

Expanding Models of Cognition within Computing Education Research

Authors Blinded For
Review

ABSTRACT

CCS Concepts

•Social and professional topics → Computing education;

Keywords

1. ABSTRACT

This paper aims to expand our sense of what’s possible in modeling cognition within computing education research. We argue that research approaches that privilege canonical knowledge do so at the expense of other productive knowledge and ways of knowing that students have. We explore applicable cognitive theory by showing how distributed cognition and symbolic forms can be a powerful framework for analysis in CSEd. Finally, we conclude with an exploration of epistemological concerns, arguing that a fundamental concern for our research community should be paying attention to what counts as **knowledge** and **knowing** in computing learning environments.

2. INTRODUCTION

Science and math education started challenging misconception models of mind in 1993 [7, 42]. By 1996, learning scientists were boldly repudiating the assumptions of misconceptions: “Not all thoughts students express need to be understood as directly reflecting stable, stored knowledge structures. What the misconceptions perspective treats as a stored construct may alternatively be treated as an act of construction.” [17]. In the more than 20 years since those papers were published, researchers—particularly in science education—have pushed cognitive models beyond the static assumptions of classical misconception accounts of knowledge. But, a preponderance of research in computing education (CSEd) has modeled and continues to model cognition through misconceptions. Such research assumes students have entrenched, wrong ideas about computation. And,

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ICER 2016 Melbourne, Australia

© 2016 ACM. ISBN 123-4567-24-567/08/06...\$15.00

DOI: 10.475/123_4

while misconceptions research in computing education has advanced our understanding of teaching and learning, its narrow focus also begets problems that are growing too big to ignore:

1. When we use misconceptions to treat student ideas as right or wrong, we establish and perpetuate a deficit model of student understanding.
2. The more classical misconceptions-based research we publish as a community, the stronger that deficit narrative becomes in our discourse.

This paper aims squarely at those problems by expanding a sense of what’s possible in modeling cognition within computing education research. We argue research approaches that inflexibly privilege canonical knowledge do so at the expense of other productive knowledge and ways of knowing that students have. First, we reanalyze prior data to show how we might recast misconceptions as the potential seeds of productive knowledge. We then argue through example that expertise in programming can involve the fluid and active construction of conceptual metaphors to deal with idiosyncrasies within and across programming languages. These examples help broaden our sense of applicable cognitive theory by showing how distributed cognition and symbolic forms can be a powerful framework for analysis in CSEd. Finally, we conclude with an exploration of epistemological concerns, arguing that a fundamental concern for our research community should be paying attention to what counts as **knowledge** and **knowing** in computing learning environments.

2.1 Misconceptions research in computing education tends to ignore students’ productive knowledge

In the past three decades, educational research has had a marked focus on students’ misconceptions in programming. It’s a focus with a sensible origin. Students get things wrong in programming — often systematically so — and in ways that seem resistant to instruction. The cause of those errors is theorized to be something cognitive, whether it’s a “bug” [37, 35, 46], a “misconception” [1, 3, 4, 15, 22, 29], a “belief” [12], or a “student-constructed rule” [11, 13] in programming. Instruction, Clancy argues, should try to identify, address, and correct these misconceptions because they can represent barriers to learning [4].

How that line of thinking and research becomes problematic is two-fold. First, when taken in total the alleged brokenness of student knowledge begins eclipsing all else in studying the cognition of learning to program. In other

words, most cognitively-focused educational research in computer science treats students as having varied degrees of deficiency with respect to canonical knowledge. Below is an unordered, partial sampling of topics about which researchers have documented students' misconceptions. Note across the list the variation in both the grain sizes of students' misconceptions and the programming languages in which they manifest:

- Objects in object-oriented programming [23]
- Algorithms and data structures [5, 34]
- Programming statements in BASIC [1]
- Programming in Java [13]
- Programming in Pascal [12]
- Parameter-passing [11]
- Arrays in Java [29]
- Objects in Java [29]
- Algorithms and computational complexity [45]
- Boolean logic [22]
- The efficiency of algorithms [15]
- The Build-Heap algorithm [40]
- Hashtables [33]
- The correctness of programs [30]
- Polynomial and mapping reduction in the Theory and Computation of Complexity [14]

Clancy [4] provides a comprehensive overview of this line of research, though in the past decade it has only grown. Indeed, roughly half the articles above were published in the ten years since Clancy's overview.

Identifying and removing barriers to student learning seems like a good thing. So, it should follow that cataloging student misconceptions and developing remedies for them should also be a good thing. But, the logical implication isn't that clean. In some cases students display productive, useful knowledge that's either ignored or outright criticized by researchers. Aligning students toward canonical knowledge makes sense, but doing so at the expense of—or in direct contradiction to—useful ways of knowing seems undesirable at best. Next, we expand on two examples from misconceptions research in programming. Specifically, we show how and why we think a misconceptions focus in programming casts aside students' useful intuitions and understandings.

The first example comes from [29]. Part of that study involved giving students snippets of Java code and asking students to diagram (or pseudo-code) how the information would be stored in memory. Below we have reproduced the code for Problem 2:

```

1 Cheese[] cheeses = new Cheese[4];
2 Meat[] meats = new Meat[2];
3 Turkey turkey;
4 Ham ham;
5 RoastBeef roastBeef;
6 boolean lettuce = true;
7 boolean tomato = true;
8 SauceType sauceType = new SauceType();
9 int numMeat;
10 int numCheese;
```

In diagramming this information, a student in the study makes a mistake:

Student3 makes incorrect assumptions about connections between variables to the extent that the

student makes a mistake concerning the types of the variables. As a result, the student places Objects of different types in an array whose type matches none of them: “And so because there's two arrays, cheese and meats, uh, all those turkey and ham and roast beef are gonna be sorted into the meats array.” [29]

The researchers are correct in the sense that `turkey` and `ham` and `roastBeef` will *not* be sorted into the `meats` array. First, there is no code here that places `turkey` and `ham` and `roastBeef` in the array; there is only code that declares them as variables. Moreover, as written, an attempt to place `turkey` and `ham` and `roastBeef` into the array would fail. Because of type restrictions in Java, only objects of class `Meat` (or objects that inherit from class `Meat`) can go in the array. `turkey` and `ham` and `roastBeef` are, perhaps confusingly, references to object instances of classes `Turkey` and `Ham` and `RoastBeef`, so in the current snippet they cannot enter an array of type `Meat` because (1) they don't yet exist as objects and (2) even if they did exist, their types don't match the array's type. The authors call this misconception *semantics to semantics*, which occurs “when the student inappropriately assume[s] details about the relationship and operation of code samples, although such information was neither given nor implied” [29].

Again, the researchers are right that the student is failing to describe the code in a way consistent with canon. But, in their non-canonical thinking Student3 evidences potentially “productive” [18, 19] insights about design. Precisely *because* there is no code stating that `turkey` and `roastBeef` and `ham` are sorted into the array, the student is *inferring* that to be true. And, while that behavior is not what's happening, it *would* be sensible to design a program where specific instances of classes `Turkey` and `Ham` and `RoastBeef` could go into an array of type `Meat`. To do so, a designer could define `Turkey` and `Ham` and `RoastBeef` as inheriting from `Meat`.

Put another way, it's true Student3 has an idea about a relationship between entities that is not specified in the code. The study authors focus only on the *downside* of the idea: the student fails to display a proper understanding of how arrays work in Java. But, there is also an upside of this idea. Because Student3 is thinking about real-world propositions like turkey and ham being kinds of meat, they might be prepared to appreciate and discuss an object-oriented way to put turkey in a `Meat` array. But, that possibility is speculative conjecture. We can't know for certain whether Student3 could be tipped into a productive object-oriented design activity around the meats example because that question was not a focus of the research.

Our second example of research that criticizes students' non-canonical understandings comes from Bonar and Soloway's 1983 study of Pascal programmers [2], data from which is also analyzed and discussed in [35]. A student in [2] was asked to “Write a program which reads in ten integers and prints the average of those integers” (Bonar & Soloway, 1983, p. 12). In pseudo-code, she wrote:

```

Repeat
(1) Read a number (Num)
    (1a) Count := Count + 1
(2) Add the number to Sum
    (2a) Sum := Sum + Num
(3) until Count :=10
```

```
(4) Average := Sum div Num
(5) writeln ('average = ',Average)
```

The interviewer then asked whether (1a) and (2a) were “the same kinds of statements.” That interchange is reproduced here:

Interviewer: Steps 1a and 2a: are those the same kinds of statements?

Subject: How’s that, are they the same *kind*. Ahhh, ummm, not exactly, because with this [1a] you are adding - you initialize it at zero and you’re adding one to it [points to the right side of 1a] which is just a constant kind of thing.

Interviewer: Yes

Subject: [points to 2a] Sum, initialized, to, uh Sum to Sum plus Num, ah - thats [points to left side of 2a] storing two values in one, two variables [points to Sum and Num on the right side of 2a]. That’s [now points to 1a] a counter, that’s what keeps the whole loop under control. Whereas, this thing [points to 2a] was probably the most interesting thing... about Pascal when I hit it. That you could have the same, you sorta have the same thing here [points to 1a], it was interesting that you could have, you could save space by having the Sum re-storing information on the left with two different things there [points to right side of 2a], so I didn’t need to have two. No, they’re different to me.

Interviewer: So - in summary, how do you think of 1a?

Subject: I think of this [point to 1a] as just a constant, something that keeps the loop under control. And this [points to 2a] has something to do with something that you are gonna, that stores more kinds of information that you are going to take out of the loop with you. [2]

Pea and Kurland’s interpretation? “Here, again, we see the student believing that the programming language knows more about her intentions than it possibly can” [36].

As in Example 1, this student has an idea about relationships in code. Pea and Kurland [36] see the *downside* of her idea: believing PASCAL can understand shades of programmer intent when, in fact, it cannot. And again, that downside is real. It could cause trouble for this programmer later on if she expects PASCAL to interpret her intent and it cannot.

In defense of the student, the question — as asked — is vague. Are those statements the same *to whom* and *in what way*? Pea and Kurland treat the data as though she meant “the same to PASCAL.” Indeed, maybe she did, in which case his interpretation has traction. But, another interpretation is that she meant to herself or to someone else reading the code. Those statements might not be the same *to her* because she treats (1a) as having a function of controlling iteration while (2a)’s job is to combine two numbers into a new sum.

These two purposes, which for the sake of description we’ll call *keeping control* (1a) and *totaling up* (2a) are, in a sense, different. The PASCAL compiler (and runtime) does not

differentiate them, but humans can. And, humans may well *want* to differentiate them. diSessa (1986) describes exactly this kind of differentiation as a consequence of separating the structural understanding of a programming language from a functional understanding of a language. As an example, he discusses the structure/function difference with respect to variables:

The structural aspects of a variable in a computer language are given primarily by the rules for setting their values and for getting access to their values. These rules apply in all contexts. In contrast, a variable’s functions might vary. Sometimes they might be described as “a flag” or more generally, as “a communications device.” At other times a variable might function as “a counter,” “data,” or “input.” [6]

The student in [2] did not show evidence of understanding the structural similarities between (1a) and (2a) in her pseudo-code. And, those authors as well as others [36] justly insist that similarity is important for students to understand. From a conceptual standpoint, seeing the structural similarity constitutes a part of “knowing” PASCAL. But, even if knowing PASCAL were not the goal, seeing the similarity helps one to take the perspective of a computing agent that has no means for discerning programmer intent. Such perspective-taking may help students avoid mistakes that arise from over-assuming what a computer “understands.”

The student did, however, show evidence of understanding a *functional* [6] difference between (1a) and (2a), but Pea and Kurland do not remark on that kind of understanding at all. Both [36, 2] also gloss over another important difference: a programmer might not know in advance which numbers are being passed in to the sum statement. So, in advance the programmer can say nothing about how the value of **Sum** will change as the loop iterates. In contrast, the programmer knows exactly how the value of Count will change with each loop iteration. Again, we claim this oversight is part of a subtle but observable trend in programming misconceptions literature.

While, or perhaps because research has been so preoccupied warring with students’ problematic knowledge, it has sometimes failed to recover the productive knowledge (or resources for building it) students have. In Example 2, the student already has a grasp that syntactically similar statements could serve different conceptual purposes. And, that understanding could, in turn, shape how they program. The student might invoke the $\square = \square + 1$ syntactical template (Or what Sherin would call “symbol template” [41]) when the situation seems to demand *keeping control*, while invoking $\square = \square + \text{number}$ when *totaling up* is the goal. And, the idea that structurally similar symbol templates can serve different functional and conceptual purposes fits precisely in line with both diSessa’s distinction of structural/functional understanding of a programming language [6] and Sherin’s theory of symbolic forms [41].

An example drives home the connection between [6] and [41] as it applies to programming. The symbolic forms *parts-of-a-whole* and *base+change* have different conceptual schemata. Parts-of-a-whole refers to the contributions of component entities while *base+change* describes a kind of accumulation [41]. Specifically, the terms in *base+change* “contribute to a whole but play different roles. One is a base value; the other

is a change to that base.” But, the two distinct conceptual schemata share what we would argue is the same symbolic structure: $\square = \square + \square$ (parts of a whole) and $\square = \square \pm \Delta$ (base + change) [41].

The problem, for learning to program, comes in needing to fluidly interpret and write code in languages that may demand incommensurable, or at least distinct, conceptual schemata. As we show later in Table 4, three current programming languages make remarkably different use of the plus sign (+) as an operator. Crucially, some of the entailing ways to make sense of how + works in those languages don’t exist in Sherin’s (2001) catalog of conceptual schemata. In other words, we would argue there are conceptual ways a *programmer* may need to think about interpreting or writing a $\square = \square + \square$ symbol template that even Sherin [41] doesn’t enumerate. One of the most obvious, for example, is the conceptual movement from seeing $\square = \square + \square$ as a statement of equality (typical in a math class) to seeing it as the assignment of a sum to a variable (typical in programming). To follow that implication, the canonical body of knowledge about which programmers must reason is itself fractured, because different languages design their operations around different symbolic and conceptual metaphors.

To return to misconceptions, what seems to drive research on students’ misconceptions in computing education is largely a need to get students to program computers and reason about computation in ways that are canonically correct. And, that’s a worthwhile goal. But, as we’ve argued, we can already identify cases where a narrow misconceptions focus is silent about (at best) or dismissive of (at worst) students’ useful intuitions. We identified the further problems when a unilateral emphasis on one language’s canon opposes the vocabulary of symbolic forms (and diverse conceptual schemata) expert programmers ultimately need. *Taken in total, that silence, dismissiveness, and narrow view of refining knowledge perpetuates a deficit-focused discourse about student’s knowledge in computing.* A knowledge-deficiency perspective also fails to address what aspects of students reasoning might get broken by attempts to “fix” such “misconceptions”.

It’s important to note that misconceptions research in computing didn’t always treat students’ non-canonical knowledge as a problem. We begin the next section by backtracking to some of the earliest work on students’ cognitive “bugs”. There, we find researchers talking more explicitly about what’s useful in students’ non-canonical knowledge—a stance largely absent in modern computing misconceptions research.

3. CORE ARGUMENTS

3.1 Not all cognitive programming bugs imply a problem with the student

What’s curious about Pea and Kurland’s [35] comments on Bonar and Soloway [2] is that Pea’s conclusions [35] about Bonar and Soloway’s data differed greatly from Bonar and Soloway’s conclusions. Pea emphasized that when the student thought two semantically-equivalent assignment statements in PASCAL were different, it was a problem of egocentrism: “students assume that there is *more* of their meaning for what they want to accomplish in the program than is actually present in the code they have written” [35]. In other words, Pea treated the data as a fairly clear example of a

class of cognitive bugs. Specifically, he saw a bug class in which students simply assumed the interpreter or runtime could infer programmer’s shades of intent.

Bonar and Soloway (1983), by contrast, were less quick to make inferences either about the nature of the bug or the intervention it entailed. Rather than jump to definitive conclusions, they were circumspect:

It is not clear exactly how to react to the bugs we have uncovered in novice understanding of programming. In some cases it may be appropriate to design new languages or constructs. Often, better instruction would take care of the problem. The intent of our studies is to better understand the source of the mismatches and misconceptions that cause novice bugs. Only once a bug is uncovered and understood are we ready to create a remedy for that bug. [2]

That Bonar and Soloway would even consider developing new constructs or languages is a noteworthy distinction. Rather than assume wholesale that students with “bugs” had wrong knowledge, the authors instead suppose cognitive bugs have plausible origins worth designing around. Moreover, they treat students’ divergence from canon as *an opportunity for research to learn from students.* Precisely because students saw functional differences in semantically-equivalent PASCAL statements, Bonar and Soloway [2] reflect that perhaps programming languages should be more expressive:

We find it quite interesting that novices seem to understand the role or strategy of statements more clearly than the standard semantics. Such roles discussed here include “counter variable,” “running total variable,” “running total loop,” and “first, then rest loop”. (See Soloway et al [1982b] for a detailed discussion of novice looping strategies.) Much work in programming languages is concerned with allowing a programmer to more accurately express his or her intentions in the program. Perhaps we can learn something from novices here - our programming systems should support recording the roles the programmer intends for various statements and variables. [2]

Again, what’s noteworthy is that rather than treating students’ non-canonical views as a burden for instruction, Bonar and Soloway instead see them as an opportunity for programming language designers to make languages better. Indeed, an empirical approach to language design has recently been articulated in Stefik and Siebert’s work regarding language syntax’s effect on novices. [43]

That insight—that instruction and design can meet novices where they are—carries through to Bonar and Soloway’s final remarks about studying and analyzing novice programming knowledge:

The experience and understanding of a novice are available for analysis. In particular, our results suggest that the knowledge people bring from natural language has a key effect on their early programming efforts. Our work suggests that we need serious study of the knowledge novices

bring to a computing system. For most computerized tasks there is some model that a novice will use in his or her first attempts. We need to understand when it is appropriate to appeal to this model, and, when necessary, how to move a novice to some more appropriate model. (Bonar & Soloway, 1983, p. 13)

We assume Bonar and Soloway structured the final line of that quote deliberately. If so, their phrasing has three consequences:

1. Appealing to novice’s existing models gets precedence. That is, understanding how to leverage novices’ existing knowledge comes first in research.
2. Changing the models novices have comes next, and explicitly “when necessary.”
3. They speak of “how to move a novice to some more appropriate model,” which does not necessarily entail rejecting a student’s existing model, treating it as a misconception to be corrected, or otherwise ignoring what productive elements are present in a given novice model.

We believe that taken together, these points convey a sense of how Bonar and Soloway [2] viewed learning and instruction in computing. Instruction explicitly includes appeals to prior models and knowledge students might already have. Learning, meanwhile, involves the movement *when necessary* to more appropriate models of computation. As we explain in the next section, such a view of understanding and refining student ideas—in contrast to diagnosing the deficiencies in and replacement of student ideas—exactly aligns with a particular branch of constructivism, where cognition is viewed as the complex activation of manifold resources for thinking and knowing.

3.2 Examples motivate the need for contextual-sensitivity in modeling programming cognition

Let’s begin with two motivating examples. Our aim with these examples is to show how a practicing programmer might employ specific, distinct conceptual models to reason locally about a piece of code.

3.2.1 Example 1: Thinking With Kinematics Can Transform a Debugging Problem

First, consider variants of the PASCAL statements from Bonar and Soloway (1983), where an assignment statement worked to increment a value and store the result back to that value. Below are JavaScript statements that, as programmers, we have actually written in code as part of our careers. Each statement adheres to the structural similarity in Bonar and Soloway’s PASCAL example [2] discussed earlier. But, the comment above each statement reflects the *functional understandings* [6] we made use of to write, interpret, and debug the code we were working on.

```
1 // Increment a counter
2 x = x + 1
3
4 // Advance a simulation forward
5 t = t + dt
6
```

```
7 // Move an object under uniform velocity
8 x_position = x_position + displacement
9
10 // Accelerate an object along the x-dimension
11 x_position = x_position + displacement(t)
12
13 // Jitter a plot point using a stochastic function
14 y_position = y_position + random_noise()
```

We stress the comments reflect *how one might think about* each programming statement. By no means are we making normative claims about how one *ought* to think about it, or whether the statement actually does what its variable names might suggest it does. Rather, “How we might think about it” reflects the kind of local meaning or interpretation we might attach to such a statement when we work with it, given our understanding of its role and context.

Consider specifically what would happen if we found ourselves needing to debug lines 2 and 11. Line 2 could very well be an incrementer in some kind of iterative code. If we had to debug loop code that employs line 2, what we might do is try inserting intermediate print statements to see the value of *x*. We might also call on our knowledge that it’s being incremented by 1 to check whether the difference between any two successive printouts is 1; if it isn’t, we immediately know something’s wrong. But to diagnose a potential problem in line 11, we might take an altogether different strategy.

In line 11’s case, it could very well be that line 11’s function is animating images on a screen. Suppose that’s the case. If there’s a problem with that code, one of the easiest ways we might notice is that the *acceleration* seems off in the animation. Consequently, in debugging we might call upon knowledge we have from kinematics to conceptually model what our code is doing. We might try inserting code that draws a dot at the animated object’s on-screen position with each iteration, leaving a trail of dots we can visually inspect. We could then look at the path of the object’s trajectory and the spacing patterns between successive dots as a first-pass test of whether the code achieves the motion we want. (If, for example, our code is supposed to achieve a cubic easing animation, but the dots in the trail are all evenly spaced, basic kinematics tells us our object isn’t accelerating at all. And, if there’s no acceleration, we can pursue a strong hunch that the `displacement(t)` function is returning the same constant value with each iteration.)

To be clear, it’s not just that graphics might improve our efficiency in debugging a statement like line 11. Rather, applying an interpretive frame that treats an assignment statement as a kinematic position update lets us *use conceptual knowledge from physics* to diagnose and fix problems in code. For a comparative perspective, consider Hutchins’s argument that for aircraft pilots, ringing an aircraft speedometer with physical markers transforms the cognition involved in landing a plane:

Without a speed bug, on final approach the PF [Pilot Flying] must remember the approach speed, read the airspeed indicator scale to find the remembered value of the approach speed on the airspeed indicator scale, and compare the position of the ASI [Airspeed Indicator] needle on the scale with the position of the approach speed on the scale. With the salmon bug set, the pilot no

longer needs to read the airspeed indicator scale. He or she simply looks to see whether or not the indicator needle is lined up with the salmon bug. Thus, a memory and scale reading task is transformed into a judgment of spatial adjacency. [25]

Our physics debugging example and Hutchins’s speed bug example [25] share two core traits:

1. Cognition gets *distributed* [26, 24] across time and space.
2. That distribution doesn’t simply enhance, but rather *transforms* the nature of the activity, allowing cognitive agents to bring to bear resources that the prior or original context didn’t afford.

In Hutchins’s [25] example, pilots distribute cognition across time (1) by making approach speed calculations mid-flight, well before they’re needed. Pilots then distribute those calculations across space (2) by ringing the speedometer with bugs to represent distinct speeds. And, it is this act of distribution that lets pilots use spatial-adjacency perception (visually making sure the needle is on the bugs) during approach instead of having to calculate, compute, and compare speed numbers during approach.

In our kinematics debugging example, cognition gets distributed across time and space (1) by creating a visual trace of the animated object’s position (a persistent trail of dots). And, it is this act of dot-tracing that lets a programmer invoke spatial-adjacency perception and conceptual kinematics knowledge (how positions uniformly sampled in time can reflect constant velocity vs. acceleration) that wouldn’t seem applicable if the reassignment statement were stripped of its context.

the cognitive act of debugging in this instance. There, For comparison, consider

If we view the assignment statement as proposing something about the motion of an object—as, for example, when sense-making in science and engineering students can view equations as “saying something” about how a system behaves [blinded] Given these motivating examples, it seems sensible to think there’s utility in a programmer having different conceptual metaphors available to think about and work with code.

3.2.2 Example 2: The Plus Sign Means Lots of Things

Our second example concerns statements that use the same symbol template for fundamentally different kinds of computation. In our JavaScript example statements, all code came from the same language. But, another phenomenon comes into play *across* different languages when they use the same symbology (written patterns) to stand for conceptually different operations. The statements below are examples of what look like semantically-equivalent operations, but in fact are not.

```
// C - Increments the value of i by 1
i = i + 1

// JavaScript - Appends "ly" to word.toString()
word = word + "ly"

# R/ggplot2 - Composes a layer of points onto a plot
p = p + geom_point()
```

The catch here is that the plus operator takes on different roles in different languages because of how those languages define its use. Statement 1 increments a number in C; Statement 2 appends the letters “ly” to a string; Statement 3 composes a layer of points onto a statistical graphics plot. These different kinds of operations become even more apparent and consequential when, for example, such statements are repeated in the same language within the same file, as we’ll see in Example 3.

3.2.3 Example 3: The Same Symbol Template Within The Same Language Can Still Mean Different Things

In the R/ggplot2 code below, the author uses multiple reassignment statements to compose a statistical graphics plot.

```
1 p = InitializeGgplot_1w()
2 p = p + GrandMeanLine(owp)
3 p = p + GrandMeanPoint(owp)
4 p = p + ScaleX_1w(owp)
5 p = p + ScaleY_1w(owp)
```

Despite the syntactic similarity, The layered creation of a plot invites a very different kind of conceptual interpretation than, say, repeatedly accumulating numbers into a running sum. One obvious reason for thinking about this code with a different interpretive frame than incrementation is that numeric addition is commutative; composing a plot is not necessarily commutative. So, despite the syntactic similarity, statements that compose a plot using reassignment (as above) do not obey the same rules as reassignments for a running total. But, it turns out the statements above don’t obey the same rules of string concatenation either. Within this code block, statements that look alike perform operations of a different nature. While some expressions (lines 2 and 3) compose visual layers onto a plot, others (lines 4 and 5) modify features of the plot, like the x and y scales.

Given these motivating examples, it seems sensible to think there’s utility in a programmer having different conceptual metaphors available to think about and work with code.

We believe that working from conceptual metaphors—in our example, attacking a debugging problem by exploiting animation and drawing from physics knowledge—achieves the same kind of transformation Hutchins describes.

Example 2 shows that across languages, programmers might have to deploy different conceptual metaphors to reason about statements in a locally-consistent way. Knowing that plots in ggplot2 can be composed layer-by-layer with reassignment is crucial if you’re trying to write or understand code that creates statistical graphics. But, we would argue that thinking about $\square = \square + \square$ as “compose new layer onto plot” can and does appeal to different kinds of knowledge when compared to thinking about $\square = \square + \square$ as “include this addend in the sum,” which itself can and does appeal to different kinds of concepts when compared to thinking about $\square = \square + \square$ as “increment the counter.”

Stepping back, we can build the following argument

1. Programming can be helped by applying conceptual models to code, particularly when relevant domain-knowledge structures can advantageously transform a problem (example 1)

2. But, conceptual models don't work all the time for all statements. Because languages are designed differently, the same syntax can actually correspond to very different operations in code (example 2). And, that's true both within and across languages (example 3).
3. Consequently, it makes less sense to treat conceptual models as right or wrong, and more sense to treat them as differentially advantageous for thinking about what a piece of code does. (Thinking a "+" implies numerical addition isn't *globally wrong* in JavaScript, but it won't explain why $1 + "1"$ yields "11" as a result.)
4. It seems plausible that successful programmers, when reasoning about or writing code, are able to dynamically access or deploy conceptual models that are advantageous given the context (language, syntax, surrounding code). Certainly prior research demonstrates novices and experts both have resources for creating, evaluating, and adapting their own conceptual metaphors to suit the context of the problem [21, 27, 28]
5. To model how programmers think with conceptual models, a suitable framework should be able to account for the dynamic, context-sensitive deployment of conceptual knowledge.
6. To model how programmers develop expertise, a suitable framework should be able to describe higher-order phenomena. Such phenomena include explaining how programmers come to have conceptual models or generate new ones, why they decide to deploy them, and how programmers consider which conceptual model (i.e., which way to think about code) is appropriate.

Taken together, these assertions propose criteria for how we might strive to model cognition in programming. Our modeling frameworks should be context-dependent, dynamic, and capable of explaining where conceptual models come from. They should also be able to account for phenomena that are not themselves conceptual, including what directs the use of certain kinds of conceptual knowledge. In the learning sciences, such models already exist and have proven useful and productive for thinking about thinking.

3.3 Manifold models of cognition can explain context-dependence and the growth of expertise

In 1993, a pair of articles in the learning sciences staked a strong claim for viewing knowledge as a network of pieces, isolated enough to be locally triggered but trainable enough to fire in larger concerted patterns [7, 42]. Informed in part by agent-based accounts of cognition [32] and complex systems models [8], the central tenets of an "in-pieces" approach hold that knowledge is emergent from interacting primitives, rather than unitary and monolithic. An example from Smith, diSessa, and Roschelle helps illustrate the point.

The authors show that we might think of a rubber band as a different conceptual entity depending on context. In several different situations—wrapped around a newspaper, suspending a bob weight, pulled taut as a string, spun to store energy in a toy plane propeller—we intuitively think about the rubber band's physical behavior differently: one as a negligible part of the newspaper's point mass, two different kinds of harmonic pendulums, and finally a torsional spring. Those differences in intuitive thinking reflect the contextual dependency of what we know about the physical world:

In each of the rubberband examples, various pieces of intuitive physical knowledge describe the mechanism at work: the rubber band binds the newspaper, grips the jar lid, and acts a source of springiness for the bobbing object. Although a mapping cannot be made from the rubberband to scientific entities, it is quite easy to map these qualitatively distinct physical processes to scientific entities and laws. For example, instances of binding almost always map to a practically rigid body. Likewise, gripping maps to friction forces, and springiness maps to Hooke's law. This suggests that applicability can depend directly on our intuitive knowledge—knowledge that exists prior to any formal scientific training [42].

The in-pieces approach to modeling cognition has been used, among other things, to explain how experts reason about fractions and decimals [42], how students reason about forces in physics [9, 7, 17, 41], how students construct and evaluate algebraic representations of physical situations [27, 28], and how knowledge transfers across contexts [20, 47]. Because its starting assumption is that knowledge is fragmented, knowledge-in-pieces can account for wide variations of how people—particularly novices—use knowledge on a moment-to-moment basis. In other words, because it assumes knowledge is local, it can still explain the kinds of globally-inconsistent ways people might reason about physical situations [7]. As a framework, an in-pieces approach ultimately argues that models of concept replacement and good/bad criteria for knowledge should be supplanted by a learning model of alignment/refinement of prior knowledge and the consideration of knowledge as productive/unproductive.

Hammer and colleagues have worked to extend the in-pieces approach to explain how students' epistemological activity—how they orient toward knowledge and knowing in a context [20, 18, 10]. Specifically, those authors use two core theoretical constructs to explain students' stances toward knowledge and knowing:

- *Epistemological Resources* [20] are the epistemological equivalent of diSessa's phenomenological primitives (p-prims) [7]. Resources, the authors propose, are the atomic units involved in how people cognize about the source of knowledge, the nature of knowledge, and epistemological activities [19, 31].
- *Epistemological Frames* are the emergent result of subsets of resources acting in concert. Drawing from both Goffman's sociological notion of frame as structures of expectations [16] and subsequent work on framing in discourse [44], epistemological frames are a participant's local answer to the question "what is it [specifically, what knowledge activity] that's going on here" [16].

Resources and frames can interact in activity settings to produce larger-scale patterns called "epistemological coherences" [38] where evidence from data suggests that a network of discrete cognitive units can nonetheless give rise to stable cognition.

An example helps ground this in-pieces approach to epistemology. Russ, Coffey, Hammer, and Hutchison, describe the

situation where an elementary student reasons about why an empty juice box collapses when you suck on the straw [39]. One student gives what the authors deem to be an excellent mechanistic account of why the juice box collapses:

In explaining the phenomenon, Erin focuses on the role played by the air located inside the juice box. She describes the air inside as actively pushing out on all sides of the box holding them out and flat. When that air is removed from the box (by sucking it out through the straw), there is no longer anything pushing from the inside to hold the box out, so the sides cave in. We call this description of the phenomenon the “inside-pusher” model because of its focus on what happens inside the box. [39]

But, as the teacher seems to steer the discussion toward canonically correct vocabulary—in this case, “pressure”—and Erin clearly pulls back from her mechanistic reasoning, seems much more diffident, and claims that pressure is hard to explain. That example highlights the disconnect between doing science as knowing vocabulary and doing science as reasoning mechanistically. Moreover, it strikingly highlights that a student who by all accounts produced an excellent explanation of how pressure works was left nonetheless with the impression that pressure was hard to explain. In other words, in that moment what counted as knowing was using the word pressure. Erin’s mechanistic explanation—beautiful and complex as it was—didn’t use that magic word, and in a possible effort to push the class toward canonical behavior, the teacher shut down Erin’s reasoning.

4. CONCLUSION

In this article, we have argued for expanding our cognitive models beyond misconceptions in computing education research. We

1. Reanalyzed historical data through the perspective of productive student knowledge, building evidence in classic computing education research that we can recover novice’s productive knowledge without simply labeling it as non-canonical or wrong
2. Explored several examples that reveal both the adaptive nature of computing expertise and the power of non-misconception frameworks—namely distributed cognition and knowledge-in-pieces/symbolic forms.
3. Stressed that models to what counts as knowledge and knowing have both historical success in other fields and obvious implications for computing education

Ultimately, the modeling of cognition only or primarily as a set of privileged, canonical knowledge and how students do or don’t have it is eclipsing the other productive knowledge and ways of knowing that students do have. While useful, misconceptions can only take us so far in our efforts to understand and improve learning in computing education.

4.1 Acknowledgments

Blinded for review.

References

- [1] BAYMAN, P., AND MAYER, R. E. A diagnosis of beginning programmers' misconceptions of BASIC programming statements. *Communications of the ACM* 26, 9 (1983), 677–679.
- [2] BONAR, J., AND SOLOWAY, E. Uncovering principles of novice programming. *Proceedings of the 10th ACM SIGACT-SIGPLAN symposium on Principles of programming languages* (1983), 10–13. ACM ID: 567069.
- [3] BONAR, J., AND SOLOWAY, E. Preprogramming Knowledge: A Major Source of Misconceptions in Novice Programmers. *Human-Computer Interaction* 1, 2 (June 1985), 133.
- [4] CLANCY, M. Misconceptions and Attitudes that Interfere with Learning to Program. In *Computer Science Education Research*, S. Fincher and M. Petre, Eds. RoutledgeFalmer, London, UK, 2004, pp. 85–100.
- [5] DANIELSIEK, H., PAUL, W., AND VAHRENHOLD, J. Detecting and Understanding Students' Misconceptions Related to Algorithms and Data Structures. In *Proceedings of the 43rd ACM Technical Symposium on Computer Science Education* (New York, NY, USA, 2012), SIGCSE '12, ACM, pp. 21–26.
- [6] DISSA, A. A. Models of Computation. In *User centered system design: new perspectives on human-computer interaction*, D. A. Norman and S. W. Draper, Eds. L. Erlbaum Associates, Hillsdale, N.J, 1986, pp. 201–218.
- [7] DISSA, A. A. Toward an Epistemology of Physics. *Cognition and Instruction* 10, 2/3 (1993), 105–225. ArticleType: primary_article / Full publication date: 1993 / Copyright © 1993 Lawrence Erlbaum Associates (Taylor & Francis Group).
- [8] DISSA, A. A. Why "Conceptual Ecology" is a good idea. In *Reconsidering conceptual change: issues in theory and practice*, M. Limón and L. Mason, Eds. Kluwer Academic Publishers, Dordrecht ; Boston, 2002, pp. 29–60.
- [9] DISSA, A. A., AND SHERIN, B. L. What changes in conceptual change? *International Journal of Science Education* 20, 10 (1998), 1155–1191.
- [10] ELBY, A., AND HAMMER, D. On the substance of a sophisticated epistemology. *Science Education* 85, 5 (2001), 554–567.
- [11] FLEURY, A. E. Parameter passing: the rules the students construct. In *Proceedings of the twenty-second SIGCSE technical symposium on Computer science education* (New York, NY, USA, 1991), SIGCSE '91, ACM, pp. 283–286.
- [12] FLEURY, A. E. Student Beliefs about Pascal Programming. *Journal of Educational Computing Research* 9, 3 (Jan. 1993), 355–371.
- [13] FLEURY, A. E. Programming in Java: student-constructed rules. *SIGCSE Bull.* 32, 1 (Mar. 2000), 197–201.
- [14] GAL-EZER, J., AND TRAKHTENBROT, M. Identification and addressing reduction-related misconceptions. *Computer Science Education* 0, 0 (Apr. 2016), 1–15.
- [15] GAL-EZER, J., AND ZUR, E. The efficiency of algorithms–misconceptions. *Computers & Education* 42, 3 (2004), 215–226.
- [16] GOFFMAN, E. *Frame Analysis: An Essay on the Organization of Experience*. Harper & Row, New York, 1974.
- [17] HAMMER, D. Misconceptions or P-Prims: How May Alternative Perspectives of Cognitive Structure Influence Instructional Perceptions and Intentions? *Journal of the Learning Sciences* 5, 2 (1996), 97–127.
- [18] HAMMER, D., AND ELBY, A. On the form of a personal epistemology. In *Personal epistemology: The psychology of beliefs about knowledge and knowing*, B. K. Hofer and P. R. Pintrich, Eds. L. Erlbaum Associates, Mahwah, N.J, 2002, pp. 169–190.
- [19] HAMMER, D., AND ELBY, A. Tapping epistemological resources for learning physics. *The Journal of the Learning Sciences* 12, 1 (2003), 53–90.
- [20] HAMMER, D., ELBY, A., SCHERR, R. E., AND REDISH, E. F. Resources, framing, and transfer. In *Transfer of learning from a modern multidisciplinary perspective*, J. P. Mestre, Ed., Current perspectives on cognition, learning, and instruction. IAP, Greenwich, CT, 2005.
- [21] HAMMER, D., GUPTA, A., AND REDISH, E. F. On Static and Dynamic Intuitive Ontologies. *Journal of the Learning Sciences* 20, 1 (2011), 163–168.
- [22] HERMAN, G. L., KACZMARCZYK, L., LOUI, M. C., AND ZILLES, C. Proof by incomplete enumeration and other logical misconceptions. *Proceeding of the Fourth international Workshop on Computing Education Research* (2008), 59–70. ACM ID: 1404527.
- [23] HOLLAND, S., GRIFFITHS, R., AND WOODMAN, M. Avoiding object misconceptions. In *Proceedings of the twenty-eighth SIGCSE technical symposium on Computer science education* (New York, NY, USA, 1997), SIGCSE '97, ACM, pp. 131–134.
- [24] HUTCHINS, E. *Cognition in the Wild*. MIT Press, Cambridge, Mass, 1995. 381 pages.
- [25] HUTCHINS, E. How a cockpit remembers its speeds. *Cognitive Science* 19, 3 (1995), 265–288.
- [26] HUTCHINS, E. Distributed Cognition. *International Encyclopedia of the Social & Behavioral Sciences* (2000).
- [27] IZSÁK, A. "We Want a Statement That Is Always True": Criteria for Good Algebraic Representations and the Development of Modeling Knowledge. *Journal for Research in Mathematics Education* 34, 3 (May 2003), 191–227. ArticleType: primary_article / Full publication date: May, 2003 / Copyright © 2003 National Council of Teachers of Mathematics.

- [28] IZSÁK, A. Students' Coordination of Knowledge When Learning to Model Physical Situations. *Cognition & Instruction* 22, 1 (Mar. 2004), 81–128.
- [29] KACZMARCZYK, L., PETRICK, E. R., EAST, J. P., AND HERMAN, G. L. Identifying student misconceptions of programming. *Proceedings of the 41st ACM technical symposium on Computer science education* (2010), 107–111. ACM ID: 1734299.
- [30] KOLIKANT, Y. B.-D., AND MUSSAI, M. “So my program doesn't run!” Definition, origins, and practical expressions of students' (mis)conceptions of correctness. *Computer Science Education* 18, 2 (2008), 135.
- [31] LOUCA, L., ELBY, A., HAMMER, D., AND KAGEY, T. Epistemological Resources: Applying a New Epistemological Framework to Science Instruction. *Educational Psychologist* 39, 1 (2004), 57–68.
- [32] MINSKY, M. L. *The Society of Mind*. Simon and Schuster, New York, 1986.
- [33] PATITSAS, E., CRAIG, M., AND EASTERBROOK, S. On the Countably Many Misconceptions About #Hashtables (Abstract Only). In *Proceeding of the 44th ACM Technical Symposium on Computer Science Education* (New York, NY, USA, 2013), SIGCSE '13, ACM, pp. 739–739.
- [34] PAUL, W., AND VAHRENHOLD, J. Hunting High and Low: Instruments to Detect Misconceptions Related to Algorithms and Data Structures. In *Proceeding of the 44th ACM Technical Symposium on Computer Science Education* (New York, NY, USA, 2013), SIGCSE '13, ACM, pp. 29–34.
- [35] PEA, R. D. Language-independent conceptual” bugs” in novice programming. *Journal of Educational Computing Research* 2, 1 (1986), 25–36.
- [36] PEA, R. D., AND KURLAND, D. M. On the cognitive prerequisites of learning computer programming. Tech. Rep. Technical Report No. 18, National Institute of Education, 1983.
- [37] PEA, R. D., SOLOWAY, E., AND SPOHRER, J. The Buggy Path to the Development of Programming Expertise. *Focus on Learning Problems in Mathematics* 9, 1 (1987), 5–30.
- [38] ROSENBERG, S., HAMMER, D., AND PHELAN, J. Multiple Epistemological Coherences in an Eighth-Grade Discussion of the Rock Cycle. *Journal of the Learning Sciences* 15, 2 (2006), 261–292.
- [39] RUSS, R. S., COFFEY, J. E., HAMMER, D., AND HUTCHISON, P. Making classroom assessment more accountable to scientific reasoning: A case for attending to mechanistic thinking. *Science Education* 93, 5 (2008), 875–891.
- [40] SEPPÄLÄ, O., MALMI, L., AND KORHONEN, A. Observations on student misconceptions—A case study of the Build – Heap Algorithm. *Computer Science Education* 16, 3 (2006), 241–255.
- [41] SHERIN, B. L. How students understand physics equations. *Cognition and Instruction* 19, 4 (2001), 479–541.
- [42] SMITH, J. P., DISESSA, A. A., AND ROSCHELLE, J. Misconceptions Reconceived: A Constructivist Analysis of Knowledge in Transition. *The Journal of the Learning Sciences* 3, 2 (1993 - 1994), 115–163.
- [43] STEFIK, A., SIEBERT, S., SLATTERY, K., AND STEFIK, M. Toward Intuitive Programming Languages. In *2011 IEEE 19th International Conference on Program Comprehension (ICPC)* (2011), pp. 213–214.
- [44] TANNEN, D., Ed. *Framing in Discourse*. Oxford University Press, New York, 1993.
- [45] TRAKHTENBROT, M. Students Misconceptions in Analysis of Algorithmic and Computational Complexity of Problems. In *Proceedings of the 18th ACM Conference on Innovation and Technology in Computer Science Education* (New York, NY, USA, 2013), ITiCSE '13, ACM, pp. 353–354.
- [46] VANLEHN, K. *Mind Bugs: The Origins of Procedural Misconceptions*. Learning, development, and conceptual change. MIT Press, Cambridge, Mass, 1990.
- [47] WAGNER, J. F. Transfer in Pieces. *Cognition & Instruction* 24, 1 (Mar. 2006), 1–71.

[My Submissions](#)
[ICER 2016](#)
[News](#)
[EasyChair](#)

ICER 2016 Submission 116


[Update authors](#)

Submission information updates are disabled.

For all questions related to processing your submission you should contact the conference organizers. [Click here to see information about this conference.](#)

All **reviews sent to you** can be found at the bottom of this page.

Paper 116

Title:	Expanding Models of Cognition within Computing Education Research
Paper	
Track:	Research papers
Author keywords:	theory design cognition epistemology
EasyChair keyphrases:	computer science education (174), programming language (110), computing education research (95), student misconception (90), conceptual model (80), learning science (70), computing education (70), productive knowledge (70), symbolic form (60), canonical knowledge (50), conceptual metaphor (50), misconception research (50), symbol template (50), conceptual schema (50), airspeed indicator scale (47), science education (40), approach speed (40), non canonical (40), base change (40), distributed cognition (40), student non (40), assignment statement (40), knowing in computing learning (40)
Abstract:	This paper aims to expand our sense of what's possible in modeling cognition within computing education research. We argue that research approaches that privilege canonical knowledge do so at the expense of other productive knowledge and ways of knowing that students have. We explore applicable cognitive theory by showing how distributed cognition and symbolic forms can be a powerful framework for analysis in CSEd. Finally, we conclude with an exploration of epistemological concerns, arguing that a fundamental concern for our research community should be paying attention to what counts as knowledge and knowing in computing learning environments.
Time:	Apr 16, 06:03 GMT

Authors

first name	last name	email	country	organization	Web page	corresponding?
Brian	Danielak	briandaniela.k+easychairconference@gmail.com	United States	Michigan State University		✓
William	Doane	wdoane@ida.org	United States	Institute for Defense Analyses		

Reviews

Review 2

<i>Overall evaluation:</i>	4: (Borderline, lean to accept)
<i>Summary of paper:</i>	Authors argue against misconception-based models of cognition in CSEd research, suggesting that using other models would be more useful.
<i>Discussion of related work:</i>	3: (some references missing, or relationship to submission not clearly described)

<p><i>Theoretical basis for the paper:</i></p> <p><i>Use of theory:</i></p> <p><i>Research methodology:</i></p> <p><i>Exposition of research methods:</i></p> <p><i>Discussion of results and conclusions:</i></p>	<p>5: (clear and strong theoretical basis, well documented with citations and clearly applied in the research)</p> <p>Using Smith et al and Hammer et al provides good theoretical justification for their view on misconceptions -- appropriate cites.</p> <p>3: (research approach and methods well-suited for the research questions/hypotheses)</p> <p>1: (no empirical data collected)</p> <p>4: (good interpretation of findings; limitations considered)</p> <p>This is not a typical empirical ICER paper -- it argues a position (anti-misconception) rather effectively based on theory (primarily Smith et al) and a reinterpretation of Kaczmarczyk et al and Pea and Kurland. Its examples in 3.1 and 3.2 are quite compelling.</p>
<p><i>Methodology and empirical basis:</i></p>	<p>The main problem with this paper is that it overstates the role of misconception-based research in CSEd. For example: "a preponderance of research in computing education (CSEd) has modeled and continues to model cognition through misconceptions"; and "most cognitively-focused educational research in computer science treats students as having varied degrees of deficiency with respect to canonical knowledge". Where is the evidence for this? Certainly CSEd misconception papers are still being published, but relatively few involve misconceptions. Moreover, there are quite a few CSEd papers published that (at least implicitly) take an anti-misconception position (some references given in the summary below).</p>
<p><i>Contribution and relevance to the international computing education research field:</i></p>	<p>4: (a clear contribution to the field)</p>
<p><i>Significance of contributions/results:</i></p>	<p>I think this paper is a clear contribution; it makes a strong argument that misconceptions research has underlying flaws that suggest better approaches. I think it weakens its case by 1) overstating the importance of misconceptions work in CSEd, and 2) not mentioning threads of CSEd work that take alternative epistemological positions, for example the phenomenographic work on learning to program and the commonsense computing work (citations for these below).</p>
<p><i>Writing and expression:</i></p>	<p>4: (well written and expressed)</p> <p>This paper effectively shows the flaws with some of the misconceptions work in CSEd. It misstates the dominance of misconceptions research in CSEd however. It would improve the paper to include examples of work that takes alternative views of student conceptions; this would not weaken the author's conclusions about misconceptions, and would provide the readers with some models of how to use conceptions differently.</p> <p>References:</p> <p>Phenomenography:</p>
<p><i>Suggestions regarding the writing or other comments:</i></p>	<p>Booth, S. A. Learning to Program. A phenomenographic perspective. Number 89 in Göteborg Studies in Educational Science. Acta Universitatis Gothoburgensis, Göteborg, Sweden, 1992</p> <p>Eckerdal, A. and Thune, M. 2005. Novice Java programmers' conceptions of "object" and "class", and variation theory. In Proceedings of ITiCSE '05. 89-93.</p> <p>Commonsense Computing:</p> <p>Simon et al., 2006. Commonsense computing: what students know before we teach (episode 1: sorting). In Proceedings of the second international workshop on Computing education research (ICER '06). 29-40.</p> <p>Lewandowski et al, 2010, Commonsense understanding of concurrency: computing students and concert tickets, Communications of the ACM, v.53 n.7, July 2010</p>

Review 3

<i>Overall evaluation:</i>	2: (I would argue to reject this paper)
<i>Summary of paper:</i>	The paper is a position statement on using distributed cognition theory to analyze CS learning experiences. They argue that this lens is more useful than focusing on misconceptions, which has been a prevalent line of research in the field.
<i>Discussion of related work:</i>	3: (some references missing, or relationship to submission not clearly described)
<i>Theoretical basis for the paper:</i>	3: (there is a theory there, but its relevance to the research is vague) The paper presents an argument for examining student learning from a point of view other than misconceptions. The organization and exposition of the paper make it difficult to infer the grounding of the work. Much of the paper is reinterpretation of previous work using a new lens, presumably without access to primary data sources. A similar argument about distributed cognition was presented last year, and provides another example of how to present this work.
<i>Use of theory:</i>	Using Distributed Cognition Theory to Analyze Collaborative Computer Science Learning Elise Deitrick, R. Benjamin Shapiro, Matthew P. Ahrens, Rebecca Fiebrink, Paul D. Lehrman, Saad Farooq August 2015 ICER '15: Proceedings of the eleventh annual International Conference on International Computing Education Research Other references that seem to discuss similar matters are: Epistemological Pluralism and the Revaluation of the Concrete Sherry Turkle and Seymour Papert Versions of this article appeared in the Journal of Mathematical Behavior, Vol. 11, No.1, in March, 1992, pp. 3–33; Constructionism, I. Harel & S. Papert, Eds. (Ablex Publishing Corporation, 1991), pp.161–191; and SIGNS: Journal of Women in Culture and Society, Autumn 1990, Vol. 16 (1).
<i>Research methodology:</i>	2: (questionable choice of research approach and methods)
<i>Exposition of research methods:</i>	1: (no empirical data collected)
<i>Discussion of results and conclusions:</i>	2: (questionable interpretation of findings) This paper seems to have a confused message – one arguing about privileged ways of knowing and where remedies should be applied when problems are detected. And the other arguing for distributed cognition as a means to achieve this end. The arguments about privilege are not well supported and little reference is made to this field of research. The arguments about distributed cognition rely heavily on excerpted results from previous studies, reinterpreted with a distributed cognition or symbolic forms lens. In research, there is room for interpretation. But the work would be significantly improved with some primary data sources on which to do the analysis – perhaps contrasting a "misconception" based interpretation with that of distributed cognition. The authors seem to posit that all research done in misconceptions is grounded in behaviorist/empiricist views of knowledge and learning. It can certainly be argued that much of this work is actually rooted in the cognitive/rationalist view of learning. Clearly a situative view of learning, such as distributed cognition, provides a different perspective but it is not clear that the original work is a pejorative as presented.
<i>Methodology and empirical basis:</i>	
<i>Contribution and relevance to the international computing education research field:</i>	2: (no obvious contribution, but the promise of future value)
<i>Significance of contributions/results:</i>	The arguments for epistemological pluralism (Turkle & Papert) are the strongest aspect of this research. However, the discussion is muddled and does not make clear what additional contribution to the body of knowledge this provides. Perhaps using a situative learning theory to analyze student learning, could provide insights to researchers about how to address

Writing and expression:

Suggestions regarding the writing or other comments:

student difficulties. But the paper stopped short of that goal.

2: (very poorly written; unlikely that it can be improved enough)

Review 1

Overall evaluation:

2: (I would argue to reject this paper)

Summary of paper:

This paper presents a philosophical argument against using misconception modeling as a way to understand how students learn to program. There was a shift from early research in programming to now in which people were still learning a lot about the problem domain of programming from the mistakes that people made when programming, vs. now when we understand the domain is difficult and now ascribe mistakes to the cognitive abilities and skills of the programmers.

Discussion of related work:

4: (covers key related work; its relationship to submission is described, but could be extended further)

Theoretical basis for the paper:

4: (theoretical basis obvious, with some citations and argument for how it is applied in the research)

Use of theory:

The paper contains plenty of citations pointing back to the original studies of programming and programming languages.

Research methodology:

2: (questionable choice of research approach and methods)

Exposition of research methods:

1: (no empirical data collected)

Discussion of results and conclusions:

3: (plausible interpretation of findings)

Methodology and empirical basis:

The authors make their points through repeated code examples from prior research along with the contemporary interpretations of student cognition.

Contribution and relevance to the international computing education research field:

2: (no obvious contribution, but the promise of future value)

Significance of contributions/results:

The paper is presented as a philosophical argument with a few examples drawn from the literature to prove the authors' points. I find it difficult to follow the authors' arguments in this narrative form.

Writing and expression:

2: (very poorly written; unlikely that it can be improved enough)

Suggestions regarding the writing or other comments:

Unfortunately for me, the paper reads more like a long stream of consciousness, or perhaps a lecture format, in which the authors are speaking their arguments out loud. This makes it very difficult for me to follow in the paper format. I believe the authors could improve the work by shifting to a more top-down piecemeal style of writing in which each argument stands alone. The end could tie all the pieces together and present the main objective.



Sample review process

The template below is provided to help guide reviewers and authors during the reviewing process. You may also find it helpful to review the [tips for authors and reviewers](#).

NOTE: Actual reviews should be completed using the EasyChair system. Reviewers will receive a link via email when papers have been assigned.

Review Template

- 1. Overall evaluation
 - 6: I strongly support accepting this paper
 - 5: I would argue for accepting this paper
 - 4: Borderline, lean to accept
 - 3: Borderline, lean to reject
 - 2: I would argue to reject this paper
 - 1: I strongly recommend rejecting this paper
- 2. Reviewer's confidence
 - 3: (high)
 - 2: (medium)
 - 1: (low)
- 3. Summary of Paper (*)

Provide a 1-2 sentence summary of the work in your own words. We use this to verify that reviews are entered for the correct submission.
- 4. Discussion of related work
 - 5: all relevant work discussed and cited, and

- relationship to submission clearly and thoroughly described
- 4: covers key related work; its relationship to submission is described, but could be extended further
- 3: some references missing, or relationship to submission not clearly described
- 2: several important or key reference(s) missing, and relationship of references to submission not apparent
- 1: no discussion of related work
- 5a. Theoretical basis for the paper
 - 5: clear and strong theoretical basis, well documented with citations and clearly applied in the research
 - 4: theoretical basis obvious, with some citations and argument for how it is applied in the research
 - 3: there is a theory there, but its relevance to the research is vague
 - 2: maybe there's a theory there, but it is vague and has no clear relevance to the research
 - 1: no obvious theory being applied
- 5b. Use of Theory (*)

Discuss the appropriateness and quality of the theoretical framework for addressing the explored research topic.
- 6a. Research methodology
 - 3: research approach and methods well-suited for the research questions/hypotheses
 - 2: questionable choice of research approach and methods
 - 1: research approach and methods inappropriate for research objectives
- 6b. Exposition of research methods

4: data collected and analyzed; methods clear and thoroughly described.

3: data collected and analyzed, but some aspects unclearly described.

2: data collected and analyzed, but unclear or inadequate description.

1: no empirical data collected

- 6c. Discussion of results and conclusions

- 5: incisive interpretation of findings and limitations
- 4: good interpretation of findings; limitations considered
- 3: plausible interpretation of findings
- 2: questionable interpretation of findings
- 1: unjustifiable interpretation of findings

- 6d. Methodology and Empirical Basis (*)

Discuss the appropriateness and quality of the chosen methodology. Evaluate the way authors have applied the research, interpreted their findings and drawn conclusions. Comment on whether the findings have been examined in the context of related work and the limitations of the research.

- 7a. Contribution and its relevance to the international computing education research field

- 5: a major and significant contribution to the field that explicitly presents results in a manner directly applicable to international research contexts
- 4: a clear contribution to the field
- 3: minor contribution or contribution is bound to a local context, perhaps with the promise of more to come
- 2: no obvious contribution, but the promise of future value
- 1: contributes little or nothing to computing

education research

- 7b. Significance of Contributions/Results (*)

Make a case for the importance of this finding for our community (or indicate your views if you believe it not so important). We hopefully have a number of high-quality papers, and your input on what is important for our community matters.

- 8a. Writing and expression

- 5: exemplary writing that enhances the quality of the paper
- 4: well written and expressed
- 3: not well written, but could probably be made acceptable
- 2: very poorly written; unlikely that it can be improved enough
- 1: extremely poorly written; hard to understand

- 8b. Suggestions regarding the writing or other comments

If you would like to draw particular aspects of the writing to the authors' attention and/or make any further recommendations, please do so here. If you have any further references to recommend, please add them here.



Designed by **Elegant Themes** | Powered by **WordPress**