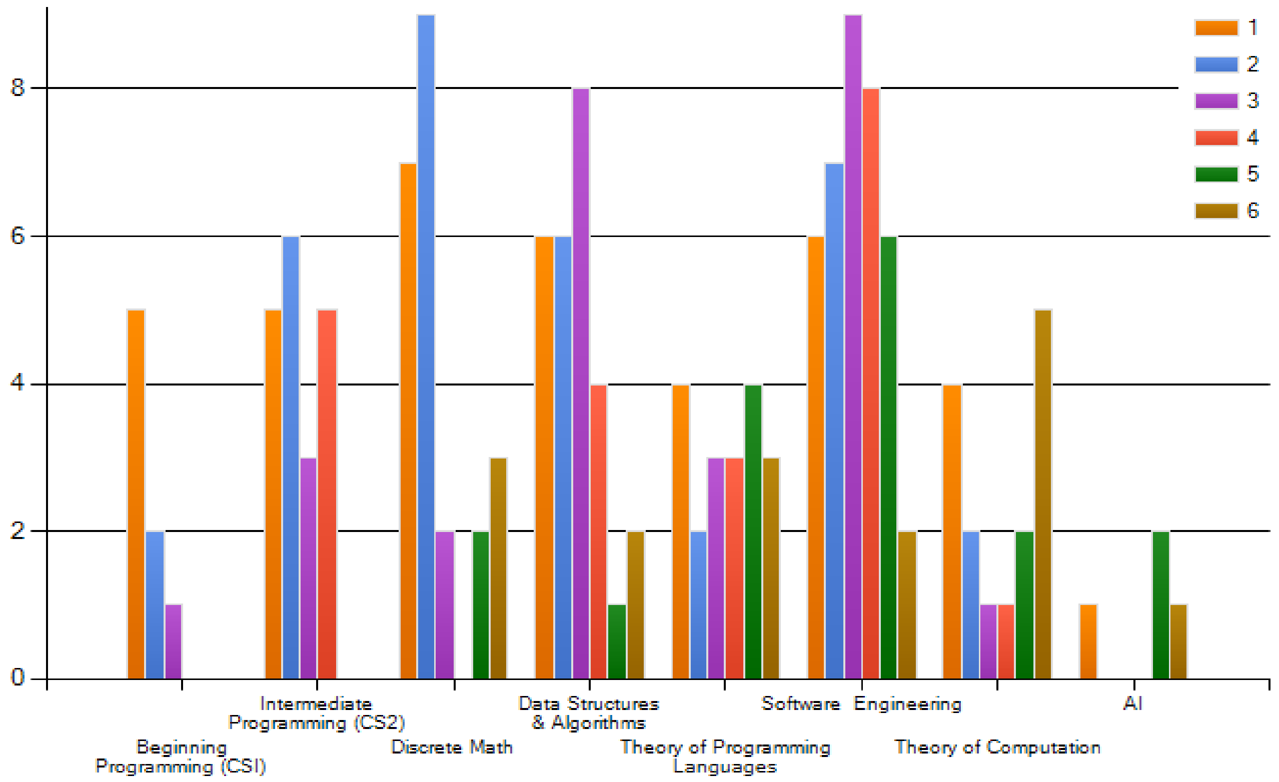


Figure 4. Which principles do you think should be a part of particular courses in the curriculum? (survey results)

engineering, in particular, were identified as excellent candidates for integrating a majority of the identified reasoning principles.

4. EVALUATION

To assess our efforts in integrating the concept inventory at our institutions, we have developed a set of learning outcomes. From these outcomes, we designed final exam questions for a sophomore-level, object-oriented programming course. These questions were used in the fall of 2008, and the spring of 2009. Here we describe one of the outcomes, a sample test question, and preliminary assessment results.

Learning Outcome for Reasoning Principle 6:

Student has the ability to carry out logical proofs of correctness in the context of code verification of simple code fragments.

Sample Question:

The following five verification conditions (VCs) correspond to a piece of code. Determine if the code is correct by checking to see if every VC can be proved. Mark on your test *all* VCs that *cannot* be proved. (Here we show only two of the ten VCs from the test.)

```
VC #1
(min_int ≤ 0)  and (0 < max_int) and
(Max_Length > 0)
and (|Q| ≤ Max_Length) and (|Q| = 2)
=======================>
|Q| <> 0
```

```
VC #2
(min_int <= 0) and (0 < max_int) and
(Max_Length > 0) and
(|Q| <= Max_Length) and (|Q| = 2) and
(Q = <I’> o Q’’) and (Q’’ = <J’> o Q’)
=======================>
|Q’| < Max_Length
```

Preliminary Results:

During the fall of 2008, 82% of students successfully answered two questions on the final exam similar to the question fragment shown above. For the spring of 2009, the success rate was 81%.

The other outcomes and corresponding student results from the fall of 2008, and the spring of 2009 are as follows: logical analysis of code – 65%, 11%; foundations of formal reasoning (tool aided) – 71%, 85%; understanding the role of specifications in modular reasoning (aided by collaborative learning) – 93%, 93%; and application of specifications to support reasoning – 77%, 55%. Poor wording compromised the question on logical analysis.

5. RELATED WORK AND CONCLUSIONS

The work presented in this paper builds on a decade of work that has confirmed the need for mathematics in the CS curriculum. There has been significant work on what should go into discrete math courses for CS students [Baldwin 04]. This discussion has continued at SIGCSE through *“Math Thinking”* BoF sessions and led to a panel discussion on a more comprehensive topic on mathematical reasoning for CS students at SIGCSE 2009 [Baldwin 09]

In parallel, the topic of teaching mathematical reasoning principles has been studied in a variety of forums at SIGCSE and elsewhere. A number of prior efforts in reasoning have focused on the introductory courses [Bucci 01, Henderson 03, McLouglin 96, Tomer 98]. Several other efforts have considered teaching reasoning in software engineering courses, in addition to introductory courses [Sitaraman 01, Henderson 03, Hilburn 96].

One of the issues that has repeatedly arisen following various mathematical reasoning discussions at SIGCSE in panel, paper, and BoF sessions is the need for a reasoning concept inventory. This paper is an effort to develop such an inventory based on careful analysis of discussions at these sessions, our own experiences in teaching at different types of institutions over a number of years, and inputs from others interested in reasoning.

The inventory in this paper is modest and focused on reasoning, unlike the more ambitious effort in [Goldman 08], where the authors identified important and difficult concepts in CS using a Delphi process. While identifying the topics in a more general CS context benefits from such a process, the inventory is simply a natural culmination of information from numerous experts, both past and present.

The paper brings together years of knowledge in the field of teaching reasoning in computer science. It presents a reasoning concept inventory at a level of detail that makes it possible to be adapted in different ways in different courses across the CS curriculum. It includes an affirmation of the ideas from educators. It contains results from evaluation of student learning outcomes for the principles. It can form the basis for teaching a fundamental CS topic.

6. ACKNOWLEDGMENTS

This research is funded in part by NSF grants DUE-1022941, DUE-0633506, DMS-0701187, DMS-0811748, CCF-0811748, and CNS-0745846.

7. REFERENCES

[1] Baldwin, D., Marion, B., Sitaraman, M., and Heerna, C., “Some developments in mathematical thinking for computer science education since computing curricula 2001” *ACM SIGCSE*, 2010, 392-393.

[2] Baldwin, D., Marion, B., Walker, H., Report of Special Group on Discrete Math for CS, *ACM SIGCSE*, 2004.

[3] Bucci, P., Long, T., and Weide, B., “Do we really teach abstraction?” *ACM SIGCSE*, Charlotte, North Carolina, United States, Pages: 26 – 30, 2001.

[4] Dony, I. and Le Charlier B., “A tool for helping teach a programming method,” *ACM SIGCSE ITiCSE*, Bologna, Italy, 2006, 212 – 216,.

[5] Goldman et al., “Identifying important and difficult concepts in introductory computing courses using a delphi process ,” *ACM SIGCSE Bulletin Volume 40* , Issue 1 (March 2008) .

[6] Gries, D., Marion, B., Henderson, P., Schwartz, D., “How mathematical thinking enhances computer science problem solving, “ *ACM SIGCSE*, 2001, Charlotte, North Carolina, pages 390-391.

[7] Henderson, P. B., “Mathematical Reasoning in Software Engineering Education,” *Communications of the ACM*, Vol. 46, No. 9, 2003, 45-50.

[8] Hestenes, D., Wells, M., and Swackhamer, G., Force Concept Inventory, *The Physics Teacher*, 30 (3), 141-151 (1992).

[9] Huffman, D., Heller, P., “What Does the Force Concept Inventory Actually Measure?” *The Physics Teacher*, 33 (3) 138-43 (1995).

[10] Hilburn, Thomas, “Inspections of Formal Specifications,” *ACM SIGCSE*, 1996, 150-154.

[11] Leonard, D.P., Hallstrom, J.O., and Sitaraman, M., “Injecting Rapid Feedback and Collaborative Reasoning in Teaching Specifications,” *ACM SIGCSE*, 2009

[12] Marion, B., Final oral report of the SIGCSE committee on the implementation of a discrete mathematics course, *ACM SIGCSE Bulletin Volume 38*, n.1, March 2006

[13] McLoughlin, H. and Hely, K., “Teaching formal programming to first year computer science students,” *ACM SIGCSE*, 1996, 155-159.

[14] Sitaraman, M., Long, T.. J., Weide, B. W., Harner, E. J., and Wang., L., “A Formal Approach to Component-Based Software Engineering and Its Evaluation,” *ICSE*, IEEE, 2001, 601-609.

[15] Sitaraman, M., Hallstrom, J.O., White, J., Drachova-Strang, S., Harton, H., Leonard, D., Krone, J., and Pak, R. Engaging students in specification and reasoning: “hands-on” experimentation and evaluation. *ACM SIGCSE ITiCSE*, 2009, 50-54

[16] Sitaraman, M., Adcock, B., Avigad, J., Bronish, D., Bucci, B., Frazier, D., Friedman, H.M., Harton, H., Heym, W., Kirschenbaum, J., Krone, J., Smith, H., and Weide, B.W., Building a Push-Button RESOLVE Verifier: Progress and Challenges, *Springer Formal Aspects of Computing*, 2010, to appear.

[17] Tew, http://computinged.wordpress.com/2010/04/01/how-computing-

[18] Tomer, T.S., Baldwin, D., and Fox, C. J., “Integration of Mathematical Topics in CS1 and CS2,” *ACM SIGCSE*, 1998, 364-365.

[19] VSTTE 2010 Competition Problems and Solutions, http://www.macs.hw.ac.uk/vstte10/Competition.html.